DISK INCLUDED

Programming in MACINTOSH® *and Think* PASCAL™

SECOND EDITION

Richard A. Rink • Vance B. Wisenbaker • Richard G. Vance

Programming in Macintosh® and THINK™ Pascal

Second Edition

Richard A. Rink
Professor of Computer Science
Eastern Kentucky University

Vance B. Wisenbaker
Dean of the College of Social and Behaviorial Sciences
Eastern Kentucky University

Richard G. Vance
Professor of Political Science and Chair, Department of Govenrment
Eastern Kentucky University

Prentice Hall, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Rink, Richard A.
Programming in Macintosh and Think Pascal / Richard A. Rink, Vance B. Wisenbaker, Richard G. Vance. — 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-093873-4
1. Macintosh (Computer)—Programming. 2. Pascal (Computer program language) I. Wisenbaker, Vance B. II. Vance, Richard G. III. Title.
QA76.8.M3R54 1995
005.265--dc20 94-33685
CIP

Publisher: Marcia Horton
Production Editor: Mona Pompili
Cover Designer: Violet Lake Studios
Copy Editor: Nick Murray
Production Coordinator: Bill Scazzero
Editorial Assistant: Delores Mars

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-093873-4

PRENTICE-HALL INTERNATIONAL (UK) LIMITED, *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL CANADA, INC., *Toronto*
PRENTICE-HALL HISPANOAMERICANA, S.A., *Mexico*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
SIMON & SCHUSTER ASIA PTE. LTD., *Singapore*
EDITORA PRENTICE-HALL DO BRASIL, LTDA., *Rio de Janeiro*

Dedicated to our children:
David and Nancy
Jon and Renan
Jana and David

Contents

PREFACE xv

Chapter One. Introduction to Macintosh Pascal

OBJECTIVES 1
1.1 COMPUTERS AND COMPUTER PROGRAMMING 1
1.2 BASIC COMPUTER ORGANIZATION AND THE MACINTOSH 2
1.3 USING THE MENUS ON YOUR MACINTOSH PASCAL DISK 5
1.4 THE PASCAL MENUS 9
1.4.1 The **Run** Menu 9
1.4.2 The **Pause** Menu 12
1.4.3 The **File** Menu 13
1.4.4 The **Search** and **Edit** Menus 14
1.4.5 The **Windows** Menu 16
1.5 USING THE PRINTER 17
SUMMARY 19
REVIEW QUESTIONS 20
PROGRAMMING EXERCISES 21

Chapter Two. Introduction to THINK Pascal

OBJECTIVES 30
2.1 INTRODUCTION TO THINK PASCAL AND THINK PASCAL WINDOWS 30
2.2 THE THINK PASCAL MENUS 37
2.2.1 The **File** Menus 37
2.2.2 The **Edit** Menu 39
2.2.3 The **Search** Menu 39
2.2.4 The **Project** Menu 42
2.2.5 The **Run** Menus 43
2.2.6 The **Debug** Menu 45
2.2.7 The **Windows** Menu 46
2.3 CREATING A SOURCE PROGRAM: MORE DETAIL 47
2.3.1 Editing a THINK Pascal Program 47
2.3.2 Using the Printer 48
2.3.3 Creating a Generic Project: A Helpful Shortcut 49
2.3.4 Creating an Instant Project 50
SUMMARY 51
REVIEW QUESTIONS 52
PROGRAMMING EXERCISES 52

Chapter Three. Constants, Variables, and Simple Input and Output

OBJECTIVES 60
3.1 PROBLEM SOLVING 60
3.1.1 Developing an Algorithm: An Example 62
3.2 THE FORMAT OF A PASCAL PROGRAM: ADDITIONAL DETAIL 64
3.3 THE CONCEPT OF A DATA OBJECT 67
3.3.1 Constants 67
3.3.2 Variables 70
3.4 INPUT AND OUTPUT 74
3.4.1 Output in a Pascal Program 74
3.4.2 Input in a Pascal Program 78
3.5 SIMPLE DATA TYPES IN MACINTOSH AND THINK PASCAL 83
3.5.1 `Real` Data Types 83
3.5.2 `Ordinal` Data Types: Standard 86
3.5.3 `Ordinal` Data Types: Nonstandard 91
3.5.4 `String` Types 95
3.6 TYPE DECLARATIONS 96
3.7 STANDARD PASCAL VERSUS THINK PASCAL 98
SUMMARY 99
REVIEW QUESTIONS 99
PROGRAMMING EXERCISES 102

Chapter Four. Basic Arithmetic Operations, Expressions, and Assignment Statements

OBJECTIVES 106
4.1 THE OPERAND AND THE OPERATOR 106
4.2 OPERATOR PRECEDENCE 113
4.3 EXPRESSIONS AND THE ASSIGNMENT STATEMENT 115
4.4 USING THE OBSERVE WINDOW TO TRACE A PROGRAM 119
4.5 ARITHMETIC FUNCTIONS 120
4.6 STANDARD PASCAL VERSUS THINK PASCAL 127
SUMMARY 128
REVIEW QUESTIONS 128
PROGRAMMING EXERCISES 132

Chapter Five. Basic Control Instructions for Looping and Branching

OBJECTIVES 138
5.1 PROBLEM ANALYSIS AND TRACING 138
5.2 CONTROL STRUCTURES FOR LOOPS 139
5.2.1 Pretest Iteration Loops 140
5.2.2 Post-test Iteration Loops 144
5.3 CONDITIONAL EXPRESSIONS 145
5.4 CONTROL STRUCTURES FOR BRANCHING 147
5.4.1 The One-Way Selector 147
5.4.2 The Two-Way Selector 148
5.4.3 Multiway Selection 154
5.5 NESTED LOOPS 159
5.6 BOOLEAN OPERATORS AND COMPOUND CONDITIONS 160
5.7 ITERATIONS REQUIRING SIMPLE COUNTERS 163
5.8 PROBLEM ANALYSIS: DEVELOPING AN ALGORITHM REQUIRING BRANCHING AND LOOPING CONSTRUCTS 165
5.9 STANDARD PASCAL VERSUS THINK PASCAL 171
SUMMARY 174
REVIEW QUESTIONS 175
PROGRAMMING EXERCISES 177

Chapter Six. Basic Graphic and Mouse Commands

OBJECTIVES 188
6.1 QUICKDRAW LIBRARY 188
6.2 DRAWING SIMPLE LINES 192
6.3 DRAWING SIMPLE GEOMETRIC PATTERNS 196
6.4 MOUSE CURSOR COMMANDS 206
6.5 SETTING THE SIZE AND DISPLAY OF TEXT AND DRAWING WINDOWS 209
6.6 SOME APPLICATIONS OF THE QUICKDRAW1 LIBRARY 211
6.7 USING COLOR GRAPHICS 234
SUMMARY 237

REVIEW QUESTIONS....239
PROGRAMMING EXERCISES....241

Chapter Seven. Procedures and Functions

OBJECTIVES....250
7.1 THE CONCEPT OF A PASCAL PROCEDURE....250
7.1.1 Definition of a Pascal Procedure....251
7.1.2 Passing Information to Formal Parameters: Value Parameters....259
7.1.3 Passing Information with Formal Parameters: Variable Parameters....264
7.2 PASCAL FUNCTIONS....277
7.3 GLOBAL VERSUS LOCAL IDENTIFIERS....282
7.4 FORWARD DECLARATIONS....285
7.5 PROCEDURAL AND FUNCTIONAL PARAMETERS....288
7.6 RECURSIVE FUNCTIONS AND PROCEDURES....294
7.7 DEVELOPING MODULAR PROGRAMS THROUGH STEP-WISE REFINEMENT...306
7.7.1 Abstraction 1....307
7.7.2 Abstraction 2....308
7.7.3 Abstraction 3....309
7.7.4 Abstraction 4....316
7.8 WHITE-BOX VERSUS BLACK-BOX TESTING....323
7.9 STANDARD PASCAL VERSUS THINK PASCAL....324
SUMMARY....326
REVIEW QUESTIONS....327
PROGRAMMING EXERCISES....332

Chapter Eight. Modularity: Building Programmer-Defined Libraries

OBJECTIVES....338
8.1 UNITS AND LIBRARIES IN THINK PASCAL: THE USES CLAUSE....338
8.2 BUILDING A THINK PASCAL PROJECT CONTAINING A PROGRAMMER-DEFINED UNIT....341
8.3 BUILDING THINK PASCAL LIBRARIES....349
8.4 USING THE USES CLAUSE WITHIN THE IMPLEMENTATION SECTION OF A UNIT....357
8.5 PREDEFINED LIBRARIES....360
8.6 APPLYING ADDED MODULARIZATION TO THE TUTOR SYSTEM....362
8.7 ALLOCATING A PROJECT AS AN APPLICATION....368
8.8 USING THE PROFILER OPTION TO COLLECT PROGRAM STATISTICS....370
8.9 USING LIGHTSBUG FOR VIEWING THE EXECUTION OF A PROGRAM....374
8.10 STANDARD PASCAL VERSUS THINK PASCAL....379
SUMMARY....379
REVIEW QUESTIONS....380
PROGRAMMING EXERCISES....383

Chapter Nine. Structured Data Types

OBJECTIVES 391
9.1 AN ARRAY AS A HOMOGENEOUS STRUCTURE....................................391
9.2 FORMAL PARAMETERS DECLARED AS ARRAY TYPES 404
9.3 MULTIDIMENSIONAL ARRAYS 408
9.4 CONCEPT OF AN ARRAY OF ARRAYS..416
9.5 APPLICATION OF ARRAYS: SORTING AND SEARCH ALGORITHMS.............421
9.5.1 Sorting Algorithms 421
9.5.2 Search Algorithms..425
9.6 AN INHOMOGENEOUS STRUCTURE: THE PASCAL RECORD.........................427
9.7 A STRUCTURE FOR CONTAINING A RANDOM SET OF ELEMENTS: THE PASCAL SET.......437
9.8 PACKED ARRAY OF CHARACTERS 446
9.9 STANDARD PASCAL VERSUS THINK PASCAL...448
SUMMARY 450
REVIEW QUESTIONS...451
PROGRAMMING EXERCISES 455

Chapter Ten. Files

OBJECTIVES 464
10.1 ADVANTAGES OF USING FILES 464
10.2 BASIC CONCEPT OF A PASCAL FILE..465
10.3 ACCESSING SEQUENTIAL FILES...467
10.4 MERGING A RECORD INTO A SEQUENTIAL FILE OF RECORDS 475
10.5 ACCESSING RANDOM FILES 481
10.6 APPLYING THE BINARY SEARCH ALGORITHM TO FILES 483
10.7 USING THE SPECIAL FUNCTION EOF..489
10.8 MERGING TWO FILES INTO A SEQUENTIAL FILE OF RECORDS....................490
10.9 TEXT FILES 495
10.10 REFERENCING DEVICES ON THE MACINTOSH AS FILE DEVICES..............499
10.11 AN APPLICATION: A SIMPLE DATABASE SYSTEM 501
10.12 STANDARD PASCAL VERSUS THINK PASCAL...522
SUMMARY 524
REVIEW QUESTIONS...525
PROGRAMMING EXERCISES 527

Chapter Eleven. Manipulation of Strings

OBJECTIVES 532
11.1 STRING TYPES IN MACINTOSH PASCAL 532
11.2 BASIC STRING PROCEDURES AND FUNCTIONS 535
11.3 PATTERN MATCHING AND OBJECT STRING REPLACEMENT........................544
11.4 SOME MISCELLANEOUS STRING ROUTINES FOR MACINTOSH AND THINK PASCAL.......550
11.5 EXAMPLE: EMULATING A `PRINT USING` STATEMENT...............................552
SUMMARY 566

REVIEW QUESTIONS 566
PROGRAMMING EXERCISES 567

Chapter Twelve. Pointers

OBJECTIVES 573
12.1 THE CONCEPT OF AN ABSTRACT DATA TYPE 573
12.2 POINTERS AND DYNAMIC VARIABLES 575
12.3 SPECIAL DATA STRUCTURES: LINKED LISTS, STACKS, QUEUES 585
12.4 APPLICATION OF POINTERS: BINARY TREES 602
12.5 HEURISTICS FOR WRITING RECURSIVE ROUTINES 612
12.6 ADDITIONAL COMMENTS ON NEW AND DISPOSE 614
12.7 MACINTOSH MEMORY MANAGER AND THE CONCEPT OF HANDLES 616
12.8 THINK PASCAL VERSUS STANDARD PASCAL 619
SUMMARY 619
REVIEW QUESTIONS 619
PROGRAMMING EXERCISES 622

Chapter Thirteen. Object-Oriented Programming in THINK Pascal

OBJECTIVES 628
13.1 INTRODUCTION TO THE TERM OOP 628
13.2 THE CONCEPT AND IMPLEMENTATION OF A CLASS 629
13.3 DECLARING AND USING OBJECTS 637
13.4 APPLYING THE RESERVED WORD INHERITED AND USING THE SELF PREFIX 642
13.5 USING TOBJECT AS THE ROOT CLASS 644
13.6 BUILDING A SIMPLE WINDOW SYSTEM 646
13.7 USING THE CLASS BROWSER AND LIGHTSBUG FOR VIEWING OBJECTS AND CLASSES 656
13.8 USING OBJECT PASCAL TO IMPLEMENT AN OBJECT LIST 658
13.9 SOFTWARE ISSUES IN USING OBJECT PASCAL 666
SUMMARY 668
REVIEW QUESTIONS 669
PROGRAMMING EXERCISES 670

Chapter Fourteen. QuickDraw library

OBJECTIVES 674
14.1 BASIS OF THE QUICKDRAW LIBRARY 674
14.2 MATHEMATICAL FOUNDATION OF THE QUICKDRAW LIBRARY 675
14.3 DEFINING A PORT USING GRAFPORT ROUTINES 682
14.4 DRAWING WITH POINTS, LINES, AND RECTANGLES 688
14.5 DRAWING WITH ARCS AND WEDGES 693
14.6 TEXT DRAWING ROUTINES 704
14.7 DRAWING WITH REGIONS AND POLYGONS 707
14.8 DRAWING PICTURES 713

14.9 TRANSFER MODES AND BIT TRANSFER OPERATIONS....716
14.10 SPECIAL GRAPHICAL ENTITIES: CURSORS AND PATTERNS....721
14.11 MAPPING AND SCALING POINTS, RECTANGLES, REGIONS, AND POLYGONS....729
SUMMARY....733
REVIEW QUESTIONS....734
PROGRAMMING EXERCISES....736

Appendices

APPENDIX A THINK AND MACINTOSH PASCAL RESERVED WORDS....744
APPENDIX B THE MACINTOSH CHARACTER SET....746
APPENDIX C INTRODUCTION TO THE SANE LIBRARY....748
APPENDIX D CREATING A MACINTOSH PASCAL APPLICATION....763
APPENDIX E USING LABELS AND THE PASCAL GOTO STATEMENT....767
APPENDIX F REFERENCES....770

Index....773

Preface

The primary objective of this textbook is to provide a solid introduction to the Pascal language for individuals using a Macintosh® System. The authors believe that either THINK Pascal® or Macintosh Pascal® can provide a stimulating environment for learning programming. These languages provide a combination of power, unique graphics qualities, and translators, providing easy access to the Macintosh's capabilities.

This book has been written to serve as both a useful introductory reference book and a self-study guide. As a textbook, it is directed toward the beginning courses on Pascal for computer science programs as well as special-interest courses on Pascal for Macintosh users. The level is elementary for Chapters 1 through 8, but shifts to intermediate in Chapters 9 through 14. An appendix is included for readers who want an introduction to SANE libraries. This book includes all the usual topics of a beginning textbook on the Pascal language, plus topics on both Macintosh and THINK Pascal that are special to the Macintosh computer. Topics included are pointers, abstract data types (ADT), files, string manipulation, procedures supported by both the QuickDraw and SANE libraries, programmer-defined units and programmer-defined libraries, profiling, the LightsBug debugger, and object-oriented programming.

This text reviews both THINK and Macintosh Pascal. For those who want to program in THINK Pascal, this edition includes three new chapters: 2, 8, and 13. Chapter 2 reviews using the THINK Pascal environment for building a project and implementing a Pascal program. Chapter 8 provides further applications of projects by discussing the implementation of program units and program libraries. In Chapter 8 we discuss the LightsBug debugger and examine the capability of THINK Pascal to profile the execution of a THINK Pascal program. Chapter 13 introduces object-oriented programming, an extension to standard Pascal. Chapters pertinent to readers using Macintosh Pascal are 1, 3-7, 9-12, and 14. Throughout the textbook, comments are given where Macintosh and THINK Pascal are different. Where relevant, a section has been included in some of the chapters comparing Standard Pascal with both THINK and Macintosh Pascal.

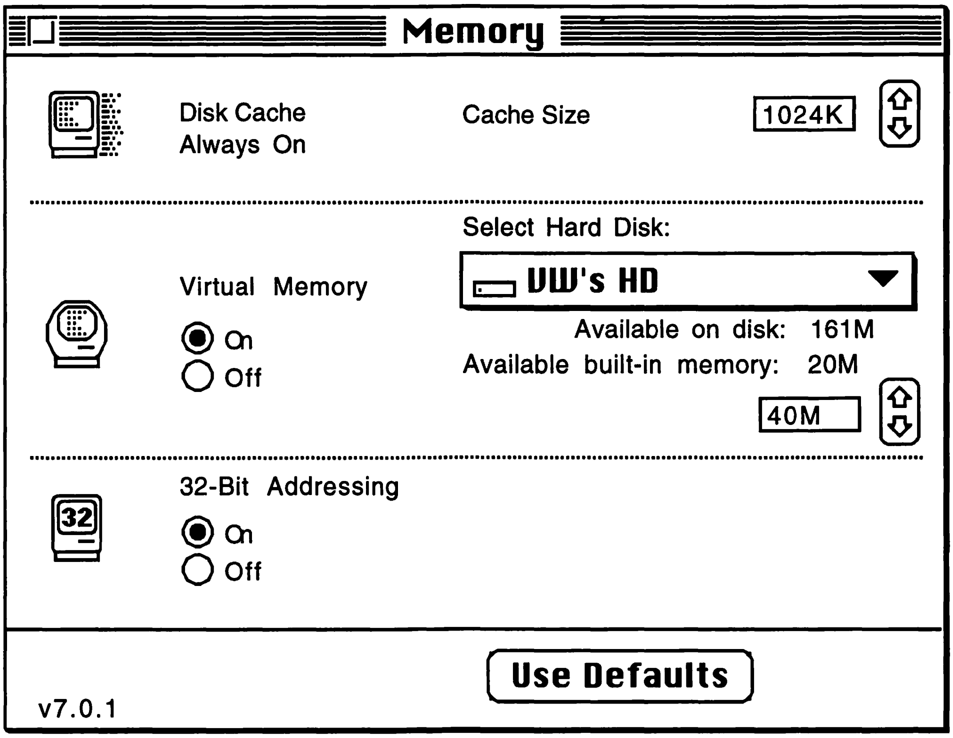
To use this book as a self-study guide, we suggest the following steps. First, read the chapter objectives and review questions before reading the first section of any chapter. Second, when you have completed reading a section within a chapter, return to the review questions, and see if you can answer any of them. Third, enter the examples in each section that are full programs, and see if they will execute. If you have any syntax or execution errors, check the program listing and make corrections. If you are successful at executing the example, try to modify it to perform one or more other actions, again testing your program to see if it will execute. After completing a chapter, again read the review questions. Once you feel that you have answered those questions correctly, choose several programming examples from the chapter, so that you can try to improve your ability to write THINK or Macintosh Pascal programs. Note that some of the programs listed in this book may appear different from a listing in the program window of the Macintosh computer. This was done so that the programs could be read more easily.

The Macintosh system is a true graphics machine. A person using either THINK or Macintosh Pascal can write computer programs in a high-level language that is able to interact with the ROM-based graphics capabilities of the Macintosh system. For example, a student in computer science using the Macintosh system and either THINK or Macintosh Pascal has a tool for understanding some of the basic graphics routines required of a workstation. By applying the QuickDraw Library routines, you can overlay several windows on the screen, only one of them being active at any time. In any active window, you can draw lines, curves, regions, pictures, polygons, or text, with control limited to actions of the mouse. Under System 7, Pascal programs written in THINK Pascal can be transformed into separate applications and executed simultaneously.

Each chapter introduces the basic principles of Pascal by defining the syntax, and through the presentation of complete THINK or Macintosh Pascal programs, shows the semantic actions. Having full Pascal programs allows you to enter the programs into a Macintosh system and observe the results of execution. You are not only able to verify the discussion in each chapter, but you can alter the example to test your own ideas. This reinforces learning of the material and builds self-confidence. As you enter a program, you can immediately correct errors and observe the results. All chapters include programming exercises in addition to the complete examples, providing practice with the material presented. Altogether there are more than 660 review questions and more than 240 programming exercises, which range in difficulty from simple to challenging. Stepwise refinement is applied in the development of all structured algorithms and programs. Throughout the book, structured programming concepts are stressed, and the later chapters employ structure charts to enable effective top-down programming design. Chapter 13 introduces a new way of thinking and programming: object-oriented programming.

We have chosen to write this book using THINK Pascal 4.0 and Macintosh Pascal 3.0, both published by THINK Technologies. While Macintosh Pascal is an interpreted version of Pascal, THINK Pascal is a full compiler. Both translators allow the use of windows for editing and composing programs, making either translator an excellent teaching language. In most instances, programs written in Macintosh Pascal are upward-compatible to THINK Pascal. While other compilers exist for writing Pascal programs, both THINK and Macintosh Pascal provide a simpler desk-top environment for editing and composing projects. All THINK and Macintosh Pascal programs in this text have been executed on Macintosh machines such as SE-30, IIcx, IIci, and IIsi, under either System 6.05, 7.0, 7.01, or 7.1. When using System 7, the authors recommend that Macintosh Pascal be executed only under 24-bit addressing with virtual memory turned off. To redefine these options, first double click on **Control Panels** from the Apple menu. This opens the Control Panels folder. Now, double click on the icon labeled

Memory. This opens the following dialog window for setting options such as cache size, virtual memory, and 32-bit addressing.



Memory

Disk Cache
Always On

Cache Size 1024K

Select Hard Disk:

Virtual Memory
[✓ On]
Off

VW's HD

Available on disk: 161M
Available built-in memory: 20M
40M

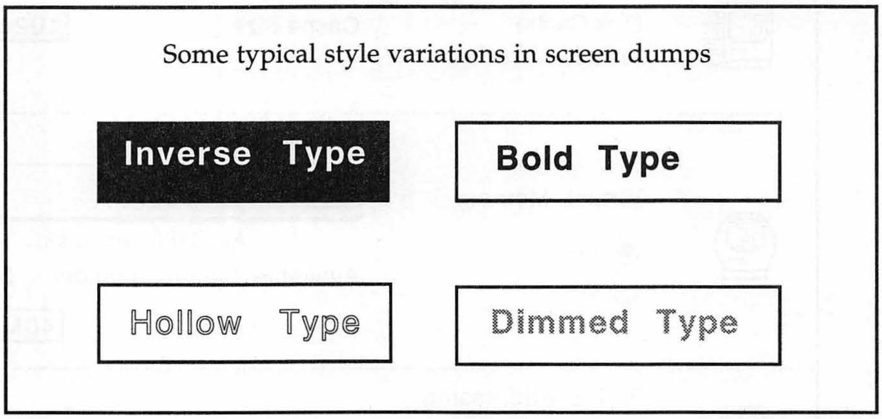
32-Bit Addressing
[✓ On]
Off

To turn off Virtual Memory and 32-bit addressing, click the Off buttons shown above.

With this dialog window open, click the Virtual Memory button to Off to turn off the virtual memory option, and click the 32-bit Addressing mode to Off to run the Macintosh system in 24-bit mode. On closing this window and on returning to the desktop, choose **Restart** from the **Special** menu. These changes take effect when the Macintosh machine restarts. While Macintosh Pascal can conveniently be executed from a floppy disk, the authors recommend that THINK Pascal be executed from a hard-disk drive with a minimum storage capacity of 20 megabytes on a Macintosh machine with at least 2 megabytes of RAM (4 megabytes if you are using the THINK Class Library) if you are using System 7. The complete THINK Pascal system requires 5.75 megabytes of disk space.

To help the reader distinguish special terms, we sometimes use a different font for them. For example, we use the `Courier` font for identifiers and the **Chicago** font for menu items. Programs and algorithms are shown in the Courier font.

Many of the figures in the text (including the one above) present what is called a screen dump: a printed representation of what actually appears on the screen. In screen dumps, the fonts used by the computer appear in the figure. Often these are the Geneva font, the Monaco font, or the **Chicago** font. Where special font styles occur in screen dumps, we have attempted to preserve them. In some cases, type will appear in an inverse mode: white type on a black background. In other cases, fonts will appear to be dimmed (gray, rather than black). In a dialog box, a dimmed font indicates that a given feature is not available under the current conditions. In a few cases, fonts will appear in hollow outline form. In the Edit windows of the Pascal software, this indicates a problem with the program line. In Pascal programs, bold is used to indicate reserved words such as **`begin`** and **`end`**. These variations are illustrated below.



We wish to acknowledge the help of several people who contributed to this project. First we wish to thank Dr. Marijo LeVan and Dr. Jerry LeVan for contributing ideas for examples as well as comments. We wish to thank Phyllis Gabbard, Suzanne Tipton, Lisa Rains, Pauline Coleman, and Kellie Lynn for their help in the preparation of the first edition of the manuscript. We would also like to thank the following reviewers of the first edition of this textbook: Herman Gollwitzer of Drexel University, Barry S. Marx of Wake Technical College, Denise Kiser of the University of California at Berkeley, Henry Etlinger of Rochester Institute of Technology, Christine Kay of DeVry Institute of Technology, and John Fleming, production editor with Prentice Hall, for the helpful suggestions they made in the preparation of the first edition.

In completing the second edition, we extend our appreciation to Marcia J. Horton, Editor-in-Chief, Prentice Hall/College Technical Division for her time and patience with us. We also extend our appreciation to Mona Pompili, our production editor, and Nicholas Murray, our copy editor.

Chapter 1

Introduction to Macintosh Pascal

OBJECTIVES

After completing Chapter 1, you will know the following:

1. What is meant by the terms *computer* and *computer program.*
2. The nature of the Macintosh Pascal windows and menus.
3. How a computer program is built and edited with Macintosh Pascal.
4. How to check, edit, execute, save, and print a Macintosh Pascal Program.

1.1 COMPUTERS AND COMPUTER PROGRAMMING

A computer is an automated machine that can process information and, in doing so, solve one or more problems. What type of information can a computer process? This usually depends on the type of problem to be solved and who is using the computer. For example, an engineer may use a computer to solve a mathematical problem, in which case the solution is a mathematical model of an engineering process. A business executive may use a computer to provide information on sales, gross profits, costs, and projected future profits and sales, with the computer providing information in both tabular and graphic formats. A student may use a computer to link with a database located thousands of miles away and submit questions to retrieve factual and deduced information from some set of information. In all these examples, the information to be processed by the computer may be numeric data (numbers) or symbolic data (characters) representing processes taking place at a particular instant of time.

When the computer is being used to solve a problem, the format or the steps necessary for solving the problem have previously been entered into the computer by means of a computer program. The word processor used for composing these paragraphs and printing each character, word, line, and paragraph of text, is a computer program. A computer program is also information to the computer, but information of a special type. We will define a *computer program* as an ordered set of instructions written in an *artificial* language (such as Macintosh Pascal or THINK Pascal). The instructions composing the computer program represent the solution to a problem. Computer languages are often referred to as *artificial languages* because they are more restrictive in the application of syntax (grammar) and semantic (meanings) rules, attempting to avoid the ambiguity found in natural languages. Another reason for the term *artificial* is the origin of these languages in a laboratory environment. Although at present Pascal may appear to be an extensive language, you will soon see that it is much simpler to understand than any of the natural languages such as English, German, French, or Russian. It is a highly structured language with specific syntax rules and semantic definitions for each of its commands. It is best if you begin by learning some of the easier syntax rules for composing data objects (nouns in Pascal), commands that provide action (sometimes referred to as *verbs*), and the rules necessary to form acceptable sentences that provide semantic meaning to the computer program.

A computer language allows an individual to communicate the solution to a problem to a computer. Why is this so important? First, computers like the Macintosh execute at a very primitive level, a machine level where all commands and information are binary. This means that they are composed of sequences of 1 and 0 bits. Writing programs at the machine level is possible, but only if the programmer has a thorough knowledge of the machine being used, has extensive programming experience, and pays considerable attention to the storage of information at specific locations (addresses) in the memory of the computer. An understanding of how basic operations are to be performed is also needed. Using a high-level language such as Pascal frees the programmer from such concerns and allows concentration on defining the steps for solving the problem. The binary machine instructions are generated when the Pascal program is compiled by the computer.

1.2 BASIC COMPUTER ORGANIZATION AND THE MACINTOSH

When describing a computer, it is useful to consider its basic organization from the viewpoint of input, output, memory, and execution. The Macintosh computer can be viewed as a von Neumann machine, named for the mathematician John von Neumann, who is credited with the stored-program concept of computing. As shown in Figure 1.1, a computer has five basic units: the central processor, memory, input, output, and the bus.

In a von Neumann machine, the function of the central processing unit (CPU) is to fetch an instruction of a computer program from main memory, decode this instruction, and then execute it; this same process is then repeated. During the execution of an instruction, the CPU is capable of performing basic arithmetic operations such as addition, subtraction, multiplication, and division; basic logical operations, such as a comparison of values; and the merging and masking of data. The CPU is also capable of controlling the actions of other units, such as memory, input, and output. For the Macintosh, the central processor is the MC68040 microprocessor, the MC68030 microprocessor, the MC68020 microprocessor, or the MC68000 microprocessor. In some Macintoshes there is also a co-processor, the MC68881/882.

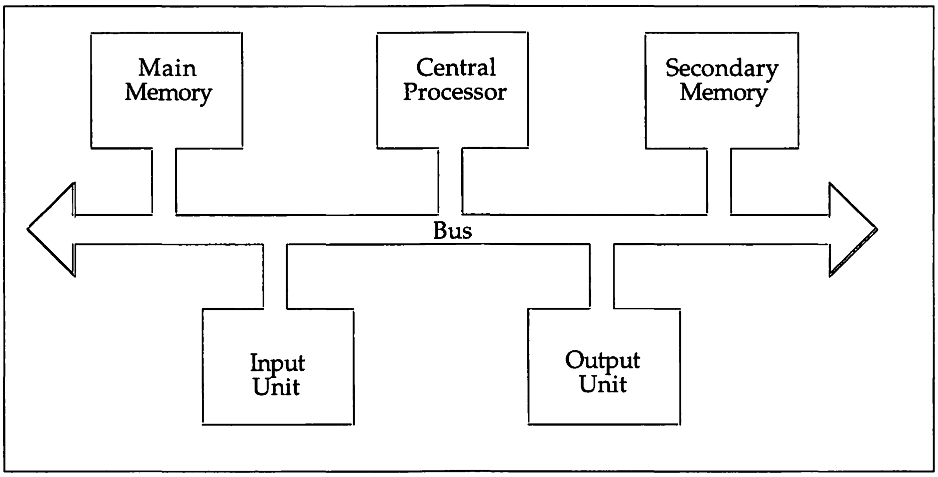


Figure 1.1 Basic organization of a computer.

Memory is divided into two distinct classes: primary, or main, memory and secondary memory for storage. The purpose of main memory is to store programs for performing one or more specific functions, as well as the data needed when a program is executed. For example, the word processing program used for writing this paragraph, Microsoft WORD, is contained in main memory along with the text as it is entered from the keyboard. Having both the program and data in main memory provides for fast response when entering additional words or making corrections to text previously entered. In the Macintosh computer, main memory is divided into two basic structures: random-access memory (RAM), and read-only memory (ROM). RAM offers flexibility, since it can both be written to and read from as long as the Macintosh remains powered. Once the Macintosh is turned off, any information stored in RAM is lost. ROM contains information needed by the Macintosh to "boot," or start up, when the power is turned on and also provides numerous instructions for interfacing with RAM memory, disk drives, the printer, windows, and menus. ROM is different from RAM in that it can be read from but not written to by the program being executed. ROM is also permanent; its contents do not vanish when the Macintosh is turned off.

Main memory can be viewed as a large, one-dimensional table of cells, each cell storing 1 byte of information. As Figures 1.2 and 1.3 show, each cell has a unique address represented by a whole number. Figure 1.2 was created with the software SUM II "SUM Tools," which allows the Macintosh user to map the memory of the Macintosh. Note the location of the word processing software (which takes up 1000 K, or 20% of the available RAM for this particular configuration).

The byte is composed of 8 individual bits, single binary digits, with each bit location containing either a 0 or a 1. Whereas humans are accustomed to communicating in natural languages, computers deal with a more fundamental level of information where all characters are represented by 1 and 0 bits. An individual character shown on the keyboard of the Macintosh can be stored in a byte location of memory.

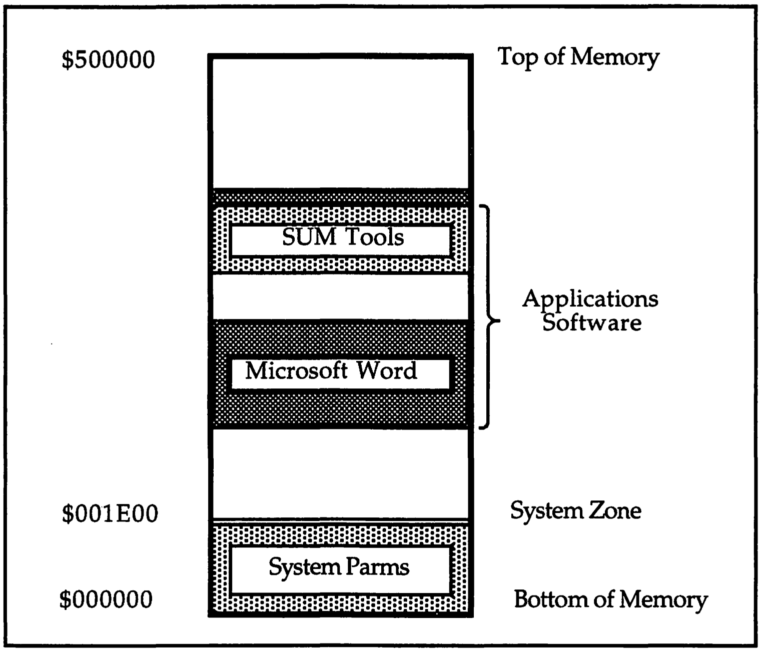


Figure 1.2 Representation of main memory for a Macintosh SE/30 computer with 5 megabytes of RAM.

By having a program that is capable of linking several bytes together, complex data such as alphabetic characters, numbers, or structures containing several different types of data can be stored. An example of the storage of an alphabetic character is shown in Figure 1.3.

Since RAM will lose the information stored in it after the Macintosh is turned off, secondary memory provides backup storage for programs and data. In the Macintosh, secondary memory is provided by a magnetic disk medium, such as a floppy diskette or hard disk. By inserting a properly formatted diskette into a disk drive, you can store information or retrieve information from what is referred to as a *file*. A file represents a collection of information stored on a diskette, analogous to the concept of storing a file of papers in a file cabinet. A file can be an application program, such as the Macintosh Pascal interpreter or the THINK Pascal compiler, a text document, a paint document, an application such as the Finder from the System Folder, and so forth. When the Macintosh's power is off, data that has been stored on a floppy diskette is not lost. This is one of the major advantages of secondary memory over primary memory.

Several different types of 3 1/2-inch floppy disk drives are used in the Macintosh. For the oldest Macintosh machines (128K and 512K), disk drives were limited to a maximum storage of 400K (409,600) bytes. The next group of Macintosh machines (Macintosh Plus and Macintosh SE) used floppy disk drives with 800K (819,200) bytes of storage capacity. The newer Macintoshes, beginning with the SE/30, use the "super drives" or "FDHD" drives which store 1.44M (1440K or 1,474,560) bytes. Hard disk drives, with the disks permanently encased, can provide 20 megabytes (the prefix *mega* means "million"), 40 megabytes, 80 megabytes, or more disk space. Hard disk drives provide faster access to the contents of files than the 3 1/2-inch floppy disk drives.

An input unit provides a means to communicate with the computer. For the Macintosh, both the keyboard and the mouse are input devices. The keyboard allows us to enter information such as text in a word processing document, while by clicking the mouse button, we can enter information such as our choice of options from a menu bar.

The output unit allows the computer to communicate with the user. In the case of the Macintosh, the screen is an output device. Another output device is a printer, such as a laser printer or a dot matrix printer. A disk drive can also serve as both an input and an output device.

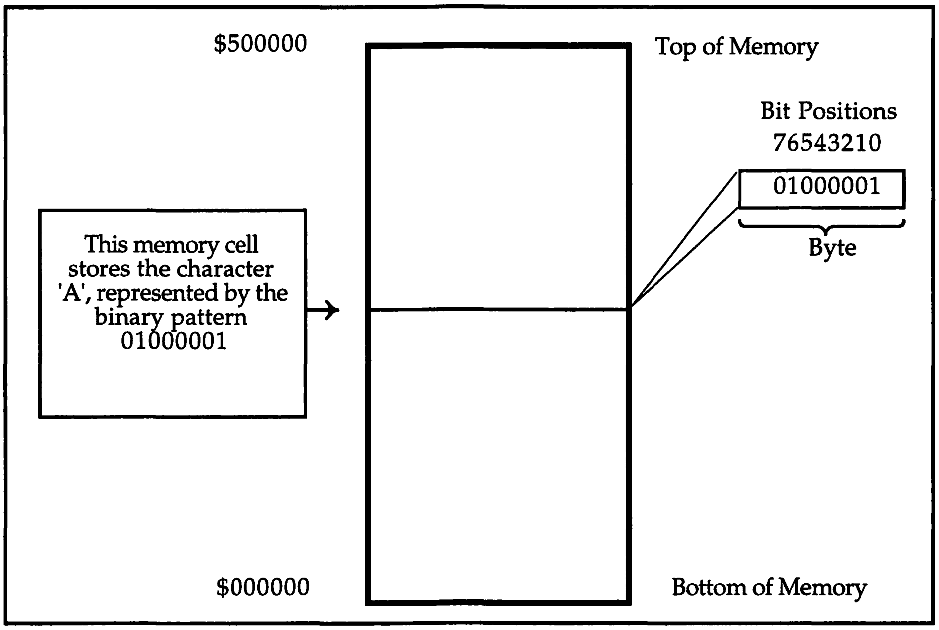

$500000
Top of Memory
Bit Positions
76543210
01000001
Byte
This memory cell stores the character 'A', represented by the binary pattern 01000001
$000000
Bottom of Memory

Figure 1.3 Representation of a single byte in the main memory of a Macintosh computer.

The bus provides an electrical path for connecting these components. Both commands and information can be passed along this path. In the Macintosh there are two types of bus structures, internal and external. The internal bus links the central processor with main memory, while the external bus provides a link between main memory and secondary memory as well as other input and output devices.

1.3 USING THE MENUS ON YOUR MACINTOSH PASCAL DISK

We assume that you are familiar with the general operation of the Macintosh and with its associated terminology. If you are not, read the manual that came with your Macintosh, or its equivalent, before continuing. To increase your dexterity in using the mouse, you may want to practice with a word processing application such as MacWrite or WORD and a painting program such as MacPaint or Super Paint. Once you have become comfortable with the routine operations of the Macintosh, you are ready to proceed to the Pascal software.

To use Macintosh Pascal, follow the instructions provided with your software to load Macintosh Pascal onto your hard disk. When this has been done, you will have a Macintosh Pascal folder similar to the one shown in Figure 1.4. The Macintosh Pascal software used by the authors and represented in the figure is version 3.0.

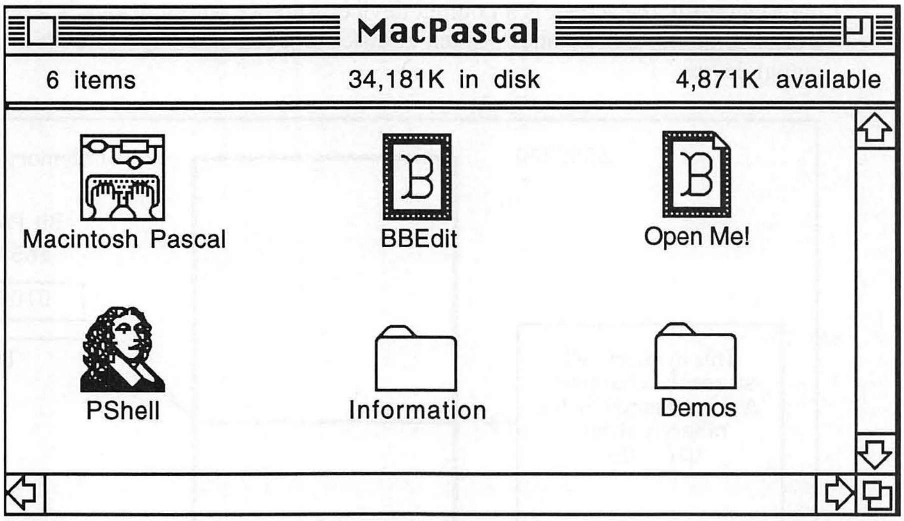
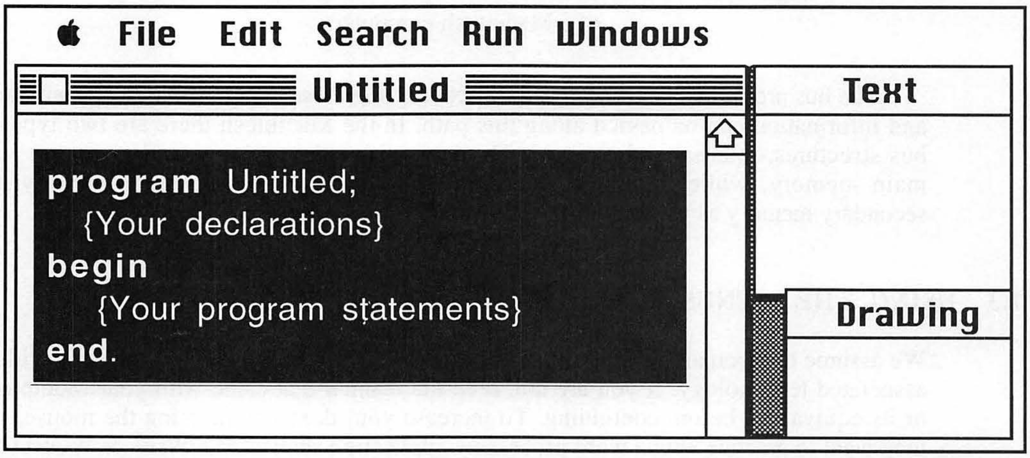


Figure 1.4 The Macintosh Pascal 3.0 folder.

To begin, double-click the Macintosh Pascal icon. After a brief delay, the three Macintosh Pascal windows shown in Figure 1.5 are displayed. The first of these is called the Program window, and it is here that your Pascal program is entered, edited, and displayed. Initially the Program window is labeled Untitled. Notice that when you move the mouse within this window, the pointer takes on the shape of an I-beam.



```
program Untitled;
  {Your declarations}
begin
  {Your program statements}
end.
```

Figure 1.5 The Macintosh Pascal windows and menus.

This means that you are allowed to insert text. The second window shown on the screen is called the Text window, and it is here that text output from your program is displayed. The Drawing window displays graphics output from your program. When you move the mouse from the Program window to either the Text window or the Drawing window, the pointer takes the shape of an arrow. Notice that in Figure 1.5 the Program window has horizontal bars at the top, indicating that it is currently the active window.

The best way to introduce yourself to Macintosh Pascal software is to type a short program and execute it. Observe that the Program window shown in Figure 1.5 contains a dark rectangle that includes some inverse print. This template is provided to help you begin writing your Pascal program. For example, each Pascal program must begin with a program heading line that has the following form:

```
program Name;
```

or

```
program Name (input, output);
```

where `Name` represents the title of the program. The name must begin with a letter, which can be followed by letters, digits, and/or the underscore character. A name may be up to 255 characters long (but will usually be considerably shorter than the limit). Uppercase and lowercase letters in a Macintosh Pascal program name are interpreted as being equivalent, allowing you to use either or a combination. All of these name rules are characteristic of what is known in Pascal as an *identifier*. An identifier serves to label constants, types, variables, program headings, and other entities that we will encounter later. Because of this widespread use of the identifier in Pascal, it is to your advantage to remember these rules.

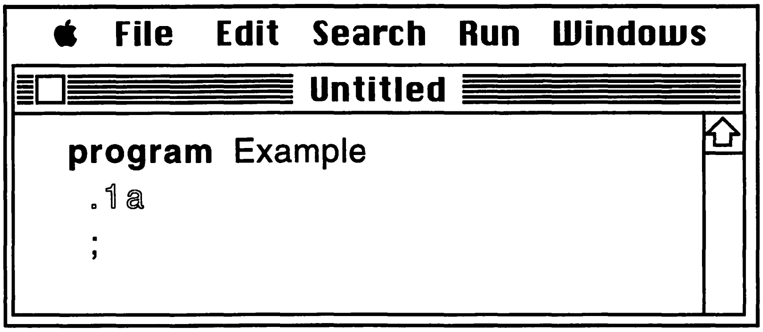
Returning to our program, first press the Backspace key to clear the Program window of the dark rectangle. (You may want to take advantage of this pretyped material later, but for now we will start with a blank window.) Next, type

```
program Example_1a;
```

You need not worry about the bold print for the word **program**; it is automatically added by Macintosh Pascal when you enter your line and press the Return key. In this example we are using `Example_1a` as the title for our program. This title conforms to all of the above rules for naming an identifier. Once you have inserted this line, type a semicolon to terminate the line of code. You may wonder what happens if a title is improperly entered. In Figure 1.6 the program title was typed `Example.1a` instead of `Example_1a`. Notice that when Macintosh Pascal detects such an error, it displays the incorrect character and the remaining characters to the right of the error in hollow outline type.

To continue with our example, type the word **begin**, and then press the Return key. The word **begin** indicates where the program is to begin execution. Next, type the remainder of the program, as shown below:

```
program Example_1a;
begin
   writeln('Sample text output');
end.
```



```
program Example
 .1a
 ;
```

Figure 1.6 A typing error in the name is quickly noted.

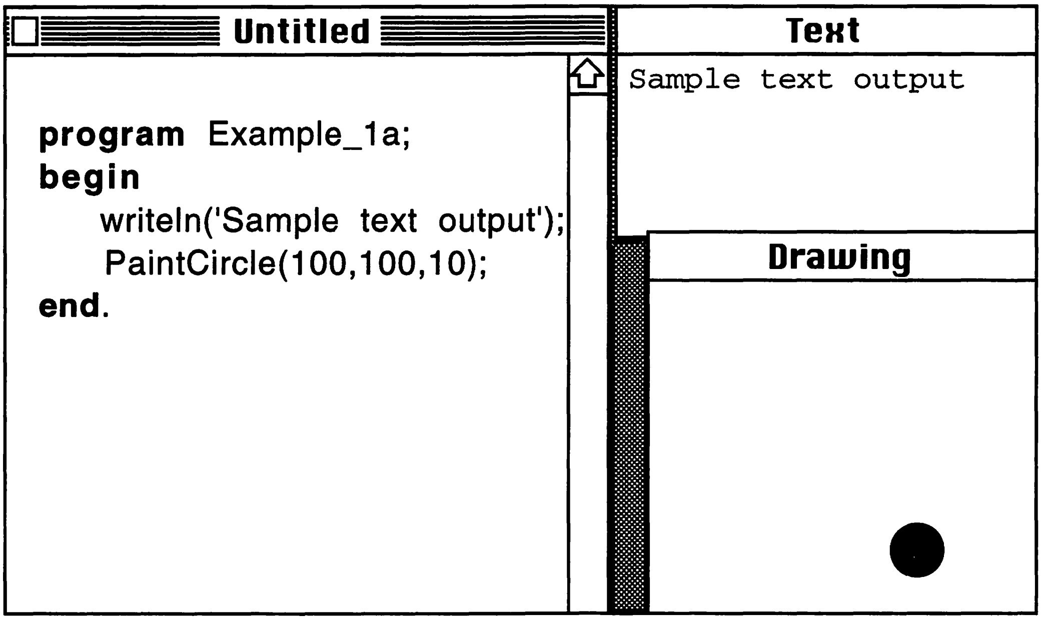
Be careful to include all the single quotation marks, periods, and semicolons. In Pascal these marks are critically important, and failure to pay careful attention to them can cause undesirable and sometimes strange things to happen to your programs. On the other hand, you need not be concerned with the indentation or the bold print, since Macintosh Pascal automatically takes care of these details.

It should be apparent that the only line of this program that actually causes any action is the third line, which results in the printed message `Sample text output` being displayed in the Text window. To verify this, pull down the **Run** menu, and select **Go**. (The Macintosh Pascal menus are shown in Figure 1.5.) The result is the immediate execution of the program.

If you would like to observe action in the Drawing window, add the command `PaintCircle` to your program as shown below:

```
program Example_1a;
begin
   writeln('Sample text output');
   PaintCircle(100,100,10);
end.
```

Again, pull down the **Run** menu and select **Go**. The result is shown in Figure 1.7. The third line of the program produces the printed message displayed in the Text window, and the fourth line causes the circle to be painted in the Drawing window. The numbers in the `PaintCircle` statement represent the coordinates of the center of the circle and the radius of the circle. Try changing these values and the wording enclosed in the single quotes of the `writeln` statement. A little experimentation with the programs presented throughout the text should help convince you of the generality, flexibility, and sheer fun of the computer program.



```
program Example_1a;
begin
   writeln('Sample text output');
   PaintCircle(100,100,10);
end.
```

```
Sample text output
```

Figure 1.7 The text and graphic output from Example_1a.

1.4 THE PASCAL MENUS

1.4.1 The **Run** Menu

Figure 1.8 shows the various Macintosh Pascal menus found on the menu bar. Initially you will be concerned with the **Run** menu. Using the mouse, move your pointer to the word **Run**, click (and hold) the mouse button, causing the **Run** menu to be displayed. Keeping the button pressed, drag the pointer down the menu. Notice that the command options[1] listed on the menu are highlighted as they are contacted by the pointer. To select a command option, release the button while that option is highlighted. For example, to execute a Macintosh Pascal program listed in the Program window, pull down the menu, and select **Go**. This command option initiates the execution of the program. Another useful command option on the **Run** menu is the **Step** command. With this option you can step through a program by executing one line at a time. This feature is useful when debugging a program, that is, trying to understand why a program is not properly executing and where it may be wrong. Select **Step** from the menu, and then observe that only the first line of the program is executed. Figure 1.9 shows the result of this first step. As you can see, a small hand appears in the Program window pointing to the left of the next executable line of code. Select **Step** again from the **Run** menu to execute this line. Continue this process until all the lines have been executed. If you wish, you can speed up this process by using the command option **Step-Step**. This causes the program lines to be executed in sequence, with a brief delay between each pair of lines.

[1] In order to avoid confusion, we generally refer to choices listed on the Macintosh and THINK Pascal menus as *command options*.

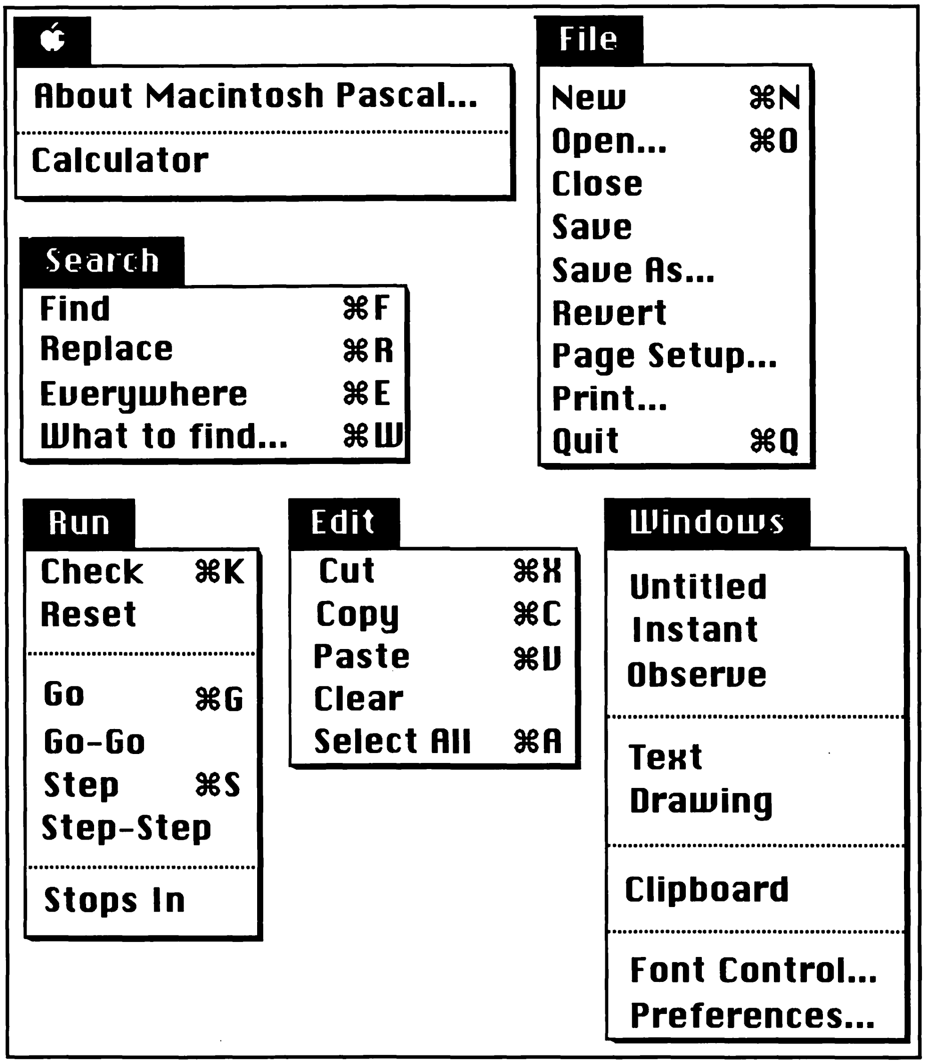
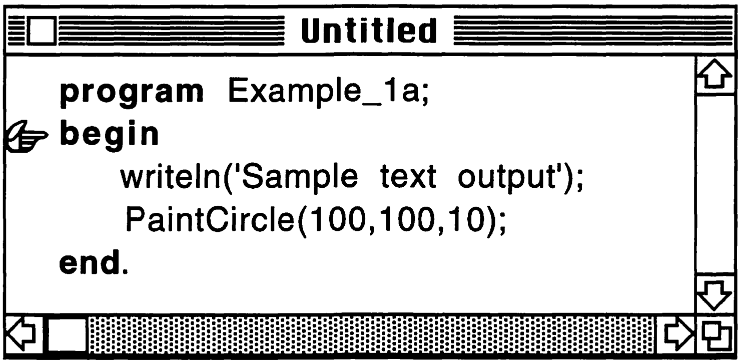


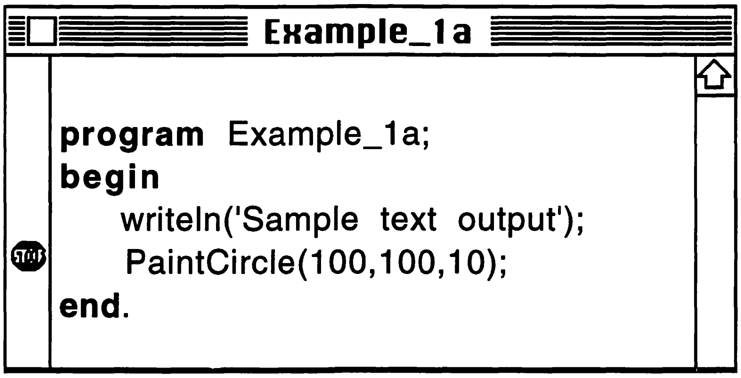
Figure 1.8 The Macintosh Pascal menus.

Another command option is **Stops In**. When this option has been selected, you may insert a stop to the left of any program line, preventing that line from being executed. To insert a stop, activate **Stops In**, move the pointer to the left of a line you do not wish to be executed, and click the mouse. This causes a stop sign to appear, as shown in Figure 1.10. By selecting the command option **Go**, you can execute the program up to the chosen line. To continue execution of a program after a stop has been encountered, select **Go** again. Figure 1.10 shows the screen after a stop has been inserted to the left of line 4 in `Example_1a`. With the stop in this position, execution of the program will cause the third line to be executed, displaying the message in the Text window, but leaving the PaintCircle command unexecuted. Selecting the **Go** option for a second time will cause the execution of the program to continue from the point where the stop appears. In the example, line 4 will be executed, painting the circle in the Drawing window.



```
program Example_1a;
begin
   writeln('Sample text output');
   PaintCircle(100,100,10);
end.
```

Figure 1.9 Stepping through a program.

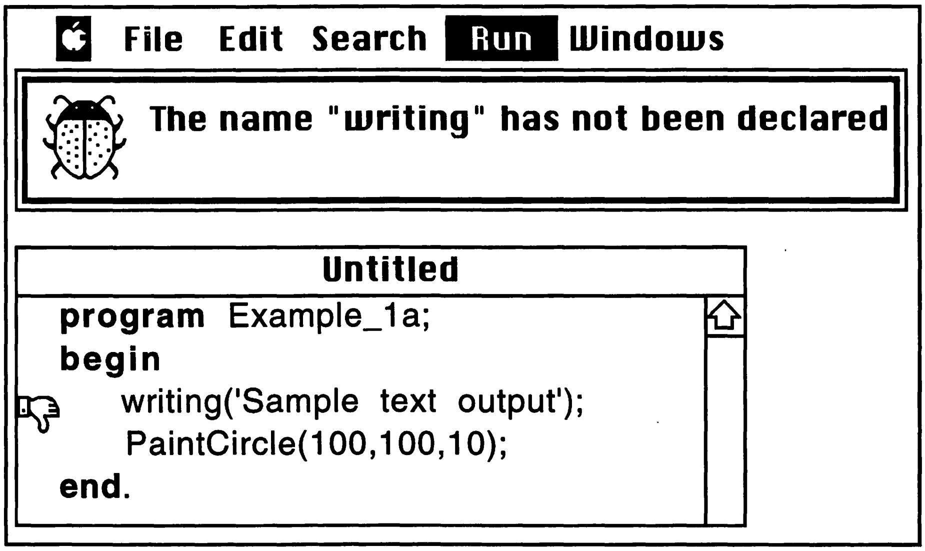


```
program Example_1a;
begin
   writeln('Sample text output');
   PaintCircle(100,100,10);
end.
```

Figure 1.10 Inserting stops in a program.

Stops can be disabled by selecting the **Stops Out** command option on the menu. This leaves all of the stops in place but renders them inoperative and invisible on the screen. When you select **Stops In**, the stops are again displayed and again become operational.

The other command options under the **Run** menu are **Check**, **Reset**, and **Go-Go**. The command option **Check** allows you to have the computer check your program for syntax errors without actual execution. In this mode, each line is examined to determine if it is a valid Pascal statement. For example, consider the program `Example_1a` in Figure 1.11. Notice that the command `writeln` has been replaced by the word *writing*. With this alteration in the program, selection of the command option **Check** results in the appearance of a dialog box at the top of the screen, indicating that a bug has been found in the program. This dialog box with its large bug represents the standard form in Macintosh Pascal for reporting any error discovered while checking or executing a line of code. Along with this error message there also appears a hand with its thumb pointing downward, positioned to the left of the line containing the bug. In the example, the error results from the illegal command `writing`. Since Macintosh Pascal will keep you from performing any other actions until the dialog box has been closed, you must click anywhere within the border of the dialog box to close it. Once the dialog window closes, you are free to correct the error or to perform other command options from the menu bar.



The name "writing" has not been declared

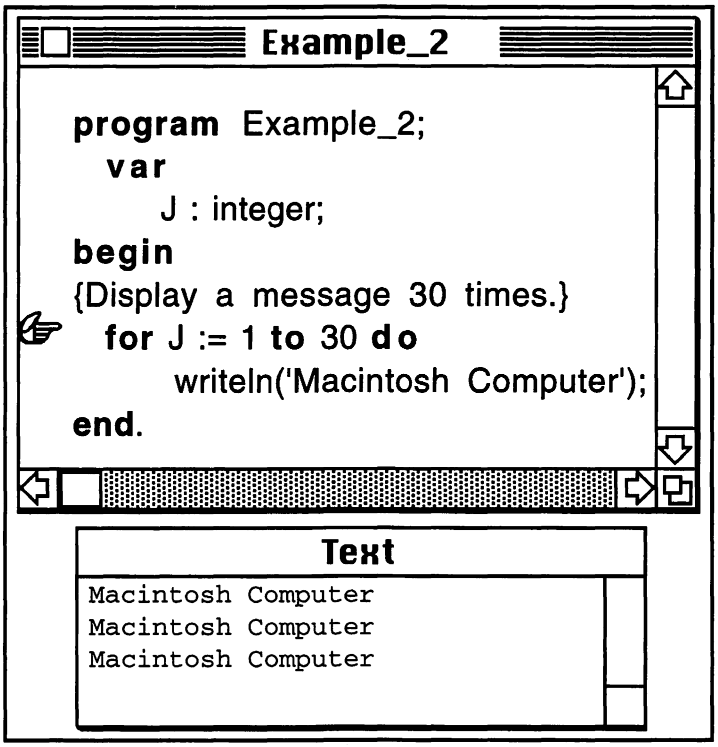
```
program Example_1a;
begin
    writing('Sample text output');
    PaintCircle(100,100,10);
end.
```

Figure 1.11 Using the **Check** command option to locate a problem.

The **Reset** command option allows you to return your program to its initial state. That is, it is returned to the state it takes prior to execution, clearing the Text and Drawing windows. You may find the **Reset** command option useful in connection with the **Step** command option. It allows you to return to the beginning of a program without completing execution of all its lines. In large programs this feature can be a significant timesaving device. Finally, the **Go-Go** command option is operational only when the **Stops In** command option has been selected. By selecting **Go-Go** you can step through a program with only a brief delay at each stop (as compared to a complete halt in execution). This command option, along with a well-placed set of stops, will allow you to see your program execute in slow motion.

1.4.2 The Pause Menu

Consider a second program called `Example_2`, shown in Figure 1.12. Enter this program as shown, and then select **Check** to check for any incorrect syntax. If no errors exist, select **Go**, and watch the menu bar while the program executes. During execution, a new menu command appears: **Pause**. By pointing the arrow at **Pause** and pressing the mouse button, you can temporarily halt execution of the program. Releasing the mouse button while the arrow remains on **Pause** allows the program to continue execution. Dragging the arrow down while the pause is in effect will activate the option **Halt.** Selection of this option causes execution of the program to stop at whatever line it has reached. Execution of the program after a halt can continue only by selection of one of the following command options: **Go**, **Step**, or **Step-Step**. Some Macintosh Pascal programs cannot or will not end their execution. If this is the case, the option **Halt** from the **Pause** menu may be the only way to stop execution (short of turning off the Macintosh).



```
program Example_2;
  var
     J : integer;
begin
{Display a message 30 times.}
  for J := 1 to 30 do
      writeln('Macintosh Computer');
end.
```

```
Macintosh Computer
Macintosh Computer
Macintosh Computer
```

Figure 1.12 The program `Example_2` and its output.

1.4.3 The **File** Menu

To save your Macintosh Pascal program, pull down the **File** menu, and select **Save As....** A dialog box like that shown in Figure 1.13 will appear on the screen. Notice that you are asked to assign a name to the file. Macintosh file names are extremely flexible, so you can use almost any name you choose, as long as you can type it in the space provided in the dialog box. (The Macintosh file system allows a maximum of 31 characters.) Illegal names will be rejected by the computer. You have the option to save the program as a text file (file of characters as seen on the screen), as an object file (file stored in machine code), or as an application by clicking on one of the three small buttons within the dialog box. See Appendix D for more on creating application programs in Macintosh Pascal.

Once your program has been saved, the name for the file will be displayed at the top of the Program window. After this, you may use the **Save** command option to update your file whenever you make changes in the Program window. It is in your best interest to select the **Save** option fairly often when you are writing or editing a program. Once you have saved your program, you may end your work session by closing your file (select

Close from the **File** menu, or click the close box) and then quitting (select **Quit** from the **File** menu).

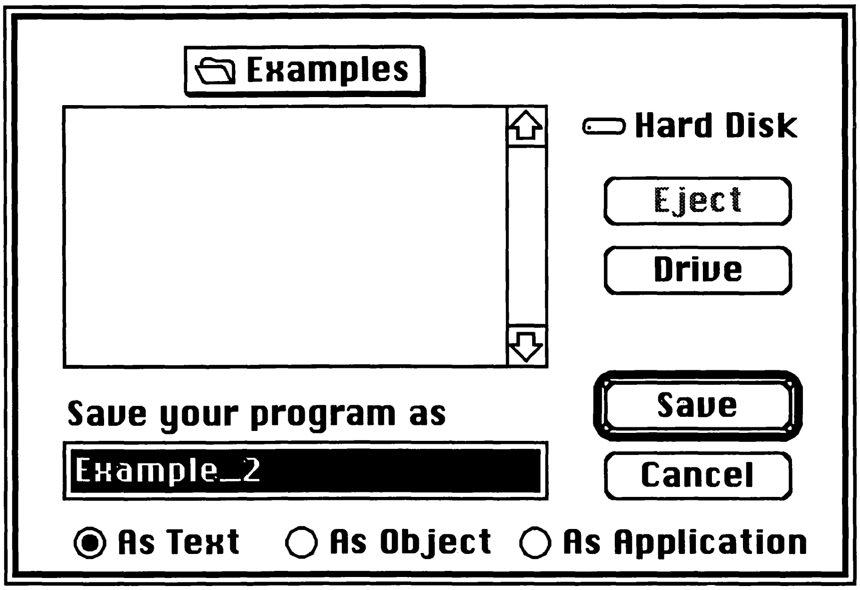
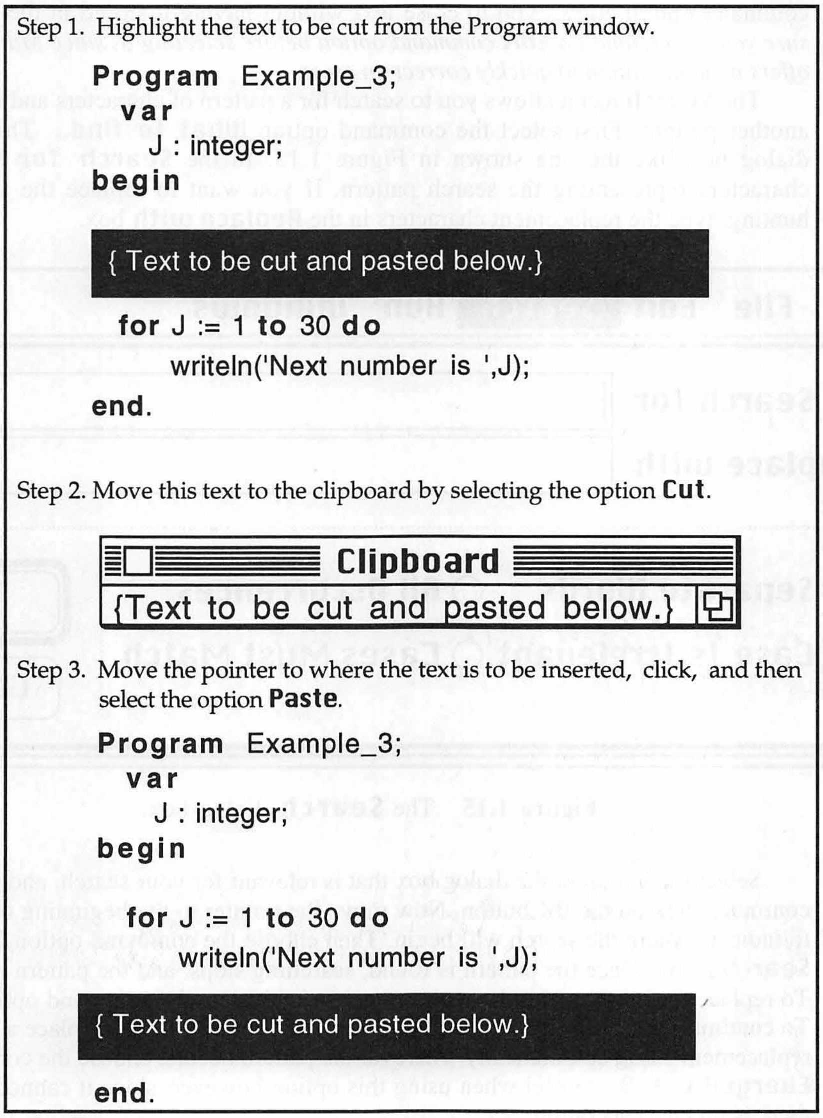


Figure 1.13 Saving a Macintosh Pascal program file.

The **Print...** and **Page Setup...** command options of the **File** menu are used when you want printed output. These options will be explained in Section 1.5. The **New** command option is used to provide an untitled Program window for a new program. If you select this option when you already have a program in the Program window, the existing program will automatically be closed when the new untitled window is opened. If you have changed the existing program since your last file, you will be asked if you want to save the changes before the window is closed. Finally, **Revert** allows you to discard changes made in the Pascal program shown in the Program window and revert to the last version of the program saved on the disk.

1.4.4 The Search and Edit Menus

Your Macintosh Pascal program can be edited in the Program window by using the **Cut**, **Copy**, and **Paste** command options under the **Edit** menu. To insert text anywhere in your program, move the I-beam pointer in the Program window to the point at which the text is to be inserted, click the mouse button to position the pointer, and begin typing. To remove text, first position the pointer either to the right or to the left of the text. Then drag the pointer across the text while pressing down on the mouse button. Release the button when you have highlighted in black the text to be removed. This process is called *selecting* the text. Figure 1.14 shows an example of text that has been selected.



Step 1. Highlight the text to be cut from the Program window.

```
Program  Example_3;
  var
     J : integer;
begin

{ Text to be cut and pasted below.}

  for J := 1 to 30 do
      writeln('Next number is ',J);
end.
```

Step 2. Move this text to the clipboard by selecting the option **Cut**.

```
{Text to be cut and pasted below.}
```

Step 3. Move the pointer to where the text is to be inserted, click, and then select the option **Paste**.

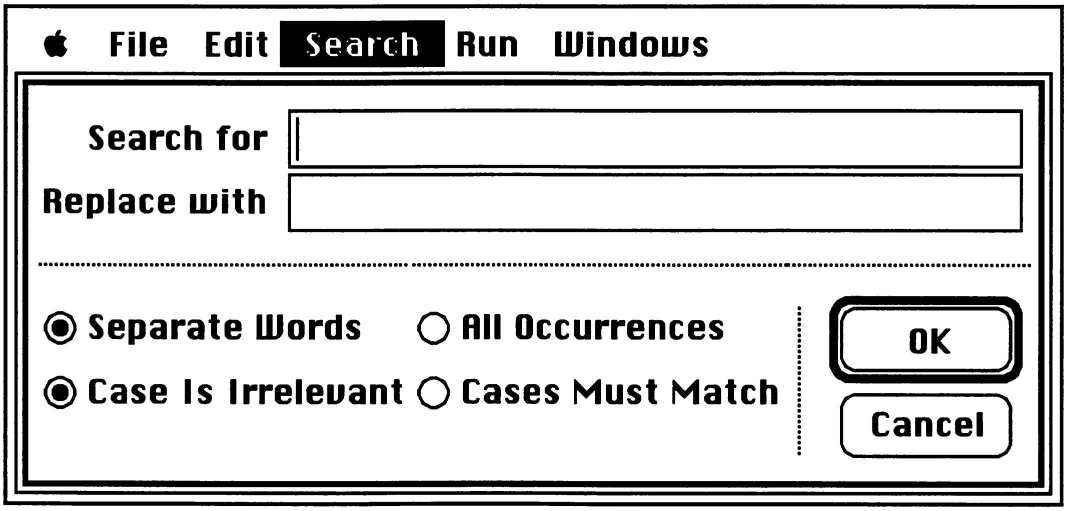
```
Program  Example_3;
  var
     J : integer;
begin

  for J := 1 to 30 do
      writeln('Next number is ',J);

{ Text to be cut and pasted below.}
end.
```

Figure 1.14 Using the command options **Cut**, **Copy**, and **Paste** from the **Edit** menu.

You may remove selected text by pressing the delete key, typing new text, or by electing the command option **Clear** or **Cut**. When the command option **Cut** is used, the text that is removed is stored temporarily on the clipboard. Text on the clipboard can then be moved to another position in the Program window by moving the I-beam pointer to the point where the material is to be inserted. After clicking the mouse, select the **Paste** command option from the **Edit** menu to paste text that is presently in the clipboard.

The command option **Copy** is similar to **Cut**, except that it allows you to copy selected text to the clipboard without erasing it from the Program window. The **Clear** command option allows you to erase text without having it stored in the clipboard. *Be sure you understand an* **Edit** *command option before selecting it, since Macintosh Pascal offers no undo option to quickly correct an error.*

The **Search** menu allows you to search for a pattern of characters and replace it with another pattern. First select the command option **What to find...** This brings up a dialog box like the one shown in Figure 1.15. In the **Search for** box, type the characters representing the search pattern. If you want to replace the string you are hunting, type the replacement characters in the **Replace with** box.



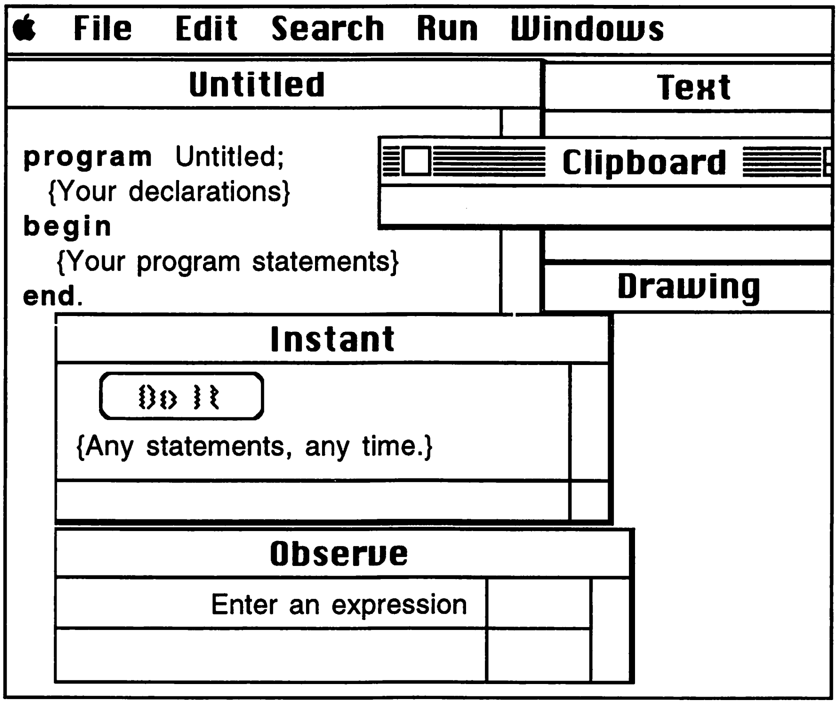
Search for
Replace with
[✓ Separate Words] All Occurrences
[✓ Case Is Irrelevant] Cases Must Match

Figure 1.15 The **Search** dialog box.

Select the button in the dialog box that is relevant for your search, and if you wish to continue, click on the **OK** button. Now move the pointer to the beginning of the program to indicate where the search will begin. Then choose the command option **Find** from the **Search** menu. Once the pattern is found, searching stops, and the pattern is highlighted. To replace the pattern with the replacement string, select the command option **Replace**. To continue searching, again select the command option **Find**. To replace a pattern with a replacement string automatically wherever the pattern occurs, choose the command option **Everywhere**. Be careful when using this option however, since it cannot be undone—there is no undo operation.

1.4.5 The Windows Menu

Three of the options available under the **Windows** menu have already been discussed: the Text window, the Drawing window, and the Program window. Figure 1.16 shows these as well as the other three windows that are available to you. The first of these is the Instant window. In this window you may enter a single Pascal statement and then execute it immediately by clicking its **Do It** button. You may move statements from your Program window to the Instant window by way of the clipboard. The Observe window allows you to display the values of selected expressions at critical points in your program (whenever execution ceases, whether at the end or at a stop you have inserted). This feature, along with the **Stops In** option, allows you to carefully study the execution of your program. The Clipboard window allows you to view the current contents of the clipboard.



```
program Untitled;
  {Your declarations}
begin
  {Your program statements}
end.
```

```
{Any statements, any time.}
```

```
Enter an expression
```

Figure 1.16 The Macintosh Pascal windows.

There are two additional options on the **Windows** menu. The **Font Control...** option allows you to select a font and font size for either the Program window or the Text window. Figure 1.17 shows the dialog box produced when this option is selected. The other option under the Windows menu will be explained in Section 1.5.

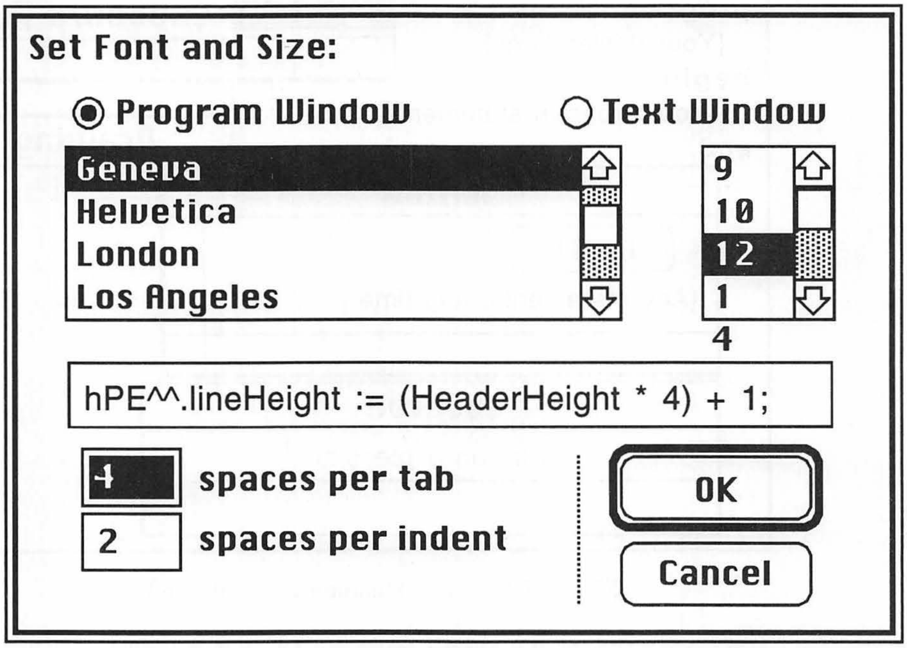
1.5 USING THE PRINTER

You can use your printer to make a paper copy (hard copy) of the material displayed in any window. For example, if you would like a hard-copy listing of a Pascal program, activate the Program window, pull down the **File** menu, and select **Print...** A dialog box similar to that shown in Figure 1.18[2] will offer you the option for choosing the quality of print, the page range, the number of copies, and paper feed (continuous or cut-sheet). After you enter this information, the printing will proceed. The **Page Setup** dialog box, shown in Figure 1.19, allows you further control over your printed output. This includes the size of paper, the orientation of the paper. reduction or enlargement, and a variety of

[2] The exact nature of this dialog box will vary depending on the printer and the system software in use.

other choices. Since both are standard Macintosh dialog boxes, it is not critical that we discuss every alternative.

By selecting the **Preferences...** option of the **Windows** menu, you may direct the output of your program to the printer as well as the screen. Figure 1.20 shows the dialog box when this command option is selected. To direct your program to your printer, click the "Output also to the Printer" box. When the program executes, whatever is directed to the Text window is also directed to the printer. Notice that this option allows you to send your output to a file. We will discuss files later in the book.



Set Font and Size:

[✓ Program Window] Text Window

Geneva, Helvetica, London, Los Angeles — selected: Geneva

9, 10, 12, 14 — selected: 12

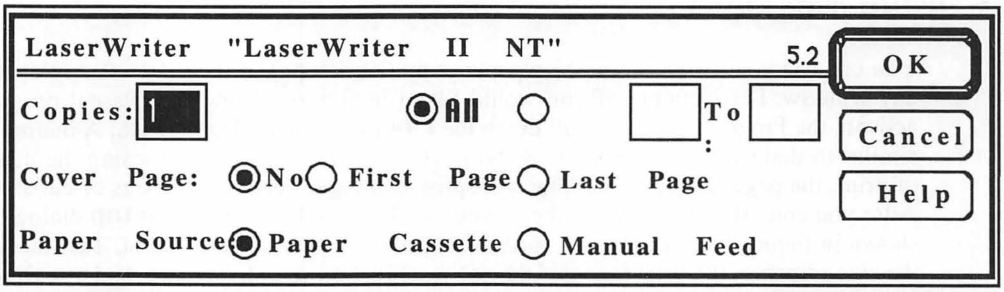
```
hPE^^.lineHeight := (HeaderHeight * 4) + 1;
```

4 spaces per tab

2 spaces per indent

Figure 1.17 The **Font Control** dialog box.

Remember that if you want to change the font of the text being displayed, you must choose **Font Control...** to do so.



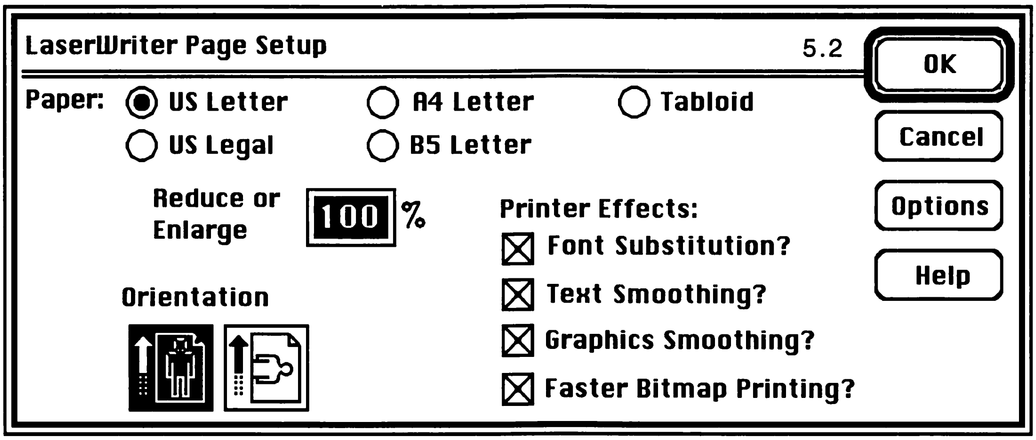
LaserWriter "LaserWriter II NT" 5.2

Copies: 1 Pages: [✓ All] From: To:

Cover Page: [✓ No] First Page Last Page

Paper Source: [✓ Paper Cassette] Manual Feed

Figure 1.18 The **Print** dialog box.



LaserWriter Page Setup 5.2

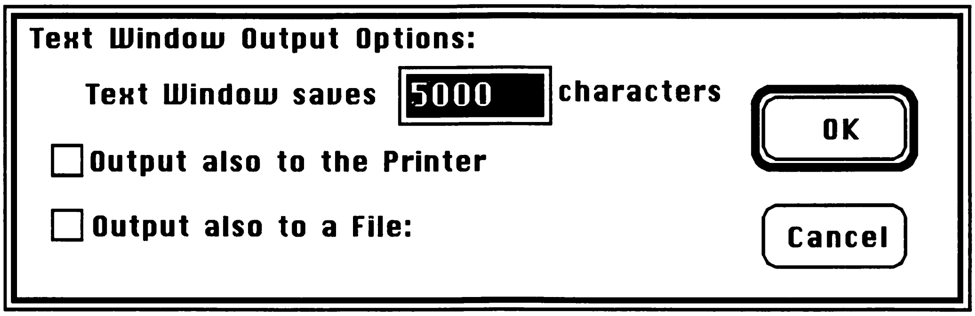
Paper: [✓ US Letter] A4 Letter Tabloid US Legal B5 Letter

Reduce or Enlarge 100 %

Orientation

Printer Effects:
[✓ Font Substitution?]
[✓ Text Smoothing?]
[✓ Graphics Smoothing?]
[✓ Faster Bitmap Printing?]

Figure 1.19 The **Page Setup** dialog box.



Text Window Output Options:

Text Window saves 5000 characters

Output also to the Printer

Output also to a File:

Figure 1.20 The **Preferences...** dialog box.

SUMMARY

In this chapter we have considered the general concept of a computer as an automated machine. We discussed the basic elements of the computer including input, output, memory, and the central processor.

We also introduced Macintosh Pascal, including all menus and windows. The Macintosh Pascal system supports five major menu options: **File**, **Edit**, **Search**, **Run**, and **Windows**. The menu option **File** supports the selections **New**, **Open**, **Close**, **Save**, **Save As...**, **Revert**, **Page Setup...**, **Print...**, and **Quit**. Most of these commands correspond to the **File** commands routinely found in other Macintosh applications. The second menu option, **Edit**, supports selection of **Cut**, **Copy**, **Paste**, **Clear**, and **Select All**. These options allow common cut-and-paste operations for editing lines of Macintosh Pascal code entered in the Program window. The menu option **Search** supports selections of **Find**, **Replace**, **Everywhere**, and **What to find....**, which are commands for searching for a string pattern and replacing one or more characters with a given pattern. Options are available for replacing only the first occurrence or for replacing all occurrences. The menu option **Run** provides several alternatives for executing a Macintosh Pascal program. This includes an option for checking the syntax of a program without execution by clicking on **Check**, the option to **Reset** a program if it has been halted and is to be executed from the beginning, and execution options such as **Go**, **Go-Go**, **Step**, and **Step-Step**. Several options for windows exist through the menu option **Windows** . These include selection of the program initially given the title Untitled, an **Instant** window for executing Macintosh Pascal commands without writing a Pascal program, the **Observe** window for tracing values during execution, the **Text** window for displaying text, the **Drawing** window for drawing graphics, the **Clipboard** for viewing lines of code cut from the Program Window, **Font Control...** for controlling the type and style of font displayed either in the Text window or the Program window, and **Preferences...** for directing output from a program to an external file or to a printer.

REVIEW QUESTIONS

1. Write a definition of the term *computer.*
2. Where have you recently seen computers being used?
3. Define the term *computer program.*
4. Why is a computer language important in solving problems?
5. What are the five basic units of a computer, and what purpose does each serve?
6. What is meant by the term *RAM?*
7. What is meant by the term *ROM?*
8. How can main memory be viewed?
9. What does a byte represent?
10. What is meant by the term *secondary memory?*
11. Define the term *file.*
12. What are the advantages of secondary memory?
13. What characters can be used in titling a file?
14. What is the maximum number of characters for a file?
15. How do you rename a file created when using Macintosh Pascal?
16. What three windows are initially shown in Macintosh Pascal?
17. The first line of any Pascal program begins with what command?
18. What is the rule for naming an identifier?
19. What are the six menu options for Macintosh Pascal?
20. What options exist when selecting the **Run** menu?
21. What options exist when selecting the **Edit** menu?
22. What options exist when selecting the **Window** menu?
23. What is the purpose of the **Instant** window?
24. What is the purpose of the **Check** option?
25. What is meant by a stop?
26. How can stops be inserted and removed in a source program?
27. What is the purpose of the **Observe** window?
28. What are the first steps in writing a program?
29. What is the difference between command options **Save** and **Save As...** ?
30. How can you edit your program?
31. What is the purpose of the Clipboard?
32. What is the purpose of the Text window? the Drawing window?
33. What is the purpose of the **Reset** command option?
34. How can a program displayed in the Program window be printed?
35. How can output from a Macintosh Pascal program be directed to a printer?
36. Use the **Instant** window to see the actions for the following commands. Be sure you click the mouse button on **Do It:**

```
writeln(45);
writeln(' { type your own name } ');
PaintCircle( 50, 45, 45 );
```

PROGRAMMING EXERCISES

In the following problems you are given Macintosh Pascal programs that are listed side by side. Please do not concern yourselves with the Pascal code, since much of what you see will be discussed in the chapters that follow. The purpose of these programs is to allow you to experience some of the types of syntax errors that can arise as you type Pascal code in the Program window. Enter the program on the left. Correct each error as it arises by looking at the corrected code to the right. Introducing your own errors beyond those given by a program on the left is highly encouraged.

1. Begin with a new Program window, and enter the program on the left by typing each line as it appears. If a syntax error appears, correct the error using the corresponding line from the program on the right. Save the program after it has been corrected, and then test the program by executing it. Select the **Go** command to execute it.

```
program Problem One(input, output)
{ This program prints your name.}
    var
        Name : string(30);
begin Body of main program }
    ShowText
{ Prompt for name. }
    write('Enter your name:  );
    readln( Name ;
{ Display your name.
    writeln('Your name is Name );
end
```

```
program Problem_One(input, output);
{ This program prints your name.}
    var
        Name : string[30];
begin { Body of main program.}
    ShowText;
{ Prompt for name. }
    write('Enter your name: ' );
    readln(Name);
{ Display your name. }
    writeln('Your name is ',Name);
end.
```

2. Repeat the procedure used in Exercise 1 for this Pascal program.

```
program Prob_Two(input, output)
{ This program displays 10 numbers. }
    uses
        QuickDraw1;
    var
        Counter  ;  integer :

begin
{ Hide all windows but the }
{ Text Window. }
    HideAll;
    ShowText
{ Display 10 numbers. }
```

```
program Prob_Two(input, output);
{ This program displays 10 numbers. }
    uses
        QuickDraw1;
    var
        Counter : integer;

begin
{ Hide all windows but the }
{ Text Window. }
    HideAll;
    ShowText;
{ Display 10 numbers. }
```

```
    for Counter <-- 1 to 10
        writeln( Counter );
end.
```

```
    for Counter := 1 to 10 do
        writeln( Counter );
end.
```

3. Repeat the steps used in Exercise 1 for the Pascal program that follows.

```
program Problem_Three(input, output);
{ This program keeps displaying }
{ numbers. Use menu option Pause }
{ to halt execution. }
    var
        Number ; integer:
begin
    ShowText
    while true
        begin
            Number = random;
            writeln( Number ' ' )
        end;
end.
```

```
program Problem_Three(input, output);
{ This program keeps displaying }
{ numbers. Use menu option Pause }
{ to halt execution. }
    var
        Number : integer;
begin
    ShowText;
    while true do
        begin
            Number := random;
            writeln( Number, ' ' )
        end;
end.
```

4. If a printer is attached to your Macintosh, what steps are needed to select the proper menu option and dialog windows for activating printing as numbers are displayed to the Text window. Try this with Exercise 3.

5. Repeat the steps used in Exercise 1 for this Pascal program.

```
progam Problem_Five(input, output);
{ This program generates 10 odd }
{ numbers and stores them in a }
{ special table called an array. }
    foruse
        QuickDraw1;
    var
        Data : array[1.10] of;
        Index, Count : integer;
begin
{ Open Text window for viewing. }
    HideAll;
    ShowText;
{ Generate 10 odd numbers. }
    Index := 1;
    for Count = 1 to 19 try
        begun
            if odd( Count ) then
                begun
                    Data[Index] <-- Count
                    Index : = Index + 1
                end;
```

```
program Problem_Five(input, output);
{ This program generates 10 odd }
{ numbers and stores them in a }
{ special table called an array. }
    uses
        QuickDraw1;
    var
        Data : array[1..10] of integer;
        Index, Count : integer;
begin
{ Open Text window for viewing. }
    HideAll;
    ShowText;
{ Generate 10 odd numbers }
    Index := 1;
    for Count := 1 to 19 do
        begin
            if odd(Count) then
                begin
                    Data[Index] := Count;
                    Index := Index + 1
                end;
        end;
{ Display the 10 odd numbers. }
    for Index : 1 to 10 do
        writeln( Data[Index] );
end.
```

```
        end;
{ Display the 10 odd numbers. }
    for Index := 1 to 10 do
        writeln( Data[Index] );
end.
```

Exercises 6, 7, and 8 contain syntax and semantic errors that are only detected when the program is checked or when the program is in execution. Select **Check Syntax** from menu option **Run** to check for any syntax errors before selecting **Go** to execute. In each exercise the program on the left contains errors, and the program on the right is correct. Our suggestion is to correct an error, then repeat selection of the option **Check Syntax** to determine the next syntax error.

6. Check and run the following Pascal program, correcting all syntax and semantic errors.

```
program Problem_Six(input, output);
{ This program detects division }
{ by zero. }
    var
        One, Two, Three, Four : real;
begin
    ShowText;
    One := 10;
    Two := 20;
    Three := 0;
    Four := Two / Three + One;
    writing ( 'Value of Four: ', Four )
end;
```

```
program Problem_Six(input, output);
{ This program detects division }
{ by zero. }
    var
        One, Two, Three, Four : real;
begin
    ShowText;
    One := 10;
    Two := 20;
    Three := 0;
    Four := Two / ( Three + One );
    writeln ( 'Value of Four: ', Four )
end.
```

7. Check and run the following Pascal program, correcting all syntax and semantic errors.

```
program Problem_Seven(input, outp¨t);
{ This program generates 10 odd }
{ numbers and stores them in a }
{ special table called an array. }
    uses
        QuickDraw;
    var
        Data : array[1..10]  of integer;
        Index, Count : integer;
begin
{ Open Text window for viewing. }
    Hide_All;
    Show_Text;
{ Generate 10 odd numbers. }
    Number := 1;
```

```
program Problem_Seven(input, output);
{ This program generates 10 odd }
{ numbers and stores them in a }
{ special table called an array. }
    uses
        QuickDraw1;
    var
        Data : array[1..10]  of integer;
        Index, Count : integer;
begin
{ Open Text window for viewing. }
    HideAll;
    ShowText;
{ Generate 10 odd numbers }
    Index := 1;
    for Counter := 1 to 19 do
        begin
            if odd( Count ) then
                begin
                    Data(Index) := Count
                    Index := Index + 1
                end;
        end;
{ Display the 10 odd numbers. }
    for Index := 1 to 20 do
        writeln( Data( Index ) );
end.
```

```
    for Count := 1 to 19 do
        begin
            if odd(Count) then
                begin
                    Data[Index] := Count;
                    Index := Index + 1
                end;
        end;
{ Display the 10 odd numbers. }
    for Index := 1 to 10 do
        writeln( Data[Index] );
end.
```

8. Check and run the following Pascal program, correcting all syntax and semantic errors.

```
program Problem_Eight(input, output);
{ This program displays 10 }
{ random numbers. }
    var
        A : array[1..100] of real;
        Index : integer;
begin
{ Generate 10 random numbers. }
    for Index := 1 to 10 do;
        A[Index]  := random;
{ Display 10 random numbers. }
    for Index := 10 downto 1 do
        writeit( A[Index] :10:3 ,' ');
    writeln();
end.
```

```
program Problem_Eight(input, output);
{ This program displays 10 }
{ random numbers. }
var
    A : array[1..100] of real;
    Index : integer;
begin
{ Generate 10 random numbers. }
    for Index := 1 to 10 do
        A[Index]  := random;
{ Display 10 random numbers. }
    for Index := 10 downto 1 do
        write( A[Index] :10:3 ,' ');
    writeln;
end.
```

9. Enter the code for the program Electric_Bill (listed on page 31 in Chapter 2). Check the program and, if no syntax error is reported, run the program. Use reasonable data, and enter it as it is requested. Save the program on your disk for later use.

10. When executed, the following Macintosh Pascal program will generate a random pattern in the form of ovals, as shown in Figure 1.21. Enter this program, check for errors, and then choose an option to run.

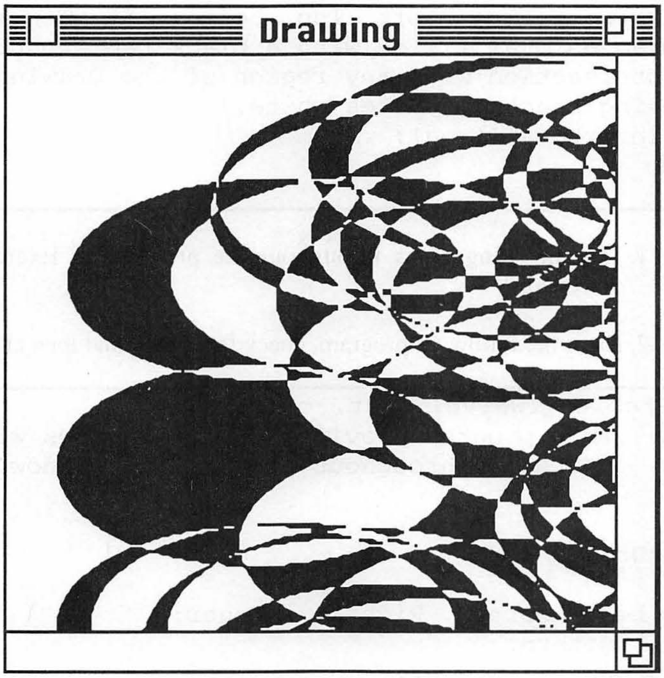


Figure 1.21 Patterns generated by execution of the program Random_Patterns.

```
program Random_Patterns(Input. Output);
{ Purpose:  This program draws ovals randomly in the Drawing }
{           window. }
   uses
      QuickDraw1;
   const
      Limit = 220;
   var
      Area : Rect;
      Counter : integer;
     Left, Top : integer;
begin
{ Open the Drawing window. }
   ShowDrawing;
{ Draw patterns in Drawing window. }
   for Counter := 1 to Limit do
      begin
      { Establish coordinates for the upper left corner of the }
      { rectangle called Area. }
         Left := - random mod 512 + 512;
      Top := - random mod 342 + 342;
      { Establish the rectangle for drawing an oval. }
```

```
          SetRect(Area, Left, Top, Left + 150, Top + 75);
        { Draw an oval filled with a black background. }
        { Intersection with any region of the Drawing window, }
        { being black, produces white.}
          InvertOval(Area);
        end;
end.
```

11. Try inserting stops throughout the program in Exercise 10 to observe its execution?

12. Enter the following program, check for errors, and then choose the option to run.

```
program Problem_Twelve(input, output);
{ Purpose:  This program provides random squares with different }
{           patterns throughout the Drawing window. }

  uses
     QuickDraw1;
  var
     Top, Left, Bottom, Right : integer;
     Pat : Pattern;
     Background : integer;

begin
{ Use function Random to choose the corners of the square }
{ and a background pattern. }
   while true do
      begin
      { Randomly select the rectangles for drawing an oval. }
         Top := abs(random) mod 201;
         Left := abs(random) mod 201;
         Bottom := Top + 30;
         Right := Left + 30;
      { Randomly select the background color. }
      Background := abs(random) mod 5;
      case background of
         0 :
               Pat := white;
            1 :
               Pat := black;
            2 :
               Pat := gray;
            3 :
               Pat := ltgray;
            4 :
               Pat := dkgray;
      end;
   { Display the oval in a rectangle with a chosen background }
   { pattern. }
   FillOval(Top, Left, Bottom, Right, Pat);
   end;
end.
```

13. Enter the following program, check for errors, and then choose the option to run.

```
program Problem_Thirteen(input, output);
{ Purpose:  This program draws a series of nested squares. }
   uses
      QuickDraw1;

begin
{Set PenSize for 15 wide and 15 high. }
   PenSize(15, 15);
   MoveTo(5, 20);
   WriteDraw('(25, 25)');
   MoveTo(145, 20);
   WriteDraw('(175, 25)');
{ Draw the first square. }
   DrawLine(25, 25, 160, 25);
   LineTo(160, 160);
   LineTo(25, 160);
   LineTo(25, 25);
{ Draw the second square. }
   DrawLine(55, 55, 130, 55);
   LineTo(130, 130);
   LineTo(55, 130);
   LineTo(55, 55);
 { Draw third square. }
   DrawLine(85, 85, 100, 85);
   DrawLine(100, 100, 85, 100)
end.
```

14. Enter the following program, check for errors, and then run. Be sure that you have saved a copy of the program before choosing the option to run. If you encounter errors, correct each error and save before again choosing the option to run.

```
program Problem_Fourteen(input, output);
{ Purpose:  This example shows how a region of arbitrary shape }
{           can be generated using several region procedures. }

   uses
      QuickDraw1, QuickDraw2;
   type
      Port = GrafPtr;
  var
      Window : Port;
      Rectangle : array[1..2] of Rect;
      J : integer;
      Actual_Rgn : array[1..3] of RgnHandle;

{ ***************************************************** }
   procedure Open_Window (var Viewport : Port);
   begin
      new(Viewport);
      OpenPort(Viewport)
   end; { Open_Window }

{ ***************************************************** }
   procedure Dispose_of_Window (var Viewport : Port);
   begin
      ClosePort(Viewport);
   dispose(Viewport)
   end; { Dispose_of_Window }

{ ***************************************************** }
   procedure Grow_Region (var Rgn : RgnHandle;
                       var Box : Rect;
                             Left, Top, Right, Bottom : integer);
   begin
      OpenRgn;
      SetRect(Box, Left, Top, Right, Bottom);
      FrameOval(Box);
      CloseRgn(Rgn);
  end; { Grow_Region }

{ ***************************************************** }
begin  { Body of the main program. }
{ Hide all Macintosh Pascal windows. }
   HideAll;
{ Establish two initial windows. }
   Open_Window(Window);
   for J := 1 to 3 do
      Actual_Rgn[J] := NewRgn;
{ Generate the first region. }
   Grow_Region(Actual_Rgn[1], Rectangle[1], 56, 73, 356, 273);
{ Generate the second region. }
   Grow_Region(Actual_Rgn[2], Rectangle[2], 206, 103, 306, 243);
{ Generate third region. }
   DiffRgn(Actual_Rgn[1], Actual_Rgn[2], Actual_Rgn[3]);
{ Show the third region. }
   SetPort(Window);
   Window^.clipRgn := Actual_Rgn[3];
   FillRgn(Actual_Rgn[3], white);
{ Paint border on the edges of third region. }
   PenSize(2, 2);
   FrameRgn(Actual_Rgn[3]);
{ Draw in the third region. }
```

```
   MoveTo(100, 160);
   TextFace([italic, underline]);
   DrawString('Third Region');
{ Close windows and dispose of regions. }
   Dispose_of_Window(Window);
   for J := 1 to 3 do
      DisposeRgn(Actual_Rgn[J]);
end.
```

Chapter 2

Introduction to THINK Pascal

OBJECTIVES

After completing Chapter 2, you will know the following:

1. The general nature of the windows and menus used in THINK Pascal.
2. The concept of a project and how to create one.
3. The concept of a source program and how to create and edit a THINK Pascal source program.
4. How to add a library to a project.
5. How to Check, Execute, Save, and Print THINK Pascal Programs.

2.1 INTRODUCTION TO THINK PASCAL AND THINK PASCAL WINDOWS

THINK Pascal is a development environment for generating Macintosh applications. It is different from Macintosh Pascal in that (1) it supports a fast compiler, while Macintosh Pascal is an interpreter; (2) it has a more advanced integrated text editor for Pascal syntax than that found in the Program window of Macintosh Pascal; (3) it has an automated Make utility for rebuilding various files linked with the development of an application; (4) it has advanced debugging tools not found in Macintosh Pascal; (5) it includes a class library and class browser for developing object-oriented programs; and (6) it includes a project manager that binds various files into a single project. Programs written in Macintosh Pascal can be compiled and executed in THINK Pascal. The reverse is not true, since THINK Pascal supports extensions that are not supported in Macintosh Pascal.

The present THINK Pascal system (version 4.0) comes as a four-disk set and contains instructions for unpacking many of the files. Following the instructions given in the manual, the complete development environment can be transferred onto a hard disk drive. *Although it may be possible to use THINK Pascal without a hard disk, you will find that to do so is limiting and difficult. We strongly recommend that you not attempt to use the software without a hard disk.* By carefully following the instructions, you create a development folder similar to the one shown in Figure 2.1, with the Pascal application program in the folder called THINK Pascal 4.0 Folder.

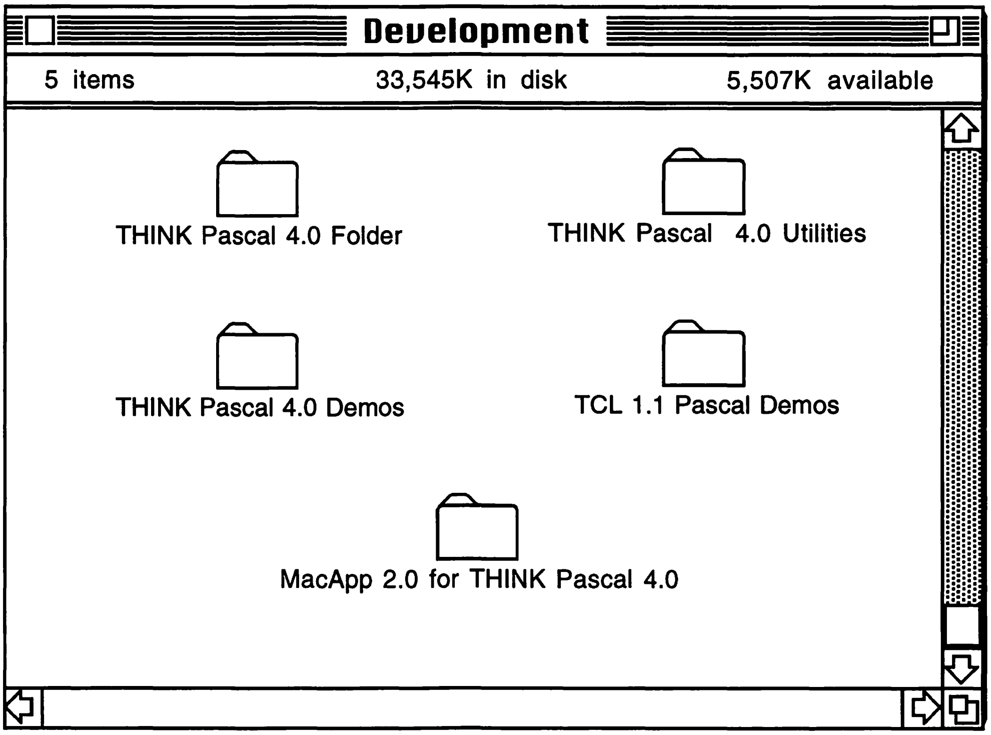


Figure 2.1 The THINK Pascal Development folder.

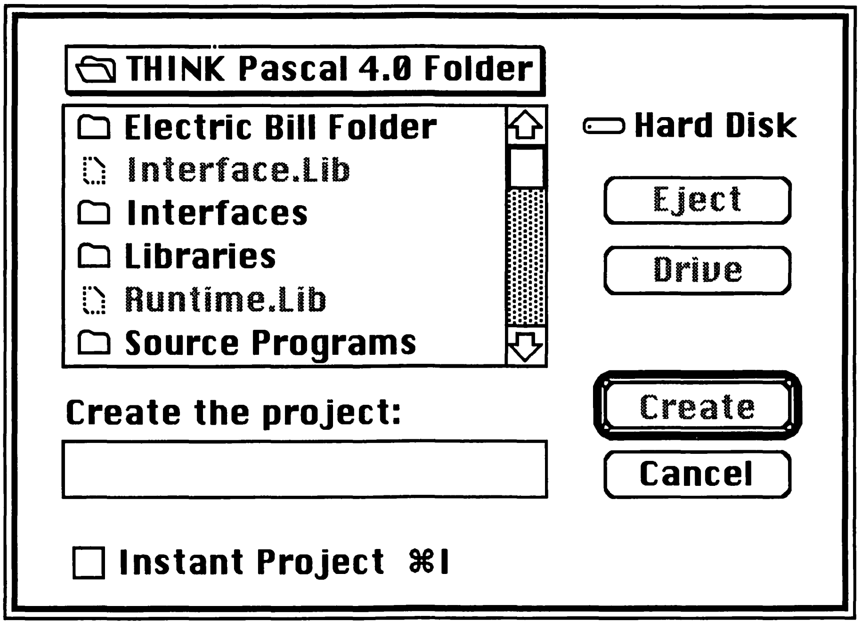
Creating a THINK Pascal program is somewhat more complicated than creating a Macintosh Pascal program. It begins with a step which may at first seem awkward and unnecessary, the construction of a project. To illustrate the process, we will use an example of a Pascal program entitled Electric Bill. The purpose of this program is to allow you to take your electric bills for the past 12 months and, after entering the consumption and cost figures for each month, get a monthly average for each of these items. The listing for this program is as follows. (We will discuss the meaning of the various lines in the program in the next chapter.)

```
program Electric_Bill (input, output);
{ Purpose:  This program computes the total consumption and }
{           cost of electricity used over a 12-month period. }
  var
    Counter, Total_Consumption, Total_Cost : integer;
    Consumption, Cost : integer;
    Average_Consumption, Average_Cost : real;
```

```
begin
   ShowText;
{ Initialize Counter and totals. }
   Counter := 1;
   Total_Consumption := 0;
   Total_Cost := 0;
{ Repeat entry of consumption and cost data until }
{ counter exceeds 12. }
   repeat
   { Enter data from the keyboard. }
      writeln('Enter consumption');
      readln(Consumption);
      writeln('Enter cost');
      readln(Cost);
   { Compute the partial sums then modify the value of Counter. }
      Total_Consumption := Total_Consumption + Consumption;
      Total_Cost := Total_Cost + Cost;
      Counter := Counter + 1;
   until (Counter > 12);
{ Compute the average values of consumption and cost.}
   Average_Consumption := Total_Consumption / 12;
   Average_Cost := Total_Cost / 12;
{ Display the results. }
   writeln('Average monthly consumption: ',
   Average_Consumption : 7 : 2);
   writeln('Average monthly cost:', Average_Cost: 6: 2)
end.
```

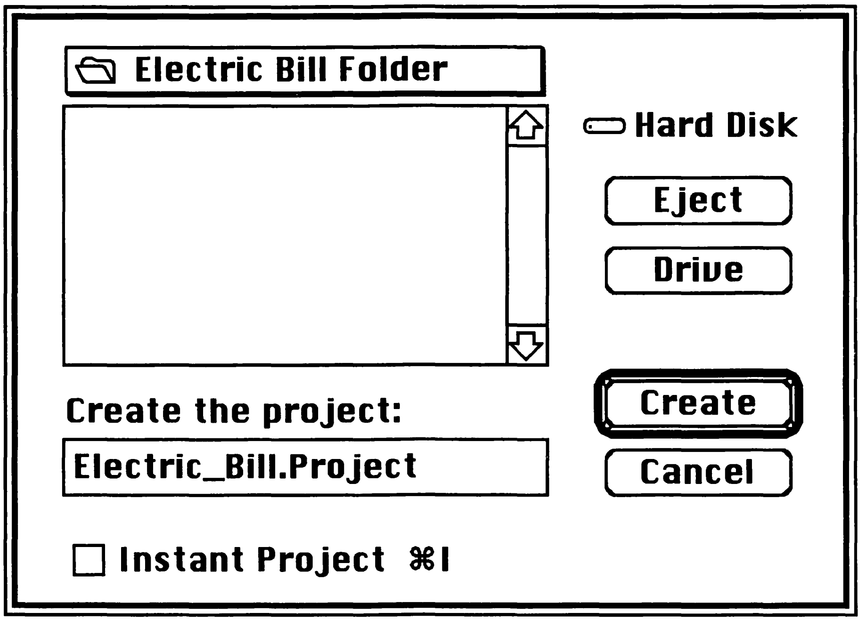
The first step in setting up a project actually occurs prior to launching the THINK Pascal software. This step is to create an empty folder for the project and supply (make accessible) any needed library files. For our example, we will create a folder entitled Electric Bill Folder. We assume you have just turned on your Mac and the Finder window is on the screen. To create this folder go to the Finder **File** menu and select **New Folder**. Then change the name of the empty folder to Electric Bill Folder. Next move this folder to the THINK Pascal Development folder, and open the THINK Pascal 4.0 Folder to locate the two library files you will need. Open the folder entitled Libraries, and drag the files `Runtime.Lib` and `Interface.Lib` into the THINK Pascal 4.0 Folder. These two files are needed by almost all projects, and locating them in the same folder as the project folder will make them accessible to the project.

The second step in creating a project is to launch the THINK Pascal application by double clicking the THINK Pascal icon. If you loaded the THINK Pascal software correctly, this application icon should be located in the THINK Pascal 4.0 Folder. Opening the application results in a dialog box similar to the one shown in Figure 2.2. Double click on the Electric Bill Folder (to keep your files neatly organized), and then click **New**. A dialog box like the one shown in Figure 2.3 will appear. *Do not be concerned if the dialog boxes you see are not exactly the same as those shown in the figures.* The differences result from the different files in the folders when the dialog boxes appear. Enter the name of the project, `Electric_Bill.Project`, as shown in Figure 2.3. THINK Pascal has adopted the convention of using the symbol π as a suffix to indicate a project file. If you wish to follow this convention, you can produce the symbol π by typing a character p while the option key is depressed.



Create the project:

Figure 2.2 THINK Pascal opening dialog box.

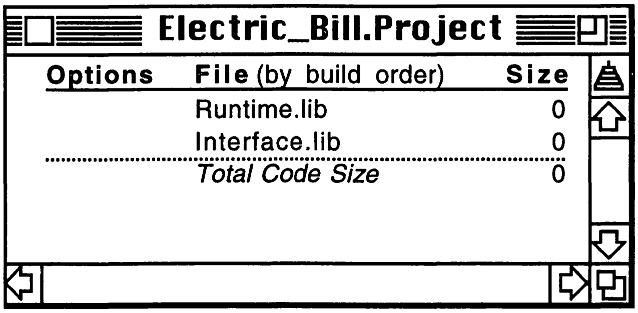


Create the project: Electric_Bill.Project

Figure 2.3 Naming and creating the project.

To simplify and perhaps clarify matters, we have chosen to use the suffix *project* to designate project files. Whichever approach you elect to take, designate the project name, and then click **Create**. With this action you will create the project file.

The creation of a project results in the opening of a Project window, as illustrated in Figure 2.4. Notice that initially the project contains only the two library files you placed in the THINK Pascal 4.0 Folder. Specifically, they are the file `Runtime.lib` and the file `Interface.lib`. These files contain standard Pascal and Macintosh Toolbox routines that almost all Pascal programs require in order to execute.


Electric_Bill.Project

Options	File (by build order)	Size
	Runtime.lib	0
	Interface.lib	0
	Total Code Size	0

Figure 2.4 The Project window.

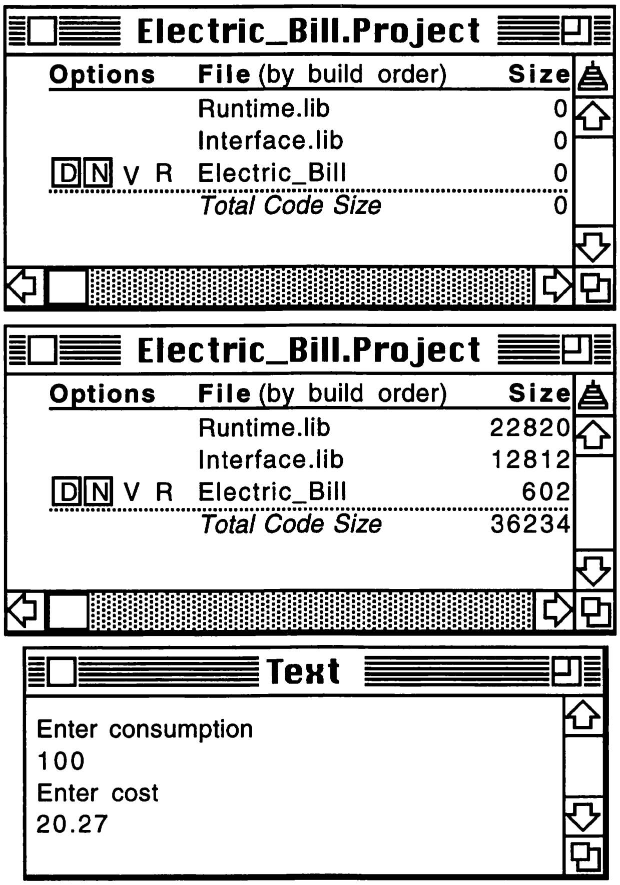
You are now ready for the third step in the creation of the project, the creation of the source file. To accomplish this, select the command option **New** from the **File** menu. This action results in the opening of an empty Edit window called **Untitled-1**. The Edit window in THINK Pascal is the counterpart to the Program window in Macintosh Pascal. Do not let this slight variation in terms confuse you. Type the program `Electric_Bill` (exactly as shown on the previous page) in this window. Be aware that punctuation can be critical. A misplaced semicolon can cause interesting results! The Edit window has been designed to help you enter correct Pascal source code. If you enter something incorrectly, you may see it displayed in outline font. If this occurs, you will know there is an error. Test this by deliberately entering incorrect code and observing the reaction of the Edit window.

When you have correctly typed the program, select **Save As...** from the **File** menu, and name the source file `Electric_Bill`. To follow the THINK Pascal convention, you would name it `Electric_Bill.p`. This convention uses the character p to denote a Pascal source file and a π to denote the project file.[1]

As a fourth step in building the project, you must add the source file to the project. To do this, select the **Add "Electric_Bill"** command option from the **Project** menu. If there is not an active Edit window, this option will appear on the menu as **Add Window**

[1] We have adopted the following convention for naming source-program files. If a program will execute only as a Macintosh Pascal file, we will give it the suffix `MAC`. If it will execute only as a THINK Pascal file, we will give it the suffix `THINK`. If it will execute in both translators, we will use no suffix. Thus, a program named `Circle.MAC` would be a Macintosh Pascal program, and a program named `Circle.THINK` would be a THINK Pascal program. We hope this will be helpful in keeping the various kinds of programs separate. Be careful to avoid confusing the Pascal program name, which appears in the first line of the program, and the Macintosh file name. The former is an identifier and subject to the rules of the identifier. The latter is a Macintosh file name and therefore much more flexible. This difference accounts for the presence of the period in a file name such as `Circle.Mac`.

and will be inactive (dimmed). If there is an Edit window but the source file has not yet been saved, this command option will appear as **Add "Untitled-1"**. Selection of **Add "Untitled-1"** will result in an intermediate step with a dialog box instructing you to save your source file. When this step has been successfully completed, the name of the source file should be shown in the Project window, as illustrated in the top portion of Figure 2.5.



```
Electric_Bill.Project
Options      File (by build order)      Size
             Runtime.lib                   0
             Interface.lib                 0
D N V R      Electric_Bill                 0
             Total Code Size               0
```

```
Electric_Bill.Project
Options      File (by build order)      Size
             Runtime.lib               22820
             Interface.lib             12812
D N V R      Electric_Bill               602
             Total Code Size           36234
```

```
Text
Enter consumption
100
Enter cost
20.27
```

Figure 2.5 The Project window after addition of the source file and then after compiling; the Text window after execution of the program.

The fifth and final step in building the project is to select the **Go** command option under the **Run** menu. This causes the source file to be compiled. A dialog box indicating that `Electric_Bill` is being compiled will appear for a few seconds. If all is well with Steps 1 through 5, the program will now execute. If this is the case, the Text window will appear, and information on your first month's electric bill will be requested. This is also shown in Figure 2.5. Notice that after the compiling step, the size of each file in the project is given, as shown in the middle part of Figure 2.5. To complete the execution of `Electric_Bill`, enter data for 12 months as prompted, and the program should provide you with summary statistics based on these entries. If for some reason you wish to interrupt a program before execution is complete, the following applies. When a program is in execution, a bug spray-can icon appears to the far right of the menu bar. This icon appears only when a program is in execution and disappears once the program has completed execution. Clicking on this icon or typing Command-Shift-Period interrupts program execution. When a program has been interrupted, an Edit window is opened with a finger pointing to the statement where the program stopped. Execution can be continued by using any one of the command options—**Go**, **Step Into**, **Step Over**, **Go-Go**, or **Step-Step**—from the **Run** menu.

The five steps in the building of a project can be summarized as follows:

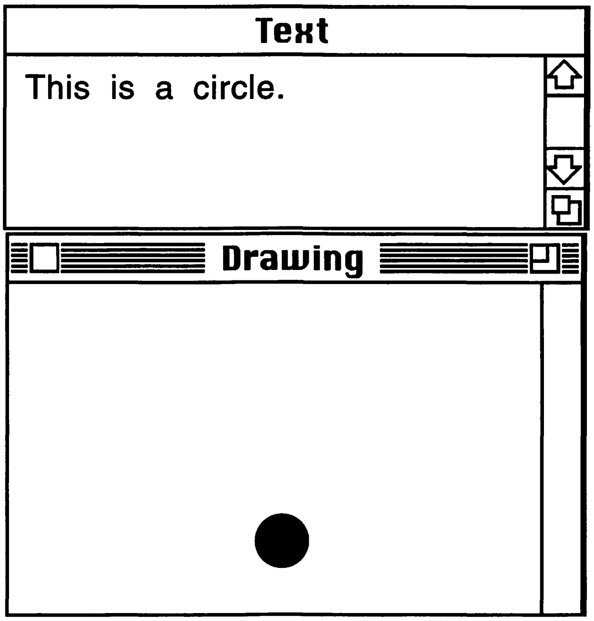
1. Create the project folder, `MyProgram Folder`, and be sure the files `Runtime.Lib` and `Interface.Lib` are located in the THINK Pascal 4.0 Folder.
2. Create the project file, `MyProgram.Project`. Be sure this file is in the project folder.
3. Enter the source code for the program in the Edit window, and save as it as your source file, `MyProgram.THINK`.
4. Add this source file to the project. (Select the command option **Add "MyProgram.THINK"** under the **Project** menu.)
5. Compile and execute by selecting **Go** under the **Run** menu.

Perhaps the best way to learn to use the Pascal software is to type a short program in the Edit window and then go through each of these five steps. If you carefully complete each step, you should be able to successfully execute the program at the end. If you have not already done so, enter the program `Electric_Bill`. Just for good measure, you might also try the shorter and simpler program, `Circle`. Go through Steps 1 and 2 as outlined above, creating a project called `Circle.Project`. Then enter the following source program in the Edit window. (You should recognize this program from Chapter 1.)

```
program Circle(input,output);
begin
   ShowText;
   writeln('This is a circle.');
   ShowDrawing;
   PaintCircle(100,100,10);
end.
```

Save this source program as `Circle.THINK`, and add the file to the project. Select **Go** to compile and execute the program. The result is shown in Figure 2.6. Notice that the output of the program appears in two different windows and that the program contains commands to ensure that the necessary windows are displayed. This is an important aspect of creating a THINK Pascal program and one of the areas where THINK Pascal and Macintosh Pascal differ somewhat.

In order to keep this chapter brief, not all of the dialog windows we discuss are shown. Some will appear in later chapters, and others should be explored by the reader. We encourage you to apply each of the command options and to view and select options from the various dialog windows in order to gain a better understanding of the command options.



```
This is a circle.
```

Figure 2.6 Output from the program `Circle`.

2.2 THE THINK PASCAL MENUS

In this section we briefly describe the THINK Pascal menus and the commands found under each one. Assuming that you are building on your experience from Chapter 1, our introduction to the THINK Pascal menus will be brief. You are encouraged to further review each menu by pulling it down and trying the various command options. Pay particular attention to the choices provided by the dialog windows that result from selection of some of the commands. Some of the menus have more than one set of command options. To see these variations, press the Shift key and the Option key (separately) while each menu is displayed. By pressing and releasing these keys several times, you can easily see the different option sets of each menu. The **Windows**, **Edit**, and **Search** menus have only one set of options. You are also encouraged to carefully read the *THINK Pascal User Manual.*

2.2.1 The **File** Menus

Figure 2.7 shows the THINK Pascal **File** pull-down menu and its alternate (Option key depressed) version. Most of the options under this menu are standard for Macintosh **File** menus and need only be reviewed briefly. The **New** command results in a blank Edit window. This is the command option you choose when a new source program is to be entered. **Open...**[2] allows you to open a previously saved THINK Pascal source file in the

[2] The three dots following a menu command indicate that the selection of this option will produce a dialog box. This is standard Macintosh symbolism.

Edit window. THINK Pascal allows several Edit windows to exist within one project. Open... also allows you to open additional Edit windows as needed. Close allows you to close the active window, and Close All allows you to close all open files. Save allows you to save the contents of the Edit window to your disk, while Save All allows you to save all open files. Save a Copy As... allows you to make backup copies without erasing the file currently on disk. Revert allows you to replace the present contents of the Edit window with the latest version of the current file that has been saved to disk.

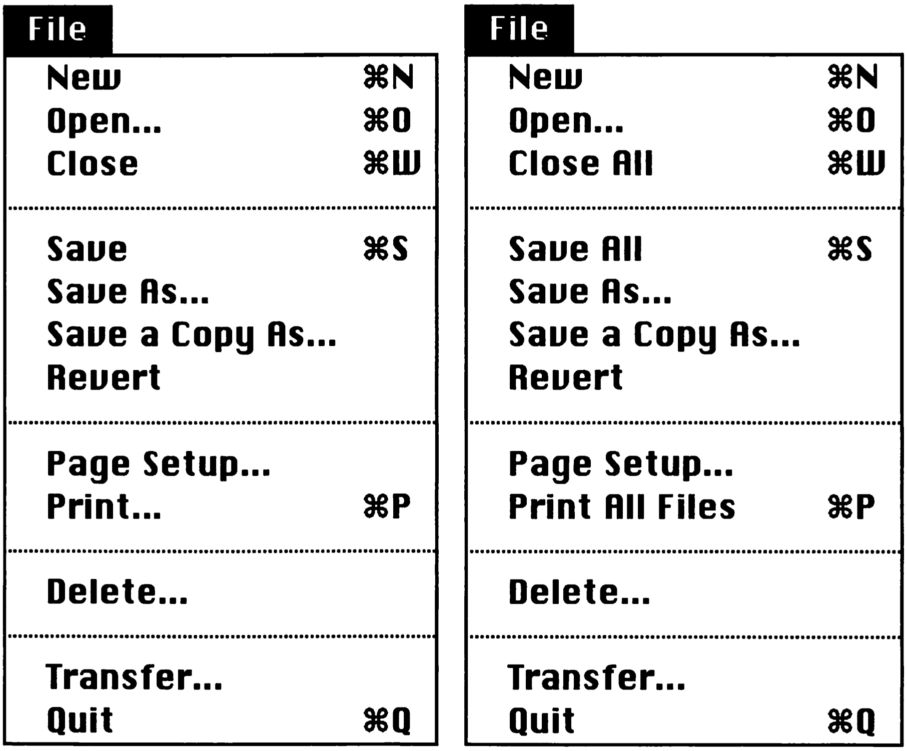


Figure 2.7 The THINK Pascal File menus. The menu on the right appears when the option key is depressed.

Page Setup... results in the standard Macintosh Page Setup dialog box with its various options (see Figure 1.19). Print... produces a dialog box (see Figure 1.18) and results in the printing of the contents of the active window. Print All Files results in the printing of all source files that have been added to the Project window. Delete... allows you to remove a file from a folder without returning to the Finder. Transfer... allows you to move to another application without first returning to the Finder. Under MultiFinder, the THINK Pascal application will remain open if you transfer to another application. Therefore you may return from the other application to THINK Pascal via MultiFinder. Finally, Quit closes the THINK Pascal application.

2.2.2 The **Edit** Menu

The various commands included under the **Edit** menu are shown in Figure 2.8. The top part of this menu includes the standard Macintosh commands **Undo**, **Cut**, **Copy**, **Clear**, **Select All**, and **Show Clipboard. Undo** allows you to undo the last edit operation. The way this command is displayed can change according to the last operation that can be undone. For example, in Figure 2.8 **Undo** is shown as **Undo Typing**, which tells you that the last recoverable operation was the typing of text. **Cut**, **Copy**, and **Paste** allow you to do further editing of your source file by cutting, copying, and pasting parts of the file, while **Clear** removes the selected portion of the source file without placing it on the clipboard. **Select All** selects the entire file for cutting or copying. **Show Clipboard** allows you to view the contents of the clipboard.

Source Options... produces the dialog box shown in Figure 2.9 with options for customizing the appearance of the source code. This option sets the characteristics for all Edit windows within a single project. Through this dialog box you are able to control four elements of the format of the display. First, you can select the font and font size in which your source file will be displayed. Second, you may determine the way in which keywords will be displayed (for example, lowercase bold print). Third, you may control the indentation and tab size for your programs. Finally, you may control whether parameter lists are displayed horizontally or vertically. Note that while **Source Options...** is helpful in controlling the way your source program is displayed, it is not critical to either your understanding of THINK Pascal or the operation of THINK Pascal. Experiment with these options until you settle on a format that pleases you.

The next command option under the **Edit** menu is **Auto-Reformat**. Selection of this option causes THINK Pascal to reformat your source file each time you press the *return* key or type a semicolon. You may turn this off if you would prefer to have your file reformatted less frequently. If you turn it off, you may still cause the file to be reformatted at any time by pressing the *enter* key on the numeric keypad.

The final option under the **Edit** menu is **Projector-Aware**. This command option applies when working on a large project using MPW projector. Refer to your *THINK Pascal User Manual* for further discussion.

2.2.3 The **Search** Menu

The next menu on the THINK Pascal menu bar is **Search**, which is shown in Figure 2.8 along with the **Edit** menu. The command options in the **Search** menu allow you to quickly find strings in your source files.[3] The **Find...** command results in a dialog box where several options are allowed in searching for a string. You may look for separate words or for strings embedded in other strings. You may specify whether or not case is relevant in the search.

[3] Strings will be discussed at some length in Chapter 11. For now, think of a string as a collection of characters. For example, the word government is a ten-character string. A string does not have to be a complete word, however. It can be part of a word (for example, gove is a string embedded in the larger. string government) or several words (*Good government is hard to find* is also a string) or even a number.

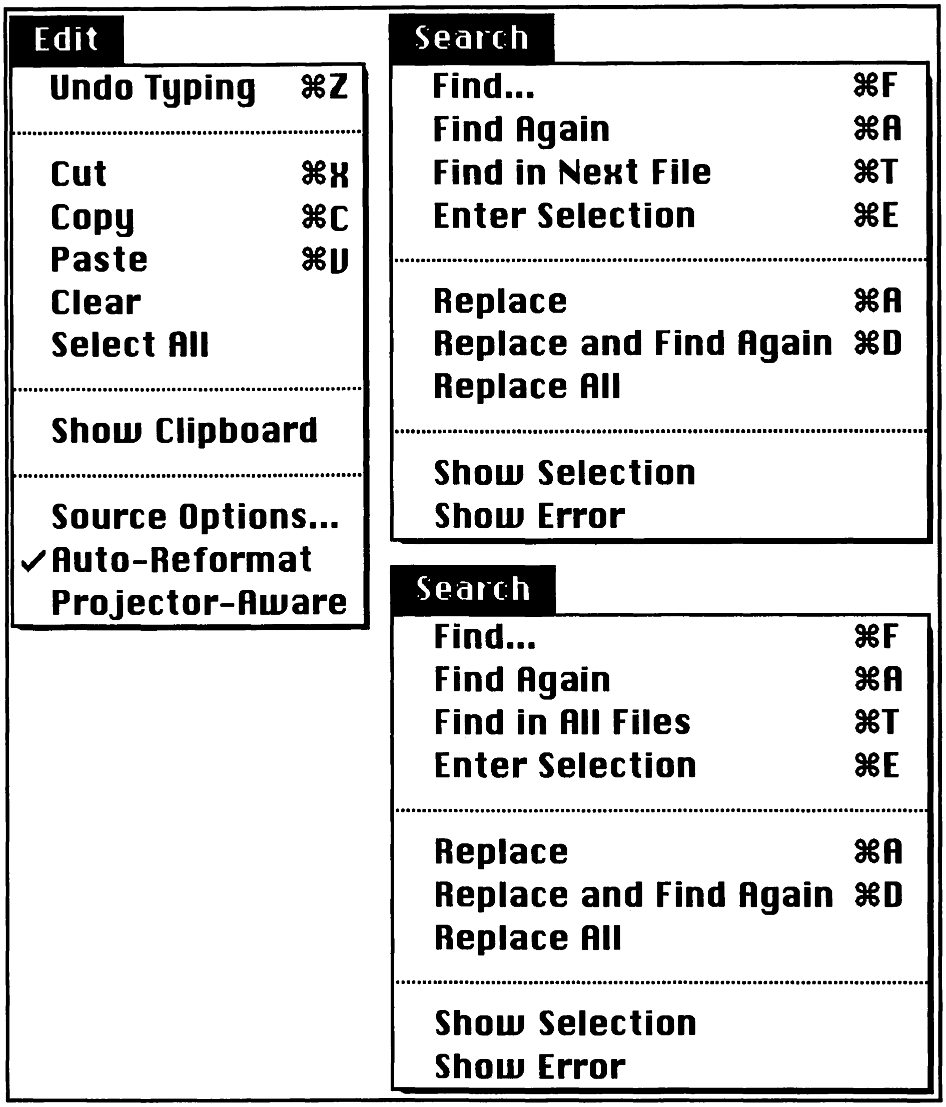
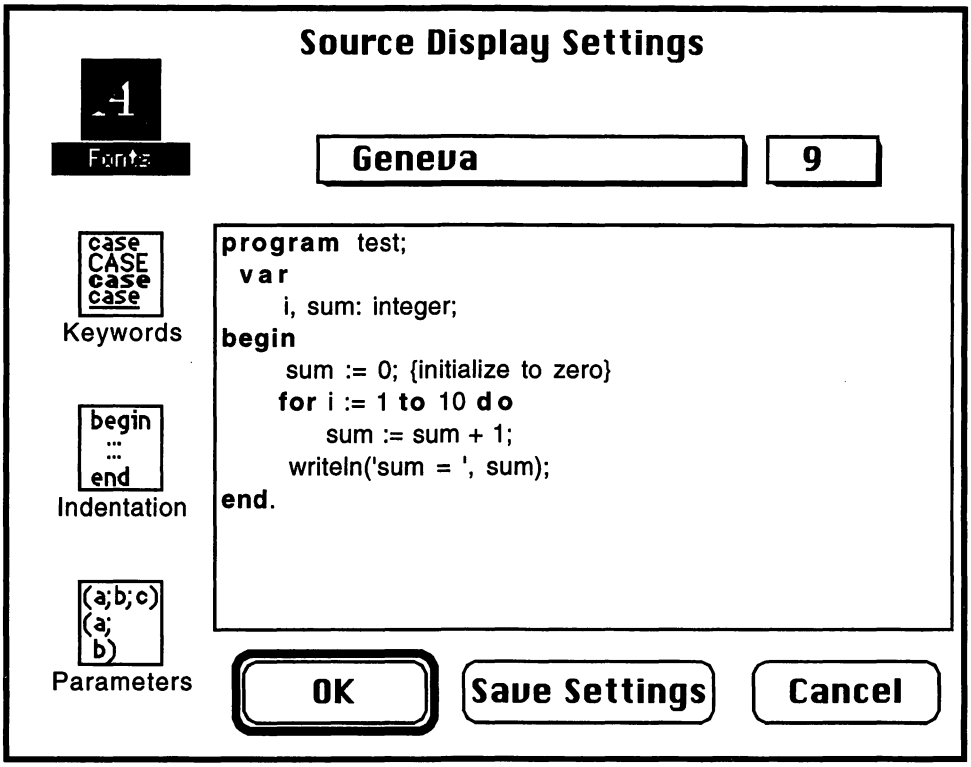


Figure 2.8 The **Edit** and **Search** menus of THINK Pascal. The lower **Search** menu appears when the Option key is depressed.

The **Find Again** command allows you to search for another occurrence of the same string. The **Find in Next File** command allows you to extend your search to more than one Edit window. By pressing the **Option** key when you pull down the **Search** menu, you can transform this option to **Find in All Files**. This option has the same effect as repeatedly selecting **Find in Next File**. The last option in this group of commands is **Enter Selection**. This option has the effect of selecting the highlighted string as the new search string. Thus, you can start a new search sequence with this command.



```
program test;
  var
      i, sum: integer;
begin
      sum := 0; {initialize to zero}
      for i := 1 to 10 do
          sum := sum + 1;
      writeln('sum = ', sum);
end.
```

Figure 2.9 The **Source Options...** dialog box.

The next group of commands, **Replace**, **Replace and Find Again**, and **Replace All**, allow you to determine how to replace a string that is found. The **Replace** command results in the currently selected string being replaced by a new string that you provide. The **Replace and Find Again** command does the same thing as **Replace** and then extends the search to the next occurrence of the string. Finally, the **Replace All** command results in all occurrences of the string being replaced. *Warning: This is a dangerous command and should be used with considerable caution. It cannot be undone with the **Undo** command.*

The last commands in the **Search** menu are **Show Selection** and **Show Error**. **Show Selection** lets you quickly return to the last insertion point in the Edit window that is presently active. This is useful if you have been scrolling through a long file and you would like to return to the last place where text has been inserted. The **Show Error** option allows you to quickly locate the part of your file that has resulted in a compile error. **Show Error** cannot be used to locate a run-time error.

2.2.4 The **Project** Menu

Figure 2.10 shows the THINK Pascal **Project** menu. The first group of command options on this menu (**New Project..., Open Project..., and Close Project**) allow you to create a new project, open an existing project, or close a project. The **Add "filename"** command adds the source program in the current Edit window to the Project window. It is assumed that you have previously saved the source file. The Edit window must be active when this command option is selected. The **Add File...** command allows you to add a file to the Project window. This file may be a source file, an object file, or a library file, If the project menu is pulled down with the option key depressed, this command is changed to **Add Files...** . This command option results in a dialog box that allows you to select multiple files from a folder and add them to the Project window in one operation. The **Remove** command lets you remove files from the Project window. In order for this command to work, you must first go to the Project window and highlight the file to be removed.

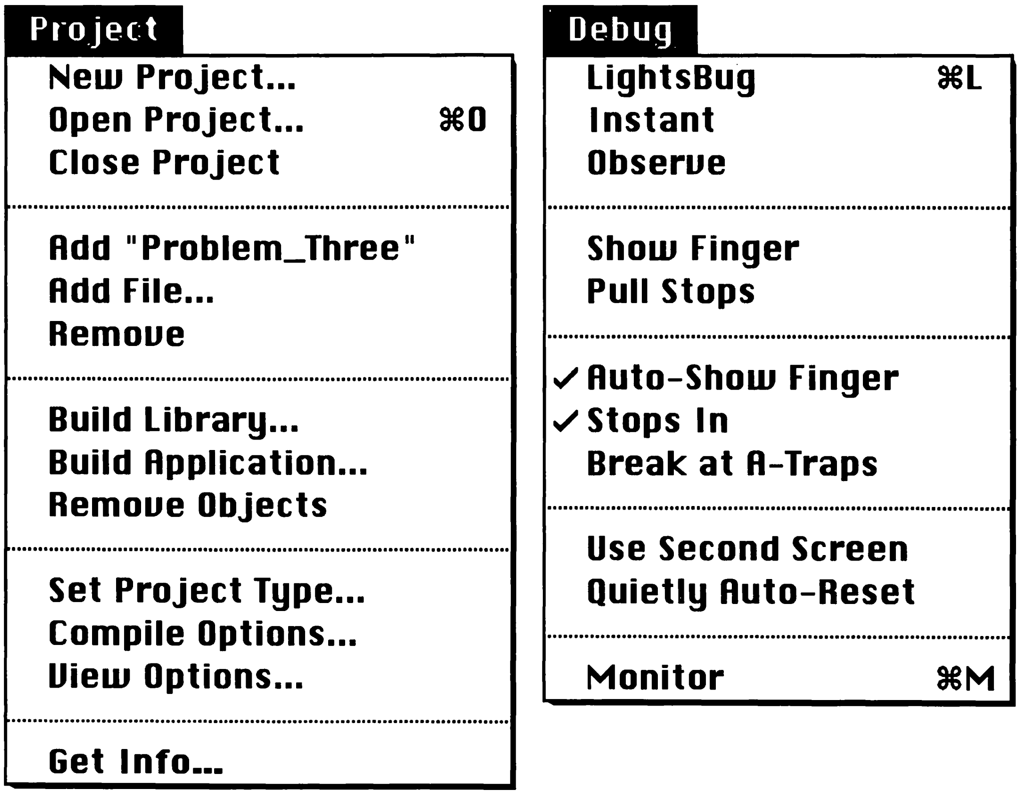


Figure 2.10 The **Project** and **Debug** menus.

The **Build Library**... command allows you to add the current project to your user library for use with future projects. You will be asked to name the file, and the suffix .LIB should be included to identify the resulting file as a library file. The **Build Application...** option allows you to save the current project as an application, a desk accessory, a driver, or a code resource. The **Remove Objects** command option removes all the compiled code from a project. Further discussion of these commands is included in Chapter 8.

The **Set Project Type...** command option allows you to designate a project as an application, a desk accessory, a driver, or a code resource. For details on this option, see the *THINK Pascal User Manual.* The **Compile Options** command allows you to determine several factors in the compiling of your code. These include options for the compilation of source programs particular to specific microprocessors (68020/68030, 68881/68882 or 68000) as well as extended use of the **uses** clause. They also include the ability to profile (analyze) the execution characteristics of a Pascal program. This command option is discussed in a later chapter.

The **View Options...** command allows the appearance of the Project window to be altered and allows the control of run-time options during the execution of a Pascal program. These include checking for integer arithmetic overflow errors, various debugging techniques, and range checking. The status of these options can be determined by the absence or presence of boxes around the options D, N, V, and R beside each file in the Project window. A box around the letter indicates that the option is active.

Finally, the **Get Info** command allows you to obtain information on the size of each file in your project. Selection of this option results in a dialog box that lists each file in the project. Select one of the files in the list to obtain information on the size of that file.

2.2.5 The **Run** Menus

Three versions of the **Run** menu (see Figure 2.11) can be obtained by pulling down the menu alone and in combination with the Option and Shift keys. The Option-key version allows the user to elect automatic options for some of the commands in the original menu. **Go** becomes **Go-Go** and **Step Into** becomes **Step-Step**. The Shift-key version of the menu, not shown in Figure 2.11, changes the **Check Syntax** command to **Compile**.

The command option **Check Syntax** checks the source file in the Edit window for proper use of Pascal grammar. It does not attempt to convert the source code to machine code and has no effect on the Program window. The command option **Compile** checks the source code in the Edit window for proper grammar and semantic meaning. If the source code is correct in the use of grammar and in semantic meaning, this command option also results in the generation of machine code and an update of the Project window.

The next command option, **Build**, forces the system to compile all files that have changed. The **Check Link** command causes all the files in the project to be linked. When the execution of a program has been interrupted, the **Reset** command causes a paused program to return to the beginning. Without **Reset**, a paused program will continue execution from the point of the pause when it is resumed by the **Go** or **Go-Go** command.

The **Go** command results in the execution of the program in the current Edit window. If the file(s) need to be recompiled prior to execution, you will be so informed. The **Go-Go** command (in the Option-key **Run** menu) is the automatic counterpart to the **Go** command. If you have stops inserted in your source code, the **Go-Go** command will result in an execution with a pause at each stop, followed by resumption of the execution.

The **Step Over** command results in the execution of the next line in your source code. If your program includes a function or procedure, the entire routine will be executed when the program line containing the call to the routine is reached. The **Step Into** command is similar to the **Step Over** command, except that when a function or procedure is encountered, the pointer goes to the first line of the routine. This allows you to step through the function or procedure line by line. Functions and procedures are discussed in Chapter 7.

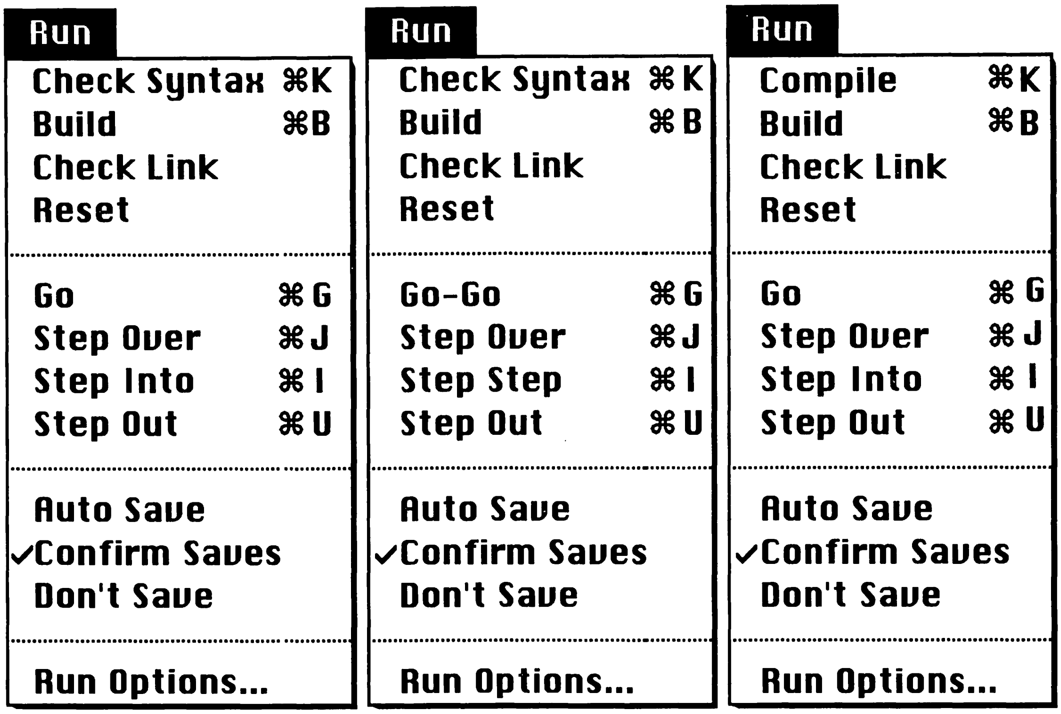


Figure 2.11 The THINK Pascal **Run** menus. The center menu appears when the Option key is depressed, and the right menu appears when the Shift key is depressed.

The **Step-Step** command is an automatic version of the **Step Into** command. It executes each statement in turn with a pause to update the Observe and LightsBug windows. The **Step Out** command causes the program to continue execution until it exits the current routine. If you enter a routine with **Step Into, Step Out** allows you to quickly exit that routine.

The **Auto-Save** option causes THINK Pascal to automatically save source files before the program is executed. The **Confirm Saves** option causes THINK Pascal to present a dialog box asking if you want to save your files when the program is executed. The **Don't Save** command causes THINK Pascal to do nothing about saving the file(s) in the Edit window(s) when the program is executed. If you elect this option, you will have to remember to save the file(s) with no aid from THINK Pascal.

The **Run Options...** command allows you to control several run-time environment settings. These options include selection of the resource file the program will use,[4] the number of characters that the Text window will save, echoing program output to the printer or a file, selection of font and font size for the Text window, and the amount of

[4] For the Macintosh, a resource includes pieces of code and data that support proper executions at specific instants of time. This can include menus containing menu bars, icons and character fonts, layout, and the content of dialog and alert boxes, as well as code within a program. A resource file is a collection of resources stored as a unit on a disk. The *THINK Pascal Resource Utilities Manual* has more information on how resource files can be created and edited THINK Pascal supports several ResEdit resource editors as well tools for compiling and decompiling resource description files.

memory allocated for the project's stack and heap. This dialog box is shown in Figure 2.15 in Section 2.3.2.

2.2.6 The **Debug** Menu

The **Debug** Menu is shown along with the **Project** menu in Figure 2.10. This menu allows the user access to several devices that are useful in locating program bugs. The **LightsBug** command results in the opening of the LightsBug window. The **Instant** command results in the opening of the Instant Window, which can be used for instant execution of individual statements when your program is paused.[5] The **Observe** command results in the opening of the Observe Window. This window allows you to observe the values of variables and expressions during the execution of a program. The **Show Finger** command makes the window that contains an execution error the active window and shows the part of the window that contains the finger. The **Pull Stops** command removes any stops previously inserted in your program. The **Auto-Show Finger** command is an automatic version of the **Show Finger** command. The **Stops In** command allows you to place stops in your program code. To do this, move the cursor to the left part of the Edit window until it takes the shape of a stop sign. Click the mouse button while the stop sign icon is visible, and a stop will be located before the current program line. Stops allow you to step through a program in stages of your own choice. This technique can be quickly refined with a little practice.

The **Break at A-Traps** command allows you to halt execution of your program prior to a call to a Macintosh Toolbox routine.[6] If you have more than one monitor, the **Use Second Screen** option, when active, will use the second screen to display the source and data windows. This option is available only when the **Use Source Debugger** option is on. **Quietly Auto-Reset** will suppress the presentation of a reset warning dialog box, which appears when you attempt to execute a halted program after changing the source code. Finally, the **Monitor** command will move you into the low-level debugger, (Macsbug or TMON).

The Option-key version of the **Debug** menu changes **Pull Stops** to **Pull All Stops**. This command results in the simultaneous removal of stops from all of the files in the project. The Shift-key version of the **Debug** menu presents **New LightsBug** as its first option. This option results in a new **LightsBug** window being opened (up to four can be open at once). This alternative menu also shows the command **Use Monitor** as a replacement for the **Monitor** command. This command lets you determine which debugger you will use when the software encounters an exception. If **Use Monitor** is on, the debugger will be TMON or Macsbug. If it is off, the debugger will be LightsBug.

2.2.7 The **Windows** Menu

Figure 2.12 shows the **Windows** menu. The first command option (shown as **Prob.Project** in Figure 2.12) appears as the name of the project. If there is no project when the menu is displayed, it will appear as **No Project**, and the command will be

---

[5] The Instant window **Do It** button is only active when a program has been halted. For example, it will work when you insert a stop in a program and then activate the **Step Over** command. Or it will work if you start execution of a program with **Go** and then halt it by clicking the spray can.

[6] Macintosh Toolbox refers to the group of ROM libraries supported by the Macintosh system.

unavailable (dimmed). Selection of this command option activates and displays the Project window.

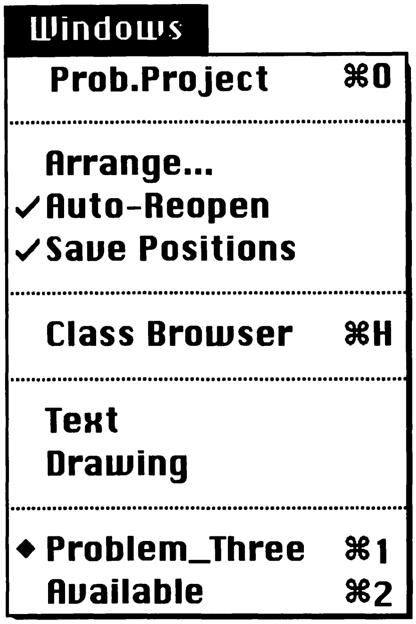


Figure 2.12 The THINK Pascal **Windows** menu.

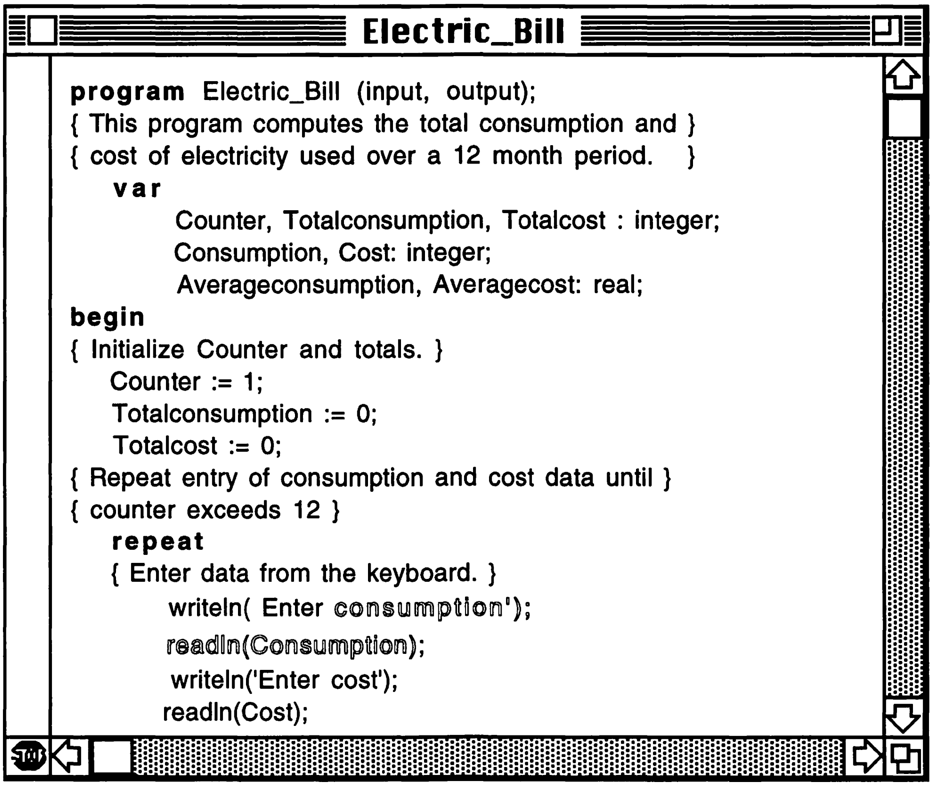
The **Arrange...** command allows you to arrange the Edit windows in several different ways, including overlapping, tiled, side by side horizontally, or side by side vertically. The **Auto-Reopen** command causes the same windows to open automatically when the project is opened. The **Save Positions** command allows you to save the positions of your windows when the project is closed. When the project is reopened, all windows appear in the same position. The **Class Browser** command opens the Class Browser Window, which is useful with object-oriented programming. The **Text** and **Drawing** commands open the Text and Drawing windows, respectively. Finally, there is a list of window commands (shown in Figure 2.12 as **Problem_Three** and **Available**) that allow you to activate any of the currently open Edit windows. A diamond shown by one of these filenames (see **Problem_Three** in the figure), indicates that the file has been edited but the changes have not yet been saved.

2.3 CREATING A SOURCE PROGRAM: MORE DETAIL

As described in Section 2.1, a THINK Pascal program is always represented as a part of a larger entity called a *project*. The project is a collection of linked files that make up the program. To build a program, you need to know how to create the project document, how to create the Pascal source program and edit it, and how to compile it and link it to other files that will serve as subroutines for it. In this section, we explore a few details that should help you understand this overall process and get you started as a THINK Pascal programmer. The details of some of these processes, however, are reserved for later chapters.

2.3.1 Editing a THINK Pascal Program

A critical part of building a project is the creation of a proper source program. This is done in the Edit window. Earlier in this chapter we discussed a program called `Electric_Bill`. You were instructed to type this program into the Edit window and then add it to a project. If you did this assignment successfully, you now have some experience with the Edit window. You should now broaden that experience by experimenting with other programs and by learning how the Edit window can help you overcome problems. For example, many minor programming errors will be identified by THINK Pascal as you work, by the appearance of the outline font in the Edit window where the problem occurs. This is illustrated in Figure 2.13, where the program `Electric_Bill` was changed to show a minor but common error.

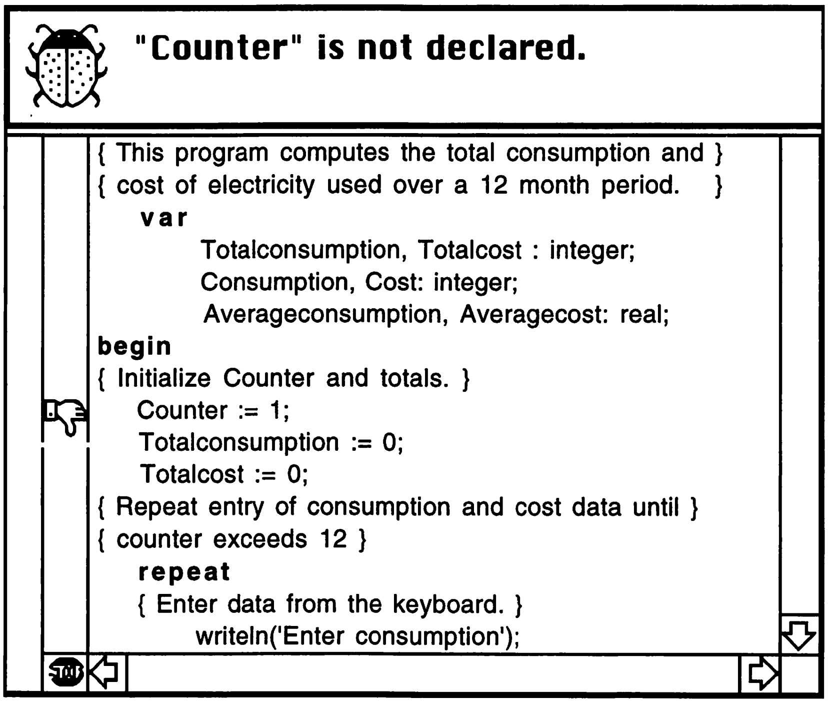


```
program Electric_Bill (input, output);
{ This program computes the total consumption and }
{ cost of electricity used over a 12 month period.   }
   var
         Counter, Totalconsumption, Totalcost : integer;
         Consumption, Cost: integer;
         Averageconsumption, Averagecost: real;
begin
{ Initialize Counter and totals. }
   Counter := 1;
   Totalconsumption := 0;
   Totalcost := 0;
{ Repeat entry of consumption and cost data until }
{ counter exceeds 12 }
   repeat
   { Enter data from the keyboard. }
         writeln( Enter consumption');
         readln(Consumption);
         writeln('Enter cost');
         readln(Cost);
```

Figure 2.13 An error in the source program may be revealed in the **Edit** window in the form of an outline font.

The error in this case is the omission of a single quote in the `writeln` expression. Notice that the resulting output indicates (by means of the hollow outline font) that something is wrong, but the computer does not identify the specific problem. You will have to learn through experience how to interpret a result such as this. Fortunately, it is not difficult to develop this skill.

Another way of spotting errors as you type the program into the Edit window is through the use of the **Check Syntax** command under the **Run** menu. At any time, you can check your source code by selecting this option. For example, modify the `Electric_Bill` source program by removing the word *Counter* from the variable declaration list. Then select the **Check Syntax** command and watch the result. The error message shown in Figure 2.14 will appear. The bug icon in the message informs you that there is an error, or bug, in your program. The description of the bug in the error message should give you a clue about the problem. In this example, the message is very helpful in identifying the problem, since it tells us that the identifier `Counter` has not been declared. Before going on to correct the problem, you must click on the error message box to remove it from the screen.



"Counter" is not declared.

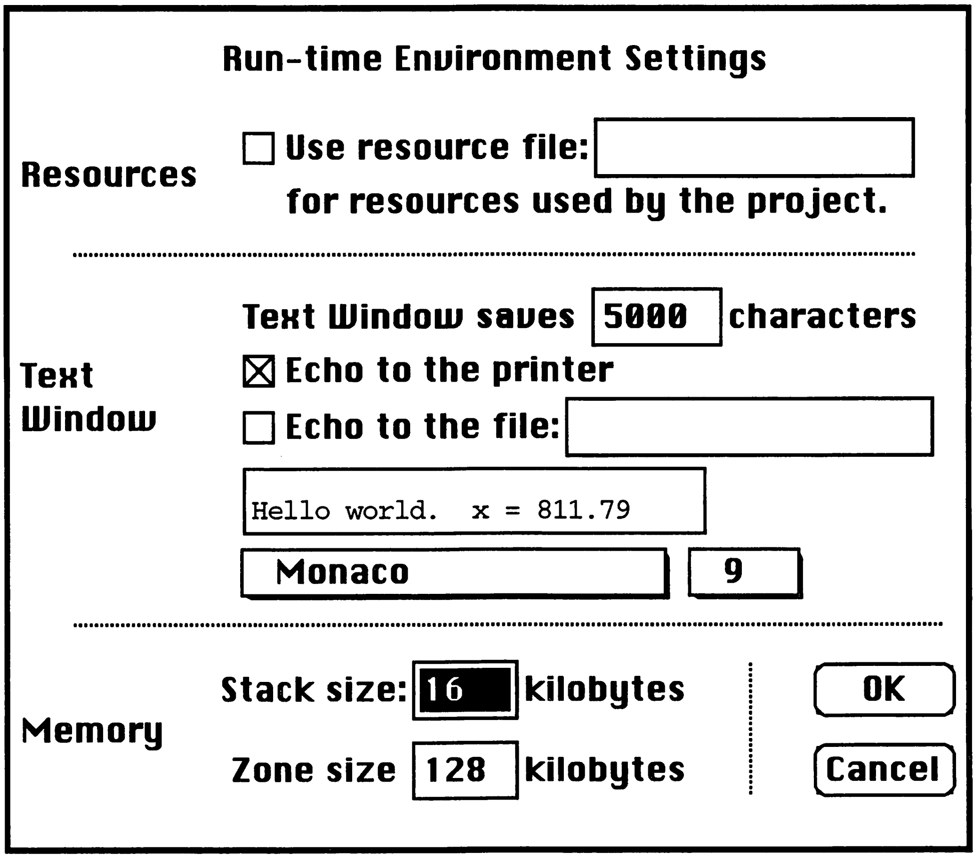
```
{ This program computes the total consumption and }
{ cost of electricity used over a 12 month period.   }
   var
        Totalconsumption, Totalcost : integer;
        Consumption, Cost: integer;
        Averageconsumption, Averagecost: real;
begin
{ Initialize Counter and totals. }
   Counter := 1;
   Totalconsumption := 0;
   Totalcost := 0;
{ Repeat entry of consumption and cost data until }
{ counter exceeds 12 }
   repeat
   { Enter data from the keyboard. }
        writeln('Enter consumption');
```

Figure 2.14 Using the **Check Syntax** command to locate a programming error.

2.3.2 Using the Printer

You can use your printer to make a paper copy (hard copy) of the material displayed in any window. For example, if you would like a hard-copy listing of a Pascal source program, activate the Edit window that contains the source code, pull down the THINK Pascal **File** menu, and select **Print...** A print dialog box (familiar to the experienced Macintosh user) will appear and offer you the usual print options, such as the quality of print, the page range, the number of copies, and paper feed. The options offered will depend upon the printer connected to your computer or the printer you have selected if you are working on a network with several printers available. An example of a print dialog box is shown in Figure 1.18.

After you enter your option choices, the printing will proceed. You may print the contents of the other windows by clicking the desired window (making it active) and then electing the **Print** command on the **File** menu. By selecting the **Run Options...** command option of the **Run** menu, you may elect to have the output of your program directed to the printer as well as to the screen. Figure 2.15 shows the dialog box when this option is selected.


Run-time Environment Settings

Resources
☐ Use resource file:
for resources used by the project.

Text Window
Text Window saves 5000 characters
☒ Echo to the printer
☐ Echo to the file:

```
Hello world.  x = 811.79
```

Monaco 9

Memory
Stack size: 16 kilobytes
Zone size 128 kilobytes

Figure 2.15 Directing output to the printer. The dialog box from the **Run Options...** command under the **Run** menu.

2.3.3 Creating a Generic Project: A Helpful Shortcut

As you get into Chapter 3 and begin entering many short Pascal programs, testing and exploring various points discussed in the text, you will find the process of creating a project for each source program to be both tedious and costly in disk space. As a result, you might want to create a generic project that you can use with any source program to test its execution. Create a project in the usual way and name it General project. When the source file has been typed into the Edit window, use the **Add "filename"** command under the **Project** menu to add the source file to the project. Compile and execute the program, and observe the effect. When you are done, select the source file name in the Project window (drag the cursor over the name), and click **Remove** under the **Project** menu. The generic project is then ready to receive the next source file. You might also want to keep a folder to collect the source programs as you type them. It is efficient to collect source programs, since they do not take up very much room on disk. Having these files available will save you time later, since some of the early programs may need to be modified and improved as you learn additional techniques. Remember, however, that you will have to add the source file to a project and compile it before you can execute it again.

2.3.4 Creating an Instant Project

Another shortcut you might find helpful is the instant project. When you select the **New Project...** option from the **Project** menu, you will see an Instant Project box near the bottom. Click this box, and type a name for your project in the appropriate place on the dialog box. Do not try to include a suffix in the name (such as π in the name Project.π). THINK Pascal will automatically create and name the folder, the project file, and an outline source file (in the Edit window), naming each with the name you provided and attaching an appropriate suffix. The outline source file is similar to the one you see in Macintosh Pascal when the Program window is first opened. It can be used or discarded using the **Delete...** command option from the **File** menu.

SUMMARY

This chapter reviews the menus and command options of THINK Pascal, a development environment for generating Macintosh applications. It is different from Macintosh Pascal because it supports a project manager that binds files into an entity called a *project*. THINK Pascal supports a fast compiler for translating Pascal source code, an advanced text editor for Pascal syntax, an automated make capability for rebuilding source files, advanced debugging tools such as a debugger and profiler, and a class library for development of object-oriented programs.

The **File** menu contains the command options **New**, **Open...**, **Close**, **Close All**, **Save**, **Save All**, **Save As...**, **Save a Copy As...**, **Revert**, **Page Setup...**, **Print...**, **Print All Files**, **Delete...**, **Transfer...**, and **Quit**. Where Macintosh Pascal allows only a single window for program development, THINK Pascal allows several Edit windows to be opened, using either the command option **New** or the command option **Open...**.

The **Edit** menu supports standard Macintosh command options such as **Undo**, **Cut**, **Copy**, **Paste**, **Clear**, **Select All**, **Show Clipboard**, as well as added command options such as **Source Options...**, **Auto-Reformat**, and **Projector-Aware**. The **Search** menu supports the command options **Find...**, **Find Again**, **Find in Next File**, **Find in All Files**, **Enter Selection**, **Replace**, **Replace and Find Again**, **Replace All**, **Show Selection**, and **Show Error**. These command options allow more string search and replacement options than found in Macintosh Pascal.

The **Project** menu has numerous command options for working with the current project, including **New Project...**, **Open Project...**, **Close Project**, **Add "filename"**, **Add File...**, **Add Files...**, **Remove**, **Build Library...**, **Build Application...**, **Remove Objects**, **Set Project Type...**, **Compile Options...**, **View Options...**, and **Get Info...**. The **Run** menu has command options that support syntax checking, the translation and building of files, as well as control over execution of a THINK Pascal program. These commands include **Check Syntax**, **Compile**, **Build**, **Check Link**, **Reset**, **Go**, **Go-Go**, **Step Over**, **Step Into**, **Step-Step**, and **Step Out**. Additional command options include **Auto Save**, **Confirm Saves**, **Don't Save**, and **Run Options....**

The **Debug** menu allows for some of the same windows to be opened when analyzing the execution of source code as in Macintosh Pascal, as well as added command options not in Macintosh Pascal. These include **LightsBug**, **New LightsBug**, **Instant**, **Observe**, **Show Finger**, **Pull Stops**, **Pull All Stops**, **Auto-Show Finger**, **Stops In**, **Break at A-Traps**, **Use Second Screen**, **Quietly Auto-Reset**, **Monitor**, and **Use Monitor**. The **Windows** menu allows the programmer to work with and position many of the various THINK Pascal windows. The command options under this menu include the Project window name, **Arrange...**, **Auto-Reopen**, **Save Positions**, **Class Browser**, **Text**, **Drawing**, as well as a list of all file names for current Edit windows.

Pressing the Shift or Option keys while the menu is down produces alternative commands for the **File**, **Search**, **Project**, **Run**, and **Debug** menus. In addition to the command options under these THINK Pascal menus, there are numerous dialog windows offering options that can affect the viewing of Edit, Text, and Project windows. With the dialog window produced by selecting the **Run Options...** command, the programmer can direct text from the Text window to a printer as well as to a file. In addition, the type of font, font size, and the total number of characters saved in a Text window can be changed.

REVIEW QUESTIONS

1. What is a project?
2. What are the general steps involved in the creation of a project?
3. What characters can be used in titling a THINK Pascal source file?
4. What is the maximum number of characters for a Macintosh file name?
5. How do you rename a file that has already been created when using THINK Pascal?
6. What differences have you seen so far in Macintosh Pascal and THINK Pascal?
7. The first line of any THINK Pascal program begins with what command?
8. What are the rules for naming an identifier?
9. What are the eight menu options for THINK Pascal?
10. What command options exist when selecting the **Run** menu?
11. What command options exist when selecting the **Edit** menu?
12. What command options exist when selecting the **Window** menu?
13. What is the purpose of the **Project** window?.
14. What is the purpose of the **Check Syntax** option?
15. What is meant by the term *debugger*?
16. How can stops be inserted and removed in a THINK Pascal source program?
17. What is the purpose of the **Observe** window? How is this window used to locate a program bug?
18. What are the first steps in writing a THINK Pascal program?
19. What is the difference between the options **Print** and **Print All** ?
20. How can you edit your program after it has been compiled?
21. What is the purpose of the Clipboard?
22. What is the purpose of the Text window? The Drawing window? The Instant window?

23. What is the purpose of the **Reset** option??
24. How can a program displayed in the Program window be printed?
25. How can output from a program be directed to a printer?
26. How do you display alternate forms of the THINK Pascal menus? Which menus have alternate forms?
27. How does one create an Instant Project?
28. What is the purpose of a generic project? What is the advantage of this approach when writing many small programs?

PROGRAMMING EXERCISES

1. Create a generic project for the purpose of testing the following program as well as the remaining programming exercises in this chapter. After the project has been created, enter the following test program in a new Edit window. This program simply displays a message in the Text window:

```
program Exercise_One(input, output);
{ Purpose: This program will be used to test the Instant window. }
begin
   writeln(' This is a sample program for testing the Instant
                window.');
end.
```

   Using the command option **Check Syntax**, check the program `Exercise_One` for syntax errors. Correct any errors that exist. Be sure to use the **Save As** and **Save** options for saving the source code to a file. To test the Instant window, place a stop at the left of the keyword **begin** and then execute this program using the **Go** command. The program will be halted with a finger pointing to the left of the word **begin**. Open the Text and Drawing windows from the Windows menu and then choose the Instant command from the Debug menu. Enter the following statements into the Instant window. After entering a source line, press the "**Do It**" button, and observe the response. This exercise will require the following command options: **New Project**, **New**, **Save As**, **Save**, **Add "filename"**, **Check Syntax** (or **Compile**), **Stops In**, **Go**, **Text**, and **Drawing**.

2. Using the generic project from Exercise 1, remove the current program using the **Remove** command after highlighting the Pascal program in the Project window. Then, after closing the current Edit window, apply the command **New**, and enter the following Pascal program.

```
program Exercise_Two(input, output);
{ Purpose: This program prompts for and displays your name.}
var
   Name : string[30];
begin   { Body of main program.}
   ShowText;
```

```
{ Prompt for a name entered from the keyboard. }
   write('Enter your full name and then press the return key: ' );
   readln(Name);
{ Display the name entered from the keyboard. }
   writeln('Your name is ', Name);
end.
```

Apply the **Check Syntax** command, and correct any improper syntax. Now apply the following options from the **Source Options...** dialog window: select font as Geneva, 10 point; select keywords lowercase and underlined; select indentation and tabs at 8 spaces. Now use the command option **Add "filename"**, and add the program to the Project window. Apply the command **Build** to complete building the project, and execute the program using the command **Go**.

3. For the program in Exercise 2, select the following options from the **Run options...** dialog box: for the Text window, select a font of type Geneva and a font size 20; for the Text window select saving only 10 characters. If a printer is available, select the option to echo to the printer. Then again build the project with the **Build** command, and observe execution of the program, using the **Go** command. Observe the execution of the program when you attempt to enter a name longer than 30 characters. Try using the commands **Go**, **Go-Go**, **Step Into**, **Step-Step**, and **Step Over** to continue execution.

4. For the program in either Exercise 2 or 3, remove the present Pascal program from the Project window, and add the following source program in a new Edit window:

```
program Exercise_Four(input, output);
{ Purpose: This program displays 10 numbers to the Text window. }
var
   Counter : integer;
begin
{ Hide all windows but the Text Window. }
   HideAll;
   ShowText;
{ Display 10 numbers to the Text window. }
   for Counter := 1 to 10 do
         writeln( Counter );
end.
```

Apply the **Compile** command option, and correct any improper syntax. Now apply the following options from the **View Options...** dialog window: select the font as Courier, 10 point; set all of the other options to your taste. Now look at the Project window and observe its new characteristics. Using the command option **Add "filename"**, add the Pascal program to the Project window. Apply the command **Build** to complete building the project, and execute the program with the command **Go**. Notice that the Pascal command HideAll will hide all of the windows before it opens the Text window. How can the Edit window be opened to again view the source file after the program ends execution?

5. Modify the program in Exercise 4 by removing the comments in the body and modifying the comment representing the purpose. In addition, cut the statements `writeln`, `HideAll`, and `ShowText`. Your Edit window should now have the following listing:

```
program Exercise_Four(input, output);
{ Purpose:  This program provides a trace during execution by }
{           displaying 10 numbers in the Observe window. }
var
   Counter : integer;
begin
   for Counter := 1 to 10 do
        ;
end.
```

Using the command option **Stops In**, place a stop before the line containing the word **begin**. Now open the Observe window, and enter the name `Counter` at the top of right column. Then apply the command option **Go** and begin execution. Since the program will be interrupted, choose the command **Step-Step** to observe the values of `Counter` as the program executes. Be sure that the box D is set for the Pascal program file listed in the Project window.

6. Modify the program in Exercise 5 to appear as shown below:

```
program Exercise_Four(input, output);
{ Purpose:  This program provides a trace during execution by }
{           displaying 11 numbers in the Observe window. }
var
   Counter : integer;
begin
   Counter := 1;
   while Counter <= 10 do
      Counter := succ( Counter );
end.
```

Add the expression " `Counter <= 10` " below the word `Counter` in the Observe window. Remove the stop before the word **begin** and place a stop before the word **while**. Again execute the program with the command **Go**. Continue to execute the program with the command **Go**, and observe all the values for the two expressions in the Observe window. Be sure that the box D is set for the Pascal program file listed in the Project window.

7. After removing the file for the present Pascal program from the Project window, create a new Edit window, and add the following Pascal program:

```
program Random_Patterns(input,output);
{ Purpose:  This program draws ovals randomly in the Drawing }
{           window. }
const
   Limit = 220;
var
   Area : Rect;
   Counter : integer;
   Left, Top : integer;
begin
{ Open the Drawing window. }
   ShowDrawing;
{ Draw patterns in Drawing window. }
   for Counter := 1 to Limit do
      begin
      { Establish coordinates for the upper left corner of the }
      { rectangle called Area. }
         Left := - random mod 512 + 512;
         Top := - random mod 342 + 342;
      { Establish the rectangle for drawing an oval. }
         SetRect(Area, Left, Top, Left + 150, Top + 75);
      { Draw an oval filled with a black background. }
      { Intersection with any region of the Drawing window, }
      { being black, produces white.}
         InvertOval(Area);
      end;
end.
```

After you have entered the program and removed all errors, add the program to the Project window and execute. You may need to open the Drawing window and adjust its size before execution.

8. The following program will result in an error during execution. Enter the program in your generic project, and observe the bug's box that is generated when `Counter` reaches a value of 10:

```
program Exercise_Eight( input, output);
{ Purpose: This program displays 11 numbers to the Text window. }
var
   Counter : 1..10;
begin
{ Hide all windows but the Text Window. }
   HideAll;
   ShowText;
{ Display 10 numbers to the Text window. }
      Counter := 1;
   while Counter <= 10 do
         begin
               writeln( Counter );
               Counter := succ( Counter );
         end;
      writeln( Counter );
end.
```

Be sure that the boxes D and R are set for the Pascal program file in the Project window. What is the last value of `Counter` displayed in the Text window? After you have observed the appearance of the bug's box reporting the error, remove the option R for the Pascal program file in the Project window, and again execute the program. What is the last value of `Counter` displayed in the Text window?

9. The following program will result in an error during execution, provided that option V is set for the source file in the Project window. Enter the program in your generic project, and observe the bug's box that is generated when 1 is added to `Number`.

```
program Exercise_Nine (input, output);
{ Purpose: This program displays 11 numbers to the Text window. }
var
   Number, Counter: integer;
begin
{ Hide all windows but the Text Window. }
   HideAll;
   ShowText;
{ Display 10 numbers to the Text window. }
   Number := 32767;
   for Counter := 1 to 10 do
      begin
         writeln(Number);
         Number := Number + 1;
      end;
   writeln( Number );
end.
```

Remove option V from the Project window for this file, and again execute the program. What values appear for `Number` in the Text window? What is wrong with these values?

10. Implement the following program, using your generic THINK Pascal project:

```
program Exercise_Ten;
{ Purpose:  This program draws a series of nested squares in the }
{           Drawing window. }
begin
{ Display the Drawing window to the screen. }
```

```
   ShowDrawing;
{Set PenSize for 15 wide and 15 high. }
   PenSize(15, 15);
   MoveTo(5, 20);
   WriteDraw('(25, 25)');
   MoveTo(145, 20);
   WriteDraw('(175, 25)');
{ Draw the first square. }
   DrawLine(25, 25, 160, 25);
   LineTo(160, 160);
   LineTo(25, 160);
   LineTo(25, 25);
 { Draw the second square. }
   DrawLine(55, 55, 130, 55);
   LineTo(130, 130);
   LineTo(55, 130);
   LineTo(55, 55);
 { Draw third square. }
   DrawLine(85, 85, 100, 85);
   DrawLine(100, 100, 85, 100)
end.
```

11. Implement the following program using your generic THINK Pascal project. Selecting the command option **Source Options...** , set all options and establish indentation and tabs at 5 spaces. Your Edit window should appear as in the listing that follows:

```
program Exercise_Eleven;
{ Purpose:  This program draws random circles having random }
{           patterns within the Drawing window. }
 var
   Top, Left, Bottom, Right: integer;
   Pat: Pattern;
   Background_Pattern: integer;
begin
{ Show the Drawing window before painting any ovals. }
   ShowDrawing;
{ Use function random to choose the corners of a square and a }
{ background pattern. }
   while true do
      begin
      { Randomly select a rectangle for drawing an oval. }
         Top := abs(random) mod 201;
         Left := abs(random) mod 201;
         Bottom := Top + 30;
         Right := Left + 30;
      { Randomly select the background pattern. }
         Background_Pattern := abs(random) mod 5;
         case Background_Pattern of
            0:
                Pat := white;
             1:
                Pat := black;
             2:
                Pat := gray;
             3:
                Pat := ltgray;
             4:
                Pat := dkgray;
          end;
{ Display the oval in a rectangle with a background pattern from }
{ the above step. }
          FillOval(Top, Left, Bottom, Right, Pat);
       end;
end.
```

This program will continue to execute indefinitely. You can terminate execution by clicking the spray-can icon to the far right of the menu bar.

12. If you have a color monitor, add the following Pascal code to the body of the program of Exercise 11. This code should follow the case statement that assigns a value to variable `Pat`. When executed, the modified program displays ovals, using random background colors as well as random patterns:

```
{ Randomly select the background color. }
   Background_Color := abs(random) mod 8;
      case Background_Color of
         0:
            Color := blackColor;
         1:
            Color := whiteColor;
         2:
            Color := redColor;
         3:
            Color := greenColor;
         4:
            Color := blueColor;
         5:
            Color := cyanColor;
         6:
            Color := magentaColor;
         7:
            Color := yellowColor;
      end;
{ Display the oval in a rectangle with a background color chosen }
{ from the above step. }
   BackColor(Color);
```

Be sure to add the following declarations under **var**:

```
Background_Color : integer ;
Color : longint;
```

Replace the command `BackColor(Color)` with `ForeColor (Color)`, and observe the change in appearance for ovals drawn on the screen.

13. Using your own programs, try the command options **Save a Copy As**, **Transfer**, **Delete**, **Remove Objects**, **Find**, **Find Again**, **Close Project**, **Open Project**, **Add File...**, **Quitely Auto-Reset**, and **Show Finger**.

Chapter 3

Constants, Variables, and Simple Input and Output

OBJECTIVES

After completing Chapter 3, you will know the following:

1. An introduction to problem solving and the algorithm.
2. The general format used in both Macintosh Pascal programs and THINK Pascal programs, including the use of the program heading, the identifier, declarations, and the reserved words `begin` and `end`.
3. The concept of a data object, including constants and variables.
4. The use of the Macintosh Pascal and THINK Pascal input commands `read` and `readln` and the output commands `write` and `writeln`.
5. Simple Macintosh data types, including the `real` data types (`real`, `double`, `extended`, and `computational`) and the `ordinal` data types (`integer`, `longint`, `char`, `Boolean`, `enumerated`, and `subrange`).

3.1 PROBLEM SOLVING

Programming involves the following steps: (1) identifying a problem that requires a solution, (2) finding the solution to the problem, (3) specifying the ordered set of steps that represent the solution, and (4) implementing these steps in a computer language. In reaching a solution to a problem, we must be concerned with analyzing the problem, finding the steps for a solution, and then formally defining the set of steps. A computer language such as THINK Pascal is a tool for implementing our solution on a computer. Although it may appear to the beginner that the computer provides an answer to a problem when a program is executed, the computer itself does not directly solve problems; it is programmed to provide an answer or answers for either one problem or a class of problems.

The first step in obtaining a solution to a problem is problem analysis: the problem is defined, and all the information needed for a solution is identified. Problem analysis consists of the following steps:

1. *Define the problem precisely.* This implies being able to write a short description defining the problem to be solved. If this cannot be done, more time is needed to think about the problem.
2. *Determine whether the problem has already been solved.* Is it possible that programs already exist for performing the task being studied? Is it possible to modify an existing program to provide a solution?
3. *List all the desired information required as input.* What is required as input to solve this problem? If you cannot recognize the input requirements, more thinking is required about the problem you are trying to solve.
4. *List all the desired information required for output, including a representation of an answer to the problem being solved.*
5. *Begin with an initial set of steps as an approximation of a solution.* Do these initial steps identify any subproblems that need to be solved (need to be broken down into smaller steps)?
6. *Refine the steps so that they are precise and explicit.* This is important because a computer program will be based on these steps. If they are not precise and explicit, it may not be possible to translate the solution into explicit computer commands. Often it is necessary to repeat Steps 1 through 6, refining the solution in stages.
7. *Trace each step of the solution with known information.* This allows us to understand how intermediate values are created and changed and to detect steps that are imprecise or not explicitly defined.

The product of this process is an *algorithm*. An algorithm is a procedure having a finite number of unambiguous steps specifying a sequence of operations that provide a solution to a problem. Consider some of the key words in this definition. First, an algorithm is a procedure. By following the steps of an algorithm, we can obtain a solution to a problem. Second, the steps are finite; they do not go on forever. Third, the total number of steps is not fixed for all problems. Finally, each step is unambiguous; that is, it is precise and explicit. A proper algorithm must satisfy the following characteristics:

1. *Finiteness.* There must exist a finite sequence of steps leading to an answer. If this is not *true*, further analysis is required.
2. *Definiteness.* There must be preciseness of meaning. Each step must be explicit and precise in defining its actions. If any step fails to have this property, it should be treated as a subproblem in itself and subjected to further analysis.
3. *Input.* A list of information to be entered through what we call *input data objects* or *input variables*.
4. *Output.* A list of information to be reported through what we will call *output data objects* or *output variables*.
5. *Effectiveness.* All steps must be able to be completed in a finite length of time by any individual tracing the steps of the algorithm.

If an algorithm fails to satisfy one or more of these five properties, it is necessary to perform additional analysis. It is important to understand that a computer program is simply one form for expressing an algorithm. A computer program that fails to execute properly also fails to satisfy one or more of the characteristics of an algorithm.

The steps of an algorithm can be expressed in several forms. It is possible to express them in a natural language such as English. The problem with this approach is that the English language can be ambiguous and thus lead to confusion. A second approach to expressing an algorithm is symbolic representation. Each step of the algorithm can be represented by means of a flowchart symbol. Unfortunately, there are numerous commands in THINK Pascal for which no standard flowchart symbols exist. In addition, developing large algorithms with flowcharts is messy. A third approach is to use an artificial language similar to the commands of Pascal for describing the steps. This is the approach that we will take, and later you will be shown how Pascal itself can serve as a vehicle for defining algorithms.

3.1.1 Developing an Algorithm: An Example

An example will help clarify the foregoing discussion. The following discussion is keyed to the steps just presented.

1. Suppose that we want to use our computer to compute average monthly consumption and cost for our electric bills over a 12-month period. Assume that we will first enter 12 values each for consumption and cost and then display the following messages

   ```
   Average Monthly Consumption :
   Average Monthly Cost        :
   ```

   with the computed values printed to the right of the colons. This description of what we want to do constitutes our definition of the problem.
2. Next, determine if the problem has already been solved. While it may be possible to purchase software for the Macintosh that will perform these calculations, we will assume that a suitable program does not exist. Thus, we must provide our own solution to the problem.
3. List the necessary input for a solution. Obviously, we need our electric bills for the past 12 months, and from these we must take the figures on consumption (the number of kilowatt-hours used) and cost (the dollar amount charged for our electric usage for each month).
4. Then list the information required for output. In the case of our electric bill, we need the average consumption and the average cost. Our output will also contain labels that will identify these figures.
5. Next we must provide an initial set of steps for the solution of our problem. These might appear as follows:

   (a) Prompt the user with a message to enter the month's consumption and cost figures.
   (b) Add consumption to a partial sum for storing total consumption over 12 months. Follow the same procedure for cost.
   (c) Repeat Steps (a) and (b) 11 times.

(d) When we have completed these 12 iterations, compute the averages by dividing the sums for consumption and cost by 12.
(e) Report these average values to the user.

These steps represent an algorithm, but do they satisfy all the requirements of an algorithm? First, the steps are finite even though Steps (a) and (b) are to be performed 12 times. Second, some of these steps must be more precise. For example, Step (b) should be more specific as to what names can be used for representing the partial and total sums. We must also be more specific on how we intend to control the iteration of Steps (a) and (b). That is, we should introduce a counter to control these iterations. Third and fourth, the steps for input and output will have to be more precise in expressing these two actions. Fifth, if you trace the steps by hand (using pencil and paper), this algorithm can be executed in a finite number of steps.

6. The sixth step in the process is to refine the Steps (a) through (e) of Step 5 so that they are precise and explicit. Because of the need for greater precision, we will rewrite those steps in a more formal style that resembles the style of a Pascal program. For now, you should at least be aware that the use of the braces { } designates a comment, which plays no active part in the program itself.

```
(a){ Prompt user to enter both consumption and cost }
   { values from the keyboard. }
   write 'Enter Consumption'
   read Consumption
   write 'Enter Cost'
   read Cost
(b){ Compute the partial summations. }
   Total_Consumption <-- Total_Consumption +
                                Consumption
   Total_Cost <-- Total_Cost + Cost
(c){ If we have not yet reached a count of 12, go }
   { back to Step (a) and repeat Steps (a) and (b). }
   If count of 12 or less, go to Step (a)
(d){ Compute the average values of consumption and }
   { cost. }
   Average_Consumption <-- Total_Consumption / 12
   Average_Cost <-- Total_Cost / 12
(e){ Display the results. }
   write 'Average Monthly Consumption:',
                                Average_Consumption
   write 'Average Monthly Cost: ',Average_Cost
   { End of solution. }
```

These steps are finite, but there is still some ambiguity; when we retrace the steps, we find that no initial values have been set for `Total_Consumption` and `Total_Cost`. We might assume these values will initially be set to zero, but we have done nothing in our algorithm to ensure this. This should be done prior to Step (a). We also need to set up a counter that will start at 1 and increase in increments of 1 with each iteration of the data-entry cycle. This counter will be tested to determine if it is less than 12, in which case the cycle should be continued, or if it is time to compute the averages.

The following instructions incorporate solutions to these problems and refine the algorithm for computing the average power consumption for one full year.

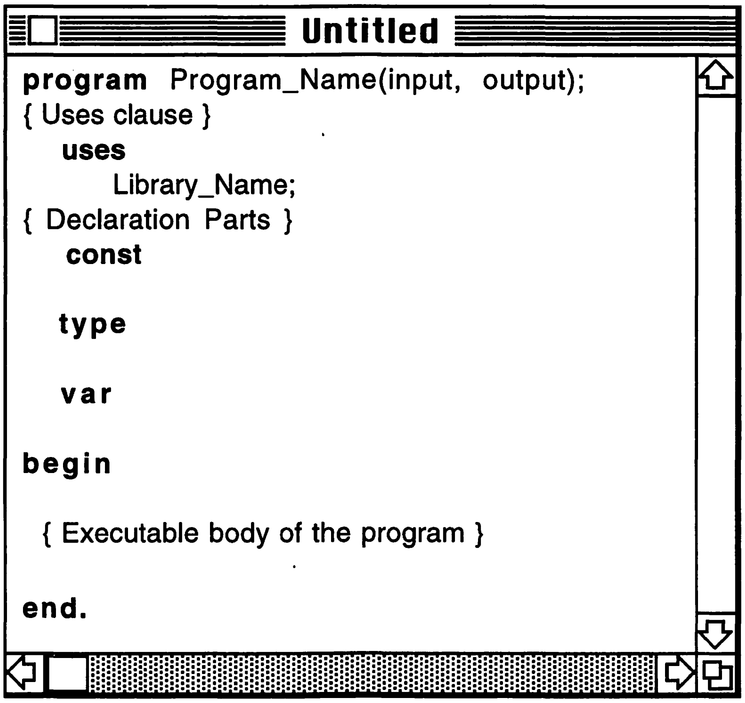
```
Algorithm Electric_Bill;
{ Initialize counter and totals. }
   Counter <-- 1
   Total_Consumption <-- 0
   Total_Cost <-- 0
{ Repeatedly enter consumption and cost until counter
   exceeds 12. }
   repeat
      write 'Enter Consumption'
      read Consumption
      write 'Enter Cost'
      read Cost
   { Compute the partial summations. }
      Total_Consumption <-- Total_Consumption +
                                   Consumption
      Total_Cost <-- Total_Cost + Cost
      Counter <-- Counter + 1
   until (Counter is greater than 12)
{ Compute the average values of consumption and cost.}
   Average_Consumption <-- Total_Consumption / 12
   Average_Cost <-- Total_Cost / 12
{ Display the results. }
   write 'Average Monthly Consumption: ',
                                   Average_Consumption
   write 'Average Monthly Cost: ', Average_Cost
{ End of algorithm. }
```

What about the algorithms for the `read` and `write` commands? For the present we will accept these as given, just as we accept the basic operations of addition and division. For your reference, the finished program is shown in Section 2.1 of Chapter 2.

3.2 THE FORMAT OF A PASCAL PROGRAM: ADDITIONAL DETAIL

A Macintosh Pascal or THINK Pascal program is composed of several parts, including a program heading, a **uses** clause, a declaration part, and a statement part. For now, we will refer to the statement part as the executable body of the program. Figure 3.1 shows the typical format for a Macintosh Pascal and THINK Pascal program.

In this figure the program heading begins with the reserved word **program** followed by an identifier representing the program title. We refer to **program** as a reserved word because it can only be used in the context for which it is defined (in this case defining the beginning of a Pascal program). Any attempt to use the word **program** in a context other than defining the beginning of a Pascal program results in this word being displayed in outline type, indicating that a syntax error has occurred. Appendix A includes a complete list of reserved words for both the Macintosh Pascal language and the THINK Pascal language.



```
program Program_Name(input, output);
{ Uses clause }
   uses
      Library_Name;
{ Declaration Parts }
   const

   type

   var

begin

  { Executable body of the program }

end.
```

Figure 3.1 Format for a Macintosh Pascal or THINK Pascal program.

Since the program title is an identifier, it is subject to the rules discussed in Section 1.3 of Chapter 1. Specifically, an identifier must begin with a letter of the alphabet, and this letter can be followed by letters of the alphabet, the digits 0 through 9, and/or an underscore (_) character. Other characters, including blanks, are illegal. In addition, an identifier can be up to 255 characters long. Upper- and lowercase letters are not interpreted as being different in identifiers; their use is completely at your discretion. Our convention in naming identifiers is to begin with a capital letter followed by letters or digits, with the underscore being used to separate full words. For example the name `Taxableincome` is represented as `Taxable_income` or `Taxable_Income`. You may prefer not using the underscore and beginning each distinct word with a capital letter. For example, you could use the name `TaxableIncome`. Remember that typing a blank or hyphen (dash) to separate words, such as `Taxable-income` or `Taxable Income`, will result in a syntax error. Finally, notice that we are using a natural name for taxable income rather than a cryptic phrase such as `TXI`, `Txi`, or `T_X_I`. An important rule in writing algorithms and in naming identifiers is to use natural names. Although they may take longer to type, they are easier to remember than cryptic names, and they allow for better program documentation.

The program heading can also be represented by the following statement: **`program`** `Program_Name(input,output);`. The two words `input` and `output` are referred to as *program parameters*. For some Pascal translators they are required even when standard input (the keyboard) is used for entering data and when standard output (the screen) is used for displaying data. In both Macintosh Pascal and THINK Pascal, this form is optional when writing programs.

Following the program statement is the **uses** clause. **Uses** was first introduced in the UCSD p-System. This statement directs the Pascal interpreter to find and include the libraries contained in the list of names that follow the reserved word **uses**. Although this clause is not always required, it allows special, predeclared constants, types, and programs to be borrowed from one or more library units and included within a Pascal program. A library or library unit contains a collection of special, predeclared objects and programs that a program can borrow. The **uses** clause is particularly important when writing THINK Pascal programs (as compared to Macintosh Pascal). A THINK Pascal program can borrow from more than 56 Macintosh libraries. An additional 36 libraries can be referenced without the **uses** clause, since they are built into THINK Pascal. In Macintosh Pascal, only three libraries can be referenced by the **uses** clause: QuickDraw1, QuickDraw2, and SANE.

After the program heading and the **uses** clause, the Pascal program may contain **constant**, **type**, and **variable** declarations. These statements list any constants, programmer-defined types, and variables to be encountered in the program. They begin with the reserved words **const**, **type**, and **var**. The executable portion of the program is enclosed between the reserved words **begin** and **end**. Notice that a period is required to terminate the last **end** of a Pascal program. Statements enclosed in special braces { } are comments and have no effect on the execution of the program.[1] However, the comments can be important to those reading the program or in documenting the program for future reference.

In Chapter 1 we listed a program named `Circle`, an example of a simple Pascal program. Its purpose is to "paint" a circle in the Drawing window by executing the code listed in the program window. Figure 3.2 gives the Macintosh Pascal listing for the program.

Taking the program line by line, we see the following:

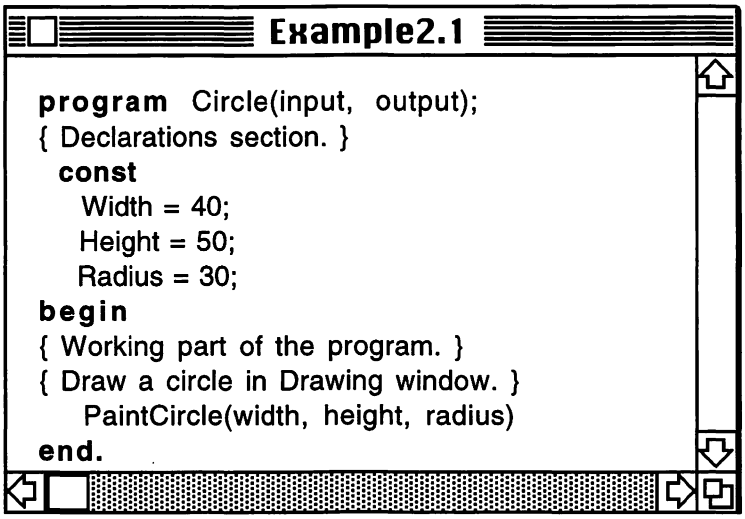
1. The program heading identifies the title of the program as `Circle`. Since no parameters or **uses** clause are necessary, that portion of the program heading is omitted. The line ends with a semicolon as required.

2. The next line in the program is a comment line identifying the declarations section of the program. Three constants are declared under the keyword **const**. They define the circle to be painted. The center of the circle is located by its *x* and *y* coordinates, named `Width` and `Height` and assigned the values of 40 and 50, respectively. Next the radius of the circle is set as a constant called `Radius` and assigned a value of 30. Each of the constant declarations is followed by a semicolon. These constants, `Width`, `Height`, and `Radius`, are three identifiers in our program.

3. The body of the program consists of two additional comments and the command `PaintCircle`, which directs the machine to draw the circle. The information necessary for painting the circle comes from the three arguments, `Width`, `Height`, and `Radius`, separated by commas and enclosed in parentheses.

4. This statement, along with comments, is bracketed between the two reserved words **begin** and **end**. This represents the executable body of the program.

[1] In THINK Pascal, compiler directives can be inserted between comment brackets. By inserting these special strings between comment brackets the programmer may direct the compiler to generate different machine code than it normally would.



```
program Circle(input, output);
{ Declarations section. }
  const
    Width = 40;
    Height = 50;
    Radius = 30;
begin
{ Working part of the program. }
{ Draw a circle in Drawing window. }
    PaintCircle(width, height, radius)
end.
```

Figure 3.2 The Macintosh Pascal version of `Circle`.

3.3 THE CONCEPT OF A DATA OBJECT

Figure 3.3 outlines the key features of a data object. As the illustration shows, a data object can be either a constant or a variable. It consists of an identifier associated with a name, one or more attributes (properties), and a value. The properties or attributes of the data object depend on its type. Variation in data types is one of the main sources of richness in Pascal. We will discuss some of the alternative types in detail in a later section of this chapter.

3.3.1 Constants

A constant is a data object whose value remains unchanged throughout the execution of a Pascal program. Constants in Pascal are of two types. The first is a constant declared at the beginning of a program, using the following syntactical format:

```
const
    Name_1 = value_1;
    Name_2 = value_2;
     . . .
    Name_n = value_n;
```

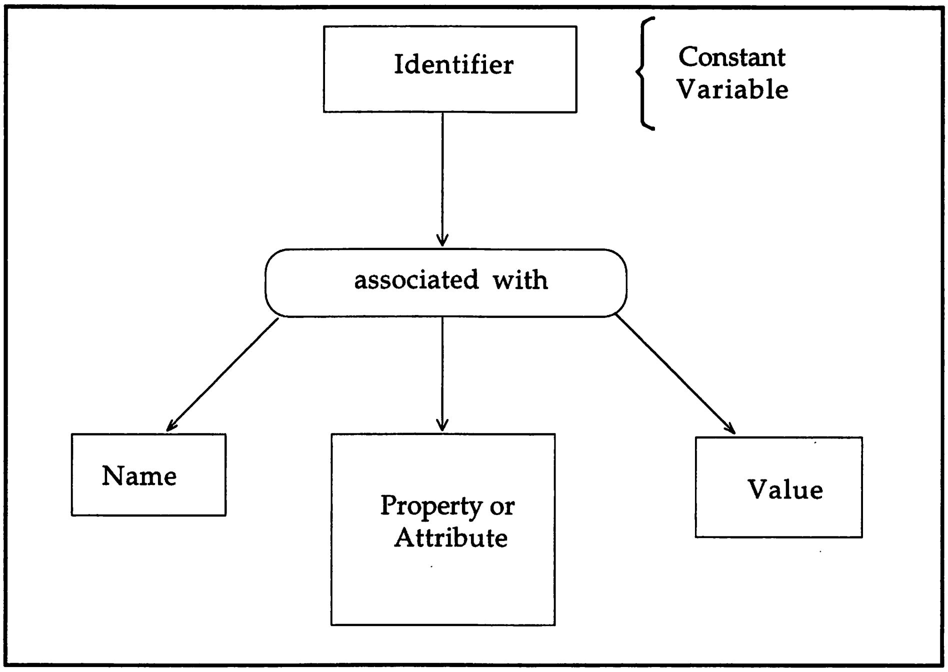


Figure 3.3 The concept of a data object.

Here a named identifier is associated with a value on the right. This value can be a number, a predeclared constant already known to Macintosh Pascal, or the name of a another constant previously declared. *It cannot be an expression.* The equal sign in the constant expression represents an equality between the name on the left and the value to its right. When the Pascal program is translated from Pascal to machine code, everywhere that the name of the constant appears, it is replaced by its equivalent value. It is a data object at the level of the Pascal language, since it has both a name and a value. The attribute associated with the constant is given by the attribute associated with the value of the constant. For example, the program `Circle` has three constants defined as follows:

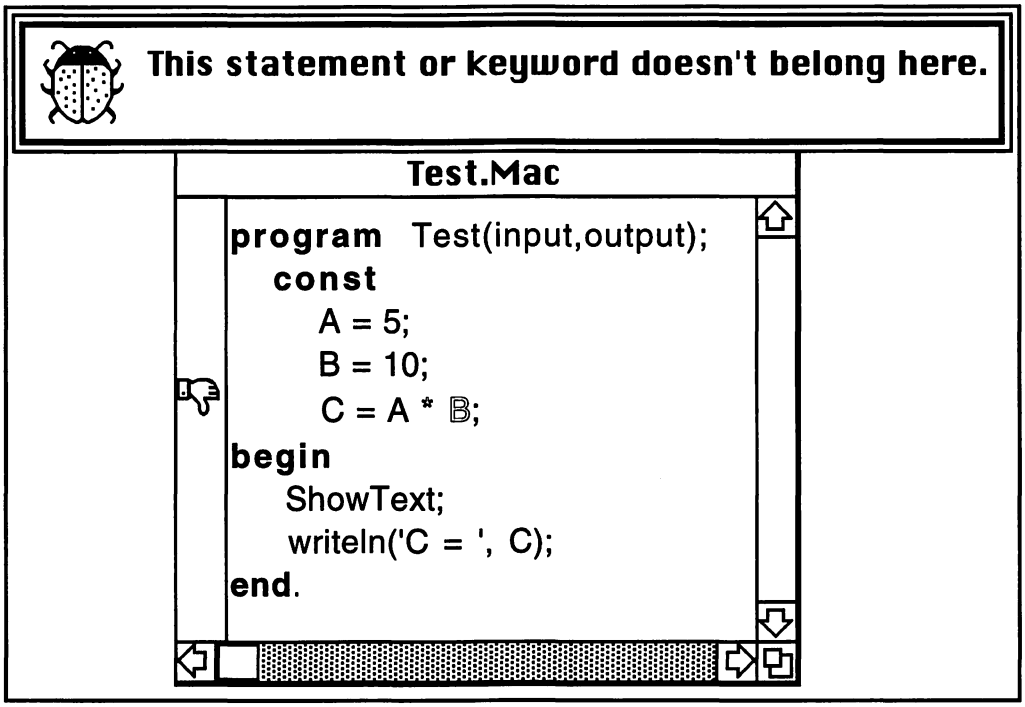
```
const
    Height = 50;
    Width = 40;
    Radius = 30;
```

Each constant is of type `integer`, since each value is written as a whole number, an integer.

Although the Macintosh Pascal syntax rules do not allow the value of a constant to be an expression, a constant can be equated with the name of another constant. Some examples showing values for constants as well as variations in writing the declarations follow:

```
const
    Max = 120.0;
    Min = - Max;
    Truth = true;
    Nontruth = false;
    Message = 'This is a string.';
    Character_Const = 'A';
```

THINK Pascal syntax rules *do* allow the value of a constant to be an expression. For example, the program `Test`, listed below, will execute in THINK Pascal, causing the value of the variable `C` to be written as 50. An attempt to execute the same program under Macintosh Pascal will cause a syntax error, as shown in Figure 3.4.



This statement or keyword doesn't belong here.

```
program  Test(input,output);
   const
      A = 5;
      B = 10;
      C = A * B;
begin
   ShowText;
   writeln('C = ', C);
end.
```

Figure 3.4 The program `Test` after attempted execution under Macintosh Pascal.

```
Program Test(input, output);
const
   A = 5;
   B = 10;
   C = A * B;
begin
   ShowText;
   writeln('C = ',C);
end.
```

In addition to multiplication, the other standard operators (+, −, /, **div**, and **mod**) can be included in the constant expressions of a THINK Pascal program.

The reserved word **const** can be repeated several times when declaring constants; for example,

```
const
    Max = 120.0;
    Min = - Max;
const
    Truth = true;
    Nontruth = false;
const
    Message = 'This is a string.';
    Character = 'A';
```

The second kind of constant found in Pascal programs is an explicit value used within an expression in an executable statement. This kind of constant is not given a name. As an example, consider the following Pascal statement from the program `Electric_Bill`:

```
Average_Consumption := Total_Consumption / 12;
```

In this statement the slash (`/`) represents division, while the characters **:=** represent the action of assigning a value to a data object. When executed, this statement will result in the value of `Total_Consumption` being divided by 12 and assigned to the object called `Average_Consumption`. The 12 is a constant with no previous declaration and has no association with an identifier. Although this constant has no name, it assumes the properties of an `integer` type due to its format (12 as compared with 12.0).

Both THINK Pascal and Macintosh Pascal have a number of built-in constants for reference. These include *pi* (3.141592653589793239), *true*, *false*, `maxint` (32,767, or $2^{15}-1$), and maxlongint (2,147,483,647, or $2^{31}-1$). These constants may be incorporated in a program without previous declaration, since they are understood by the Pascal software.

Real constants in Pascal must be represented by at least one digit preceding the decimal point and at least one digit following the decimal point.

3.3.2 Variables

A variable is a data object whose value can be changed during the execution of a program. Variables are declared after constants and types by using the following syntactical form:

```
var
    Name_1 : datatype;
    Name_2 : datatype;
    . . .
    Name_n : datatype;
```

Here the word **var** is reserved for declaring data objects to be variables. As with declaring constants, the reserved word **var** can be repeated several times when writing variable declarations. The rules for naming variables are the same as those for any identifier. In addition, however, it is wise to select variable names that are meaningful.

Cryptic variable names can make a program difficult to read, especially if it has been left to sit idle for several months. Following are some examples of variable declarations:

```
var
    Max_Value : integer;
    Interest : real;
    Debt, Balance, Income : real;
    Months, Days : integer;
```

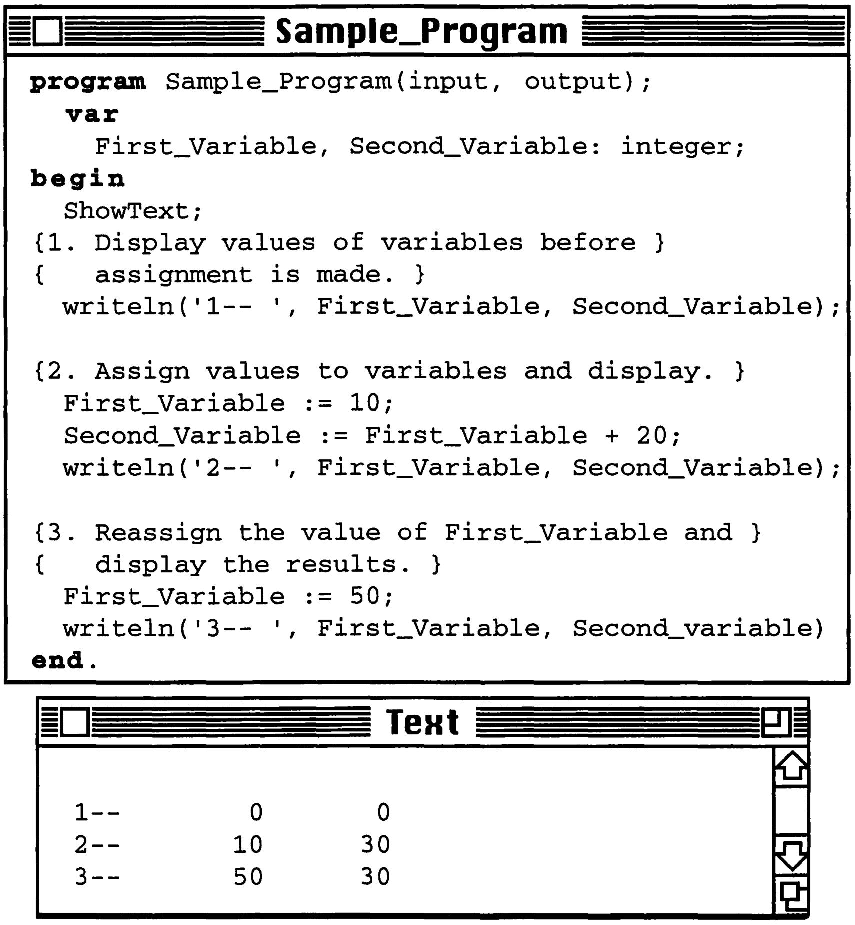
Notice that you are allowed to list several names, separated by commas, on the same line if they are of the same data type. For example, Debt, Balance, and Income are all declared to be real data types and are listed together.

A variable satisfies the properties of a data object; it has a name, an attribute given by an explicit declaration of a data type, and a value that can only be assigned during the execution of a program. Values are given to variables only through the execution of an assignment or input statement. In memory a variable can require one or more cells to store its value, depending on the attribute (data type) associated with the variable. This is different from a constant, which never requires explicit memory cells for storing its value. At this level of the machine, the name of a variable represents an address where the value of the variable is stored. When a new value for the variable is entered from the keyboard, or when a new value is assigned to the variable by an assignment statement, the value that is presently stored is lost, and the new value is assigned. In other words, the new value is written over the old value. When the value of a variable is needed for computation, it is copied from memory and used. The value presently stored for the variable remains unchanged by this process.

The program Sample_Program, shown in Figure 3.5, illustrates the use of a variable. The program declares two variables named First_Variable and Second_Variable in the declarations. The executable portion of the program consists of writeln commands used for displaying messages and the values of the variables to the Text window. The program is divided into three sections for ease of reading. In the first section, values of each of the two variables are displayed prior to assignment of a value by the programmer, as shown in the Text window in the first line. This raises an important point. All numeric variables in either THINK or Macintosh Pascal have a value of zero when execution begins.[2]

In the second section of Sample_Program, a value of 10 is assigned to First_Variable, and a value of First_Variable + 20 to Second_Variable. Line 2 in the Text window displays these values. Finally, the value of First_Variable is changed to 50 in order to illustrate that the number contained in its memory cell may be changed. The third line displayed in the Text window confirms this change. (Notice that the value of Second_Variable still reflects the first value assigned to First_Variable, since nothing has been done to change the value of Second_Variable.)

[2] It is not good programming practice to depend upon this initial value. If you want a variable to begin with a value of zero, you should initialize it to that value as a part of your program. This practice was illustrated in the program Electric_Bill.



```
program Sample_Program(input, output);
  var
    First_Variable, Second_Variable: integer;
begin
  ShowText;
{1. Display values of variables before }
{   assignment is made. }
  writeln('1-- ', First_Variable, Second_Variable);

{2. Assign values to variables and display. }
  First_Variable := 10;
  Second_Variable := First_Variable + 20;
  writeln('2-- ', First_Variable, Second_Variable);

{3. Reassign the value of First_Variable and }
{   display the results. }
  First_Variable := 50;
  writeln('3-- ', First_Variable, Second_variable)
end.
```

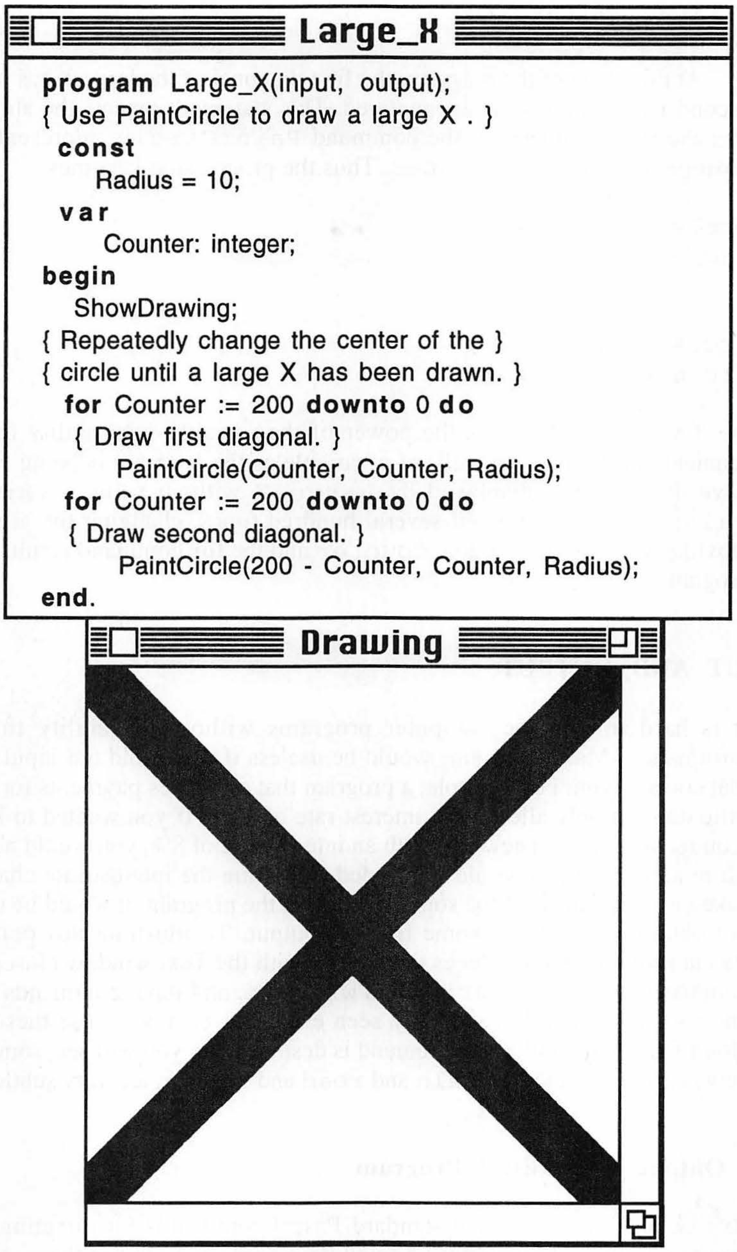
```
1--        0        0
2--       10       30
3--       50       30
```

Figure 3.5 `Sample_Program` and its output.

Now that we have introduced the concepts of constants and variables, let us illustrate their use in another program, `Large_X`. This program uses the `PaintCircle` command in a new way. By drawing many circles, each with a different center location, the command `PaintCircle` becomes a brush for drawing a large *X* in the Drawing window. Figure 3.6 shows both the program and its result. In order to keep this example simple, we employ a special control construct, the **for** statement, which we will discuss in detail later. For now you need only know that this statement allows us to change the value of a variable repeatedly during the execution of a program. The change is over a range specified in the statement. In this example we change the value of the variable `Counter` from 200 to 0. Thus we begin the process with the value of `Counter` set at 200, drawing the circle with the following arguments of the command `PaintCircle`:

```
PaintCircle(200,200,10);
```

The value of `Counter` is then changed by the **for** statement to 199, so that the arguments of the command `PaintCircle` are



```
program Large_X(input, output);
{ Use PaintCircle to draw a large X . }
 const
    Radius = 10;
 var
     Counter: integer;
begin
   ShowDrawing;
{ Repeatedly change the center of the }
{ circle until a large X has been drawn. }
  for Counter := 200 downto 0 do
   { Draw first diagonal. }
       PaintCircle(Counter, Counter, Radius);
  for Counter := 200 downto 0 do
  { Draw second diagonal. }
       PaintCircle(200 - Counter, Counter, Radius);
end.
```

Figure 3.6 The program `Large_X` and its output.

```
PaintCircle(199,199,10);
```

This process continues until the value of `Counter` becomes 0, and the command `PaintCircle` has the following arguments:

```
PaintCircle(0, 0,10);
```

At this stage of the program, the first diagonal of the large *X* has been drawn, and the second **for** statement is encountered. This statement repeats the above process, except that the first argument of the command `PaintCircle`, representing the horizontal position, is now 200 – `Counter`. Thus the progression becomes

```
PaintCircle(0, 200, 10);
PaintCircle(1, 199, 10);
. . .
PaintCircle(199, 1, 10);
PaintCircle(200, 0, 10);
```

`Large_X` illustrates the power of the variable—the ability to change the value (contents of the memory cell) of a variable as the program is being executed. We could have drawn the X displayed by `Large_X` without using a variable by typing the `PaintCircle` command several hundred times, changing the arguments each time. Obviously, using the variable `Counter` and the **for** command results in a more efficient program.

3.4 INPUT AND OUTPUT

It is hard to imagine computer programs without the ability to input and output information. Many programs would be useless if you could not input the specific values that concern you. For example, a program that computes payments for a loan would be of little use if it only allowed an interest rate of 10%. If you wanted to know the payments required to finance a new car, with an interest rate of 8%, you would need a new program. More new programs would be needed each time the interest rate changed in the future. Likewise, regardless of the sophistication of the program, it would be useless if you could not obtain its results in some form of output. To illustrate this point, try executing a Pascal program that produces text output with the Text window closed. In Pascal, output commands are given as `write` and `writeln`, and input commands are given as `read` and `readln`. You have already seen examples of how to use these commands, but a closer examination of each command is desirable. As you will see, some of the differences between `write` and `writeln` and `read` and `readln` are very subtle.

3.4.1 Output in a Pascal Program

`Write` and `writeln` are standard Pascal commands for directing the output from a program to the Text window.[3] As you have seen, you can use these statements to display

[3] With THINK Pascal you must always include the command ShowText as part of your program to display the Text window. Failure to do so will make the `write` and `writeln` commands useless, since you will be unable to see the result. This command is unnecessary with Macintosh Pascal, because the Text window is automatically displayed, unless another command is used to prevent it.

messages (prompts), values associated with data objects, or both. `Write` is an executable statement having the following form:

```
write( par_1, par_2, . . . , par_n );
```

where par_1, par_2, . . . , par_n are referred to as *parameters* and can be constants, variables, or expressions. When this statement is executed, the value of each parameter is displayed in the Text window, starting with the value of the first parameter on the left and ending with the last parameter on the right. For example, the statement

```
write( A, B, A + C );
```

begins execution by first displaying the value for variable A, followed by the value for variable B, followed by the value for the expression A + C. This statement could also be written as three `write` statements:

```
write( A );
write( B );
write( A + C );
```

A `write` statement does not terminate a display; the next `write` or `writeln` statement that is executed will continue to display data along the same line. This is a useful feature if you want a horizontal display of values.

`Writeln` is also an executable statement having the following form:

```
writeln( par_1, par_2, . . . , par_n );
```

This command differs from the `write` command in that the display line is terminated after a `writeln` is executed. Execution of the next `write` or `writeln` statement displays data on a new display line. For example, the statement

```
writeln( A, B, A + C );
```

which is equivalent to

```
write( A );
write( B );
writeln( A + C );
```

will display the values of A, B, and A + C on one line. The following three `writeln` statements will display each of their parameters on separate display lines:

```
writeln( A );
writeln( B );
writeln( A + C );
```

The unadorned statement

```
writeln;
```

can be used to terminate a display line as well as to display a blank line. For example, the following statements cause one blank line appear between each of the three values displayed in the Text window:

```
writeln(' Value of A: ', A );
writeln;
writeln(' Value of B: '; B );
writeln;
```

On the other hand, the statement

```
write;
```

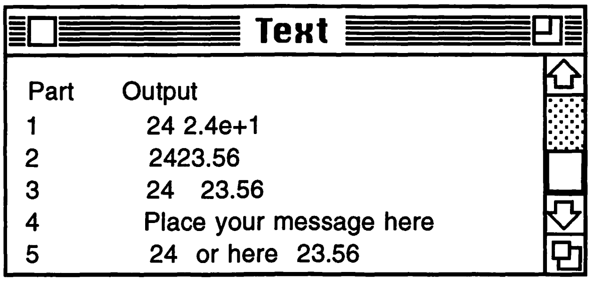
is equivalent to displaying a null character, a character having no image and no length. In brief, this statement has no effect on output. The statements `writeln()` and `write()` are syntactically incorrect in both Macintosh Pascal and THINK Pascal. Either statement will cause an error message when the program is checked.

The following program, `Display_Text`, illustrates the use of these two commands with emphasis on their differences. The program is divided into segments in order to simplify the discussion. (Segments are separated by comments.)

```
program Display_Text(input, output);
{ Purpose:  Introduction to the write and writeln statements. }
   var
{ Declare two "typical" numeric variables. }
      First_Variable : integer;
      Second_Variable : real;
begin
   ShowText;
{ Provide a heading for the display. }
   writeln('Part   Output');
{ Assign values to the variables. }
   First_Variable := 24;
   Second_Variable:= 23.56;
{ 1.  Display the values of the two variables. }
   write('1', First_Variable);
   writeln(Second_Variable);
{ 2.  Clean up the format of the second variable. }
   write('2', First_Variable);
   writeln(Second_Variable : 5 : 2);
{ 3.  Separate the variables in the display. }
   write('3', First_Variable, '  ');
   writeln(Second_Variable : 5 : 2);
{ 4.  Displaying a message in the output. }
   writeln('4', '       ', 'Place your message here');
{ 5.  Displaying text and variable values together. }
   write('5', First_Variable, '  or here  ');
   write(Second_Variable : 5 : 2);
   writeln
end.
```

Consider each of these segments separately. First, the program heading consists of the identifier, `Display_Text`, followed by the declaration of two numeric variables: an `integer` variable called `First_Variable` and a `real` variable called `Second_Variable`. The standard ShowText command insures that the THINK Pascal user can see the results.

Second, there is a segment that produces a heading for the display. This consists of the command `writeln('Part      Output')`. As you can see in Figure 3.7, the material within the single quotes is printed exactly as it appears in the program. This includes the blank spaces between the words `Part` and `Output`, which are included to align the headings with the remainder of the display. Usually the programmer uses trial and error to determine the exact number of blank spaces to include.



```
Part    Output
1         24 2.4e+1
2         2423.56
3         24   23.56
4         Place your message here
5          24  or here  23.56
```

Figure 3.7 The output from `Display_Text`.

Third, there is a section in which the variables are assigned their initial values. The assignment is achieved with the Pascal symbol :=, which we read as "becomes." For example, "`First_Variable` becomes 24."

Fourth, the assigned values of the two variables are printed. To help you follow the discussion, the number 1 is included in the display, preceding the display of the values 24 and 2.4e+1. Notice the use of the `write` command rather than the `writeln` command, which we used previously. This allows the values of both variables to be printed on the same display line.

There are two problems with the display from Section 1 of the program. First, the value of `Second_Variable` is displayed in the form of scientific notation (which is not comfortable for many people and also loses detail, as 23.56 is reported as 2.4e+1). Second, the two values are displayed as one stream of digits with no space between them, making the display difficult to read.

Fifth, in Section 2 of the program, the scientific notation is removed by means of the format `: 5 : 2` in the command `writeln(Second_Variable : 5 : 2)`. The first of these format numbers (5) gives the minimum size (minimum number of characters including digits, sign, and decimal point) of the field to be occupied by `Second_Variable`. Since the value assigned to `Second_Variable` is 23.56, the size of the field must be at least five (four digits and a decimal point). If you specify too few spaces, the Pascal system will produce the necessary additional space for displaying the new value. Thus the command `writeln(Second_Variable : 3 : 2)`, would result in the same display. The second format number (2) indicates the number of digits to be displayed to the right of the decimal point. The output from this segment of `Display_Text` is labeled 2 in Figure 3.7.

Sixth, we solve the problem of the values of the variables, 24 and 23.56, being displayed in one stream (that is, without any separation). This is done by including a space (a blank enclosed within single quotes) in the command

```
write('3', First_Variable,' ');
write(Second_Variable : 5: 2);
```

The space following `First_Variable` will cause the two numbers to be separated, as shown in Line 3 of the output display. The same effect can be achieved with the single statement

```
write('3', First_Variable, Second_Variable : 6: 2);
```

where the overall length of the field for the `real` variable, `Second_Variable`, is enlarged to allow a preceding blank space.

Seventh, as shown in Line 4 of the display, a text message may be included in the output. This may be a straightforward message, as shown in Line 4, or it may be integrated with the variable output, as shown in Line 5. The entire output of `Display_ Text` is shown in Figure 3.7.

Remember the following guidelines when using the `write` and `writeln` commands. Use the `write` command if you want to continue displaying output on the current line appearing in the Text window. Use the `writeln` command if you want to display output on a new line in the Text window. To display the value of a `real`-type variable, indicate the overall size of the field and the number of places desired after the decimal point. Any messages to be included in the display should be contained in single quotes. If more than one variable is included in a `write` or `writeln` statement, commas are required to separate the list of items. Use trial and error to obtain a display with a pleasing appearance.

3.4.2 Input in a Pascal Program

`Read` and `readln` commands are standard Pascal input commands for entering data into a Pascal program. When executed, either command will cause the Macintosh computer to pause and accept input from the keyboard.

The `read` and `readln` commands are executable statements having the following forms:

```
read( variable_1, variable_2, . . . , variable_n );
readln( variable_1, variable_2, . . . , variable_n );
```

where `variable_1, variable_2, . . . , variable_n` are the names of previously declared variables. When executed, each statement will halt execution until the values of all variables have been typed, and the Return key has been pressed.

The relationship between `read` and `readln` is similar to the relationship between `write` and `writeln`. After execution of a `read` statement, execution of a `read, readln, write,` or `writeln` command will continue to display data on the same line. A `readln` command terminates the display line after all variables have been entered. A new line will be displayed when the next input or output command is executed. Whereas the statement `read;` has no effect on either input or output, the statement `readln;` can be used to terminate a display line after input has been entered. For example, the statement

```
readln( A, B, C );
```

can be replaced by either of the following sets of commands

```
read( A, B, C );
readln;
```

or

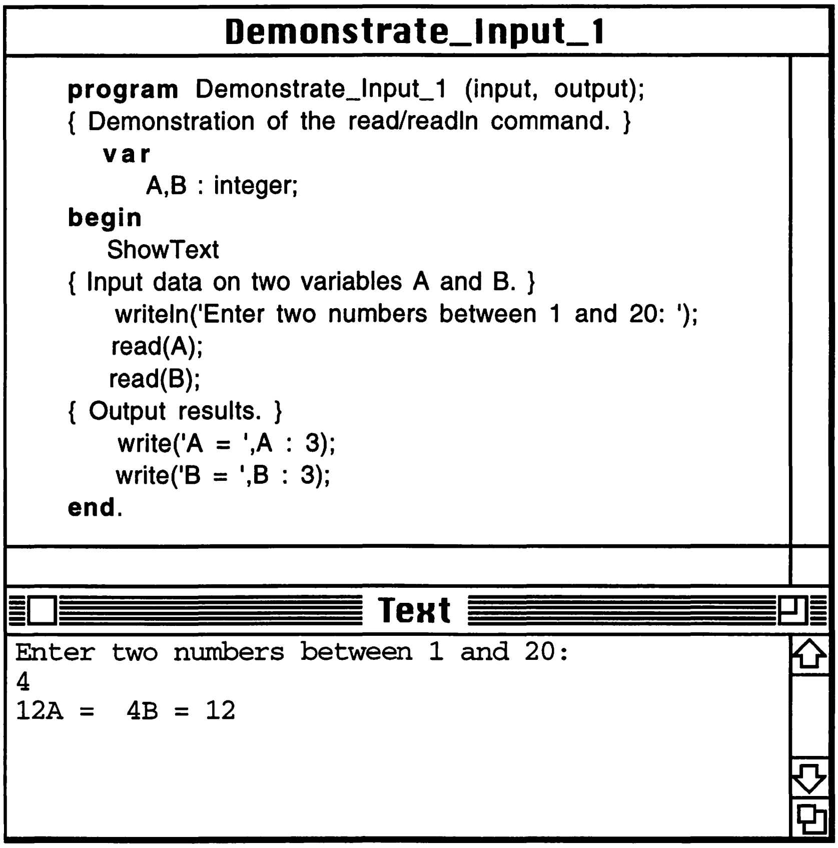
```
read( A, B, C );
writeln;
```

The statements `read()` and `readln()` are not understood by either THINK Pascal or Macintosh Pascal and will cause syntax errors. Figure 3.8 shows a program that demonstrates the difference between `read` and `readln`. Titled `Demonstrate_Input_1`, this program accepts input for two variables and then displays their values to the Text window. In this example, the Return key must be pressed after each value is entered. Notice that in the top portion of Figure 3.8, the data displayed in the Text window is somewhat confusing because the input (4 and 12) and output (`A = 4` and `B = 12`) were not well controlled by the programmer. Figure 3.9 shows the same output after one `read` command has been replaced by a `readln` command, and one `write` command has been replaced by a `writeln` command. As the figure shows, the `readln` and `writeln` commands terminate the display lines, eliminating the confusion. You should be aware that the difference between `read` and `readln` shown here is not seen if `Demonstrate_Input_1` is executed as a THINK Pascal program. (Try `Demonstrate_Input_1` under both THINK and Macintosh Pascal to observe this difference.) Thus, with THINK Pascal the programmer has less to control with a `read` statement.

As an additional example, let us develop a program for computing the gas mileage and cost of a trip. For input we have the distance traveled, the total number of gallons used, and the cost per gallon of gas. For output we will report mileage in miles per gallon, the total cost, and the cost per mile traveled. The initial steps in our algorithm follow:

1. Enter the trip data for distance traveled, gallons of gas consumed, and cost per gallon.
2. Compute the miles per gallon, total cost for the trip, and cost per mile.
3. Output the three values computed in Step 2.

Does our algorithm satisfy all of the characteristics required of a good algorithm? First, it has a finite number of steps, none of which needs to be repeated. Second, it has input and output. Third, it is sufficiently simple to be traced by hand. What it lacks is the property of definiteness. The step for computing the values needed as output is not clear, nor is it clear in Step 3 what values are to be displayed. The following is a refinement of our initial algorithm, including comments.


Demonstrate_Input_1

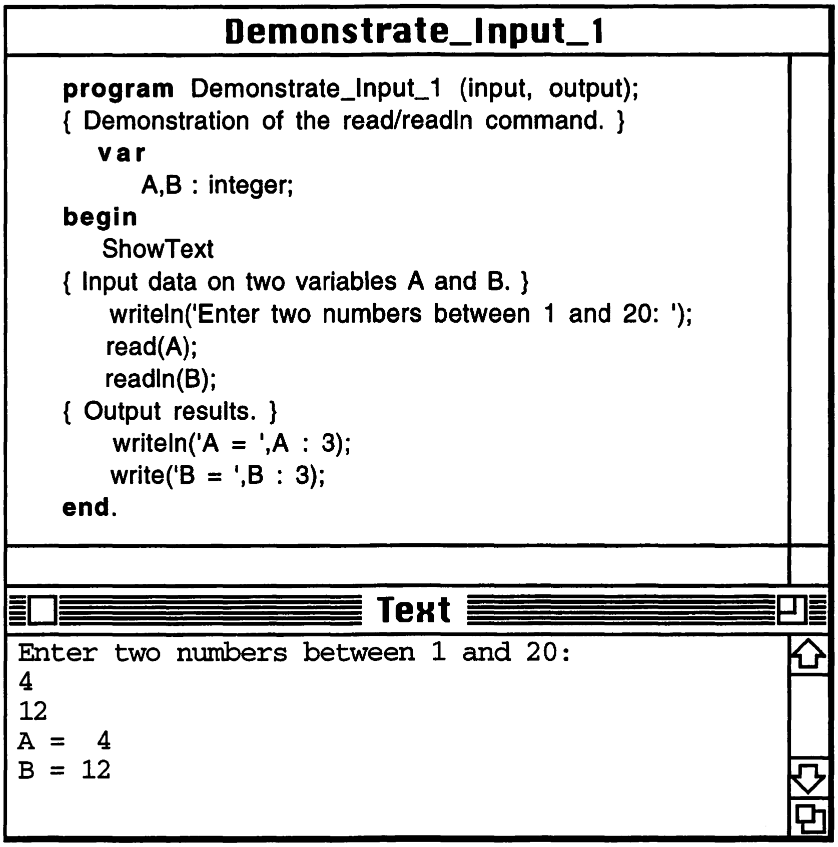
```
program Demonstrate_Input_1 (input, output);
{ Demonstration of the read/readln command. }
  var
     A,B : integer;
begin
   ShowText
{ Input data on two variables A and B. }
    writeln('Enter two numbers between 1 and 20: ');
    read(A);
    read(B);
{ Output results. }
    write('A = ',A : 3);
    write('B = ',B : 3);
end.
```

Text

```
Enter two numbers between 1 and 20:
4
12A =  4B = 12
```

Figure 3.8 A comparison of `read` and `readln`, with output produced by execution of a Macintosh Pascal Program.

```
Algorithm Trip_Analysis;
{ Prompt user to enter trip data. }
   write 'Enter the number of miles you drove : ';
   read Distance_Traveled;
   write 'Enter the gallons of gasoline used: ';
   read Gallons_Used;
   write 'Enter the price per gallon for gasoline: ';
   read Price_per_Gallon;
{ Compute the miles per gallon, total cost, and cost
   per mile.}
   Mileage <-- Distance_Traveled/ Gallons_Used;
   Total_Cost <-- Price_per_Gallon * Gallons_Used;
   Cost_per_Mile <-- Total_Cost / Price_per_Gallon;
{ Output the computed values. }
   write Mileage;
   write Total_Cost;
   write Cost_per_Mile;
{End of algorithm.}
```



```
program Demonstrate_Input_1 (input, output);
{ Demonstration of the read/readln command. }
  var
     A,B : integer;
begin
   ShowText
{ Input data on two variables A and B. }
    writeln('Enter two numbers between 1 and 20: ');
    read(A);
    readln(B);
{ Output results. }
    writeln('A = ',A : 3);
    write('B = ',B : 3);
end.
```

```
Enter two numbers between 1 and 20:
4
12
A =  4
B = 12
```

Figure 3.9 An improved version of Demonstrate_Input_1.

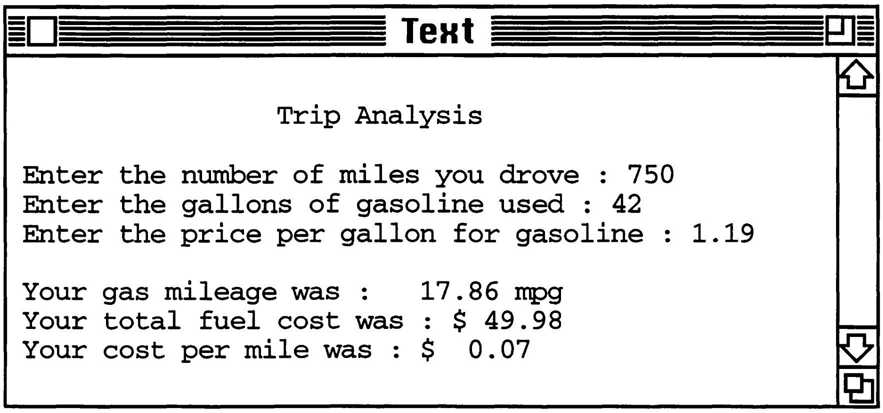
Notice that we have used the commands read and write to indicate input and output, respectively. The distinction between read and readln, and between write and writeln is unimportant in the algorithm. We are more concerned with the steps in solving the problem than with the details of displaying it on the screen. We will consider the extra formatting needed to prompt the user and display data as we write and test the program.

The following is the Pascal program for the algorithm Trip_Analysis. It illustrates the use of both input and output commands.

```
program Trip_Analysis(input, output);
{ Purpose:  Analysis of the cost of a trip and consumption of }
{           gasoline. }
   var
      Cost_per_Mile, Distance_Traveled, Gallons_Used : real;
      Mileage, Price_per_Gallon, Total_Cost: real;
begin
   ShowText;
{ Display output title. }
```

```
   writeln;
   writeln('                Trip Analysis');
   writeln;
{ Prompt user to enter trip data. }
   write('Enter the number of miles you drove : ');
   readln(Distance_Traveled);
   write('Enter the gallons of gasoline used : ');
   readln(Gallons_Used);
   write('Enter the price per gallon for gasoline : ');
   readln(Price_per_Gallon);
   writeln;
{ Compute the mileage, total cost, and cost per mile. }
   Mileage := Distance_Traveled  /  Gallons_Used;
   Total_Cost := Price_per_Gallon  *  Gallons_Used;
   Cost_per_Mile := Total_Cost / Distance_Traveled;
{ Output results to the Text window. }
   write('Your gas mileage was :   ');
   writeln(Mileage : 4 : 2, ' mpg');
   write('Your total fuel cost was : $');
   writeln(Total_Cost : 6 : 2);
   write('Your cost per mile was : $');
   writeln(Cost_per_Mile : 6 : 2)
end.
```

With the exception of the three lines of calculations and the comments, this program is a series of `read`, `readln`, `write`, and `writeln` commands. The output from the program is shown in Figure 3.10.



```
                 Trip Analysis

Enter the number of miles you drove : 750
Enter the gallons of gasoline used : 42
Enter the price per gallon for gasoline : 1.19

Your gas mileage was :   17.86 mpg
Your total fuel cost was : $ 49.98
Your cost per mile was : $  0.07
```

Figure 3.10 The output from `Trip_Analysis`.

After the heading `Trip_Analysis`, three lines are displayed, each of which consists of a prompt for input and the user's response to the prompt. The first prompt, `Enter the number of miles you drove:`, results from the `write` command,

```
write('Enter the number of miles you drove : ');
```

This is followed by the `readln` command, `readln(Distance)`. This command causes the execution of the program to pause until the user enters a response. By entering 750 and pressing the Return key, the user causes the variable `Distance_Traveled` to be assigned the `real` value 750.0. This approach is repeated for entering values of `Gallons_Used` and `Price_per_Gallon`.

3.5 SIMPLE DATA TYPES IN MACINTOSH AND THINK PASCAL

Macintosh and THINK Pascal support a variety of data types. In general, these can be classified as simple (sometimes called *scalar*) and structured. In this chapter we concentrate on the former kind; the latter are covered in later chapters. Figure 3.11 shows a breakdown of simple data types available under Macintosh and THINK Pascal.

I. Real Data Types
  A. Real
  B. Double
  C. Extended
  D. Computational
II. Ordinal Data Types
  A. Standard
    1. Integer
    2. Longint (long integer)
    3. Char (character)
    4. Boolean
  B. Nonstandard
    1. Enumerated
    2. Subrange

Figure 3.11 Classification of simple data types in Macintosh and THINK Pascal.

The broadest distinction is between `real` and `ordinal` data types. Real types involve real numbers (as opposed to ordinal numbers). These include the types `real`, `double`, `extended`, and `computational`. `Ordinal` data types involve ordinal numbers. The standard `ordinal` types are `integer`, `longint` (long `integer`), `char` (character), and `Boolean`. The nonstandard `ordinal` types are the `enumerated` type and the `subrange` type. We will discuss each of these types in the following sections.

3.5.1 Real Data Types

In Pascal, a number that includes a decimal point is referred to as a *`real`* number. Examples of `real` numbers are 0.0093, 1.29, 43.7 and 69500.00. This last number, 69500.00, can also be written as 6.95 * 10^4 (this is called *scientific notation*). Since Pascal cannot handle the superscript in this notation, the number is represented in a floating-point notation; that is, the number 69500.00 is written as 6.95E+4 or 6.95e+4. In this notation the letter E or e represents a factor of 10, and the +4 represents the exponent. In Pascal, real data types are those that accept floating-point numbers.

The distinction among the real data types real, double, and extended is in the number of significant digits retained and in the range of values that the numbers can represent. The approximate limits are given in Figure 3.12.

Real Data Type	Significant Digits	Range of Number		
Real	7–8	$1.5 * 10^{-45}$	to	$3.4 * 10^{38}$
Double	15–16	$5.0 * 10^{-324}$	to	$1.7 * 10^{308}$
Extended	19–20	$1.9 * 10^{-4951}$	to	$1.1 * 10^{4932}$
Computational	18-19	$9.2 * 10^{18}$	to	$9.2 * 10^{18}$

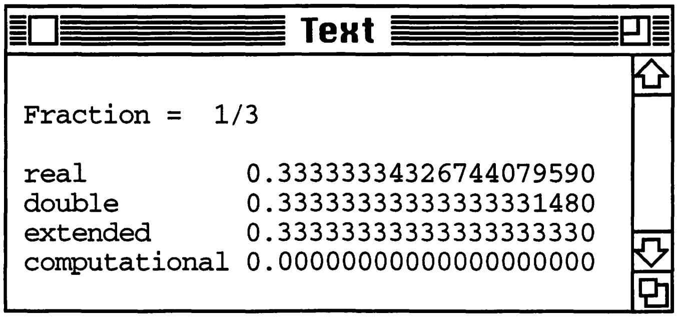
Figure 3.12 A comparison of real data types in Macintosh and THINK Pascal.

These numbers can be negative as well as positive. The comp (computational) type differs from the other types in that the number involved must be an integer. (There can be no decimal fraction and no exponent.) This type is convenient for accounting applications where exact representation of numbers is desired. When you use this form, you must add the decimal point to the computed result. For example, in a program designed to compute interest earned, all of the computations would be based on measuring the results in pennies; conversion to dollars and cents would be the responsibility of the programmer. In both Macintosh and THINK Pascal, each of the real types is converted to extended mode when real arithmetic operations are performed.

The program Real_Numbers illustrates the differences among these types. This program takes advantage of the endless string of digits produced when certain fractions are expressed as decimal numbers. Since a computer can only approximate this endless string, the results displayed by the program show the accuracy of the approximation with each of the different real data types. The listing for the program Real_Numbers follows.

```
program Real_Numbers(input, output);
{ Purpose:  Demonstrate variations on real numbers. }
   const
      Numerator = 1;
      Denominator = 3;
   var
      First: real;
      Second: double;
      Third: extended;
      Fourth: computational;
begin
   ShowText;
{ Write 1/3 as four different types of real numbers. }
   First := Numerator / Denominator;
   Second := Numerator / Denominator;
   Third := Numerator / Denominator;
   Fourth := Numerator / Denominator;
   writeln;
   writeln('Fraction = ', Numerator : 2, '/', Denominator : 1);
   writeln;
   writeln('real          ', First : 22 : 20);
   writeln('double        ', Second : 22 : 20);
   writeln('extended      ', Third : 22 : 20);
   writeln('computational ', Fourth : 22 : 20);
end.
```

Four variables are declared: one `real`, one `double`, one `extended`, and one `computational`. Each of these variables is assigned the value of 1/3 within the body of the program, with the result being displayed. The results of the execution of this program are given in Figure 3.13.



```
Fraction =  1/3

real          0.33333334326744079590
double        0.33333333333333331480
extended      0.33333333333333333330
computational 0.00000000000000000000
```

Figure 3.13 Output from `Real_Numbers` with the fraction 1/3.

As you can see, the different data types produce distinctly different results. The `real` type produces a string of seven 3s. With the `double` type, the length of the string is 16. With the `extended` type, the string contains nineteen 3s. The `computational` type produces no 3s, since this type can only represent `integer` numbers.

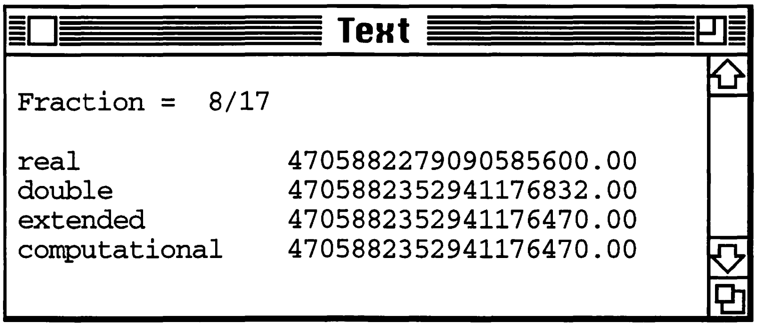
This can be more clearly demonstrated by using other fractions as values and by making the ratio greater than 1 in order to allow the `computational` type to be included in the comparison. We achieve the latter step by scaling each of the numerators as follows:

```
First  := (Numerator * Scalefactor) / Denominator;
Second := (Numerator * Scalefactor) / Denominator;
Third  := (Numerator * Scalefactor) / Denominator;
Fourth := (Numerator * Scalefactor) / Denominator;
```

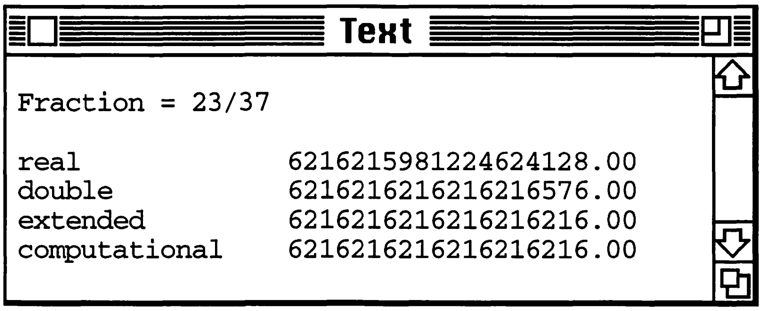
where `Scalefactor` is a constant assigned a value of 10000000000000000000 or 1.0 * 10^{19}. Figure 3.14 shows the output after these changes for two additional fractions, 23/37 and 8/17. In both cases the differences between `real` and `double` and between `double` and `extended` are readily apparent.

When reading a value for a variable that is of type `real`, `double`, `extended`, or `computational`, characters are scanned from left to right. Both the `read` and `readln` statements will skip all blanks and end-of-lines (returns) that precede the digits of a `real` number. On reaching a proper digit, the `read` statement will continue to read the sequence of digits and other characters that form a proper signed `real` number. The first nondigit encountered terminates the entry of a `real` value. A nondigit can be a blank, a return, or some other character such as a letter of the alphabet. Consider the following example where `A` is a `real` type:

```
readln( A );
```



```
Fraction =  8/17

real               4705882279090585600.00
double             4705882352941176832.00
extended           4705882352941176470.00
computational      4705882352941176470.00
```



```
Fraction = 23/37

real               6216215981224624128.00
double             6216216216216216576.00
extended           6216216216216216216.00
computational      6216216216216216216.00
```

Figure 3.14 Output from `Real_Numbers` (modified) with the fractions 8/17 and 23/37.

If you type the characters:

```
-12.345This is a real number
```

the variable `A` will become −12.345. Input terminates on seeing the character T, since this is a nondigit in a `real` number. In addition, because this is a `readln` command instead of a `read` statement, the input line is terminated. Any additional input following this command is assumed to begin on a new line.

3.5.2 `Ordinal` Data Types: Standard

The term *ordinal* implies a specified position in a numbered series or order. Standard `ordinal` data types include `integer`, `longint`, `char`, and `Boolean`. `Integer` and `longint` data types have values within a specified set of `integers`. For Macintosh and THINK Pascal, these sets have the limits given in Figure 3.15. In Macintosh Pascal, integer values are automatically converted to `longint` values when integer arithmetic operations are performed.

Data Type	Largest Allowed	Smallest Allowed
Integer	32767	-32767
Longint	2147483647	-2147483647

Figure 3.15 A comparison of the limits of the two `integer` data types.

To understand why `longint` is important, consider the program `Second_Counter` for converting the time of day as input into seconds. The program computes and reports the number of seconds that have elapsed between midnight and the time of day specified on input.

```
program Second_Counter(input, output);
{ Purpose:  Program for converting hours and minutes to seconds }
{           as an integer value. }
   const
     Length_of_Hour = 3600;
     Length_of_Minute = 60;
   var
     Hours, Minutes, Seconds, Elapsed : integer;
begin
   ShowText;
{ Enter a time--no traps, so use care. }
   writeln('Enter the exact time as prompted--');
   writeln('use international time ');
   writeln;
   write('Hour ?');
   readln(Hours);
   write('Minute ?');
   readln(Minutes);
   write('Second ?');
   readln(Seconds);
{ Compute seconds elapsed since midnight. }
   Elapsed := Hours * Length_of_Hour + Minutes * Length_of_Minute
            + Seconds;
   writeln;
   writeln('The number of seconds which have ');
   writeln('elapsed since midnight is ', Elapsed : 2, '.');
end.
```

The program converts hours and minutes to seconds and adds the results. An interesting feature is that the program will execute only if the time of day is 09:06.07 A.M. or earlier. Any later time will cause the program to fail due to overflow error.[4] The program is unable to continue execution because the value of the `integer` variable

[4] The THINK Pascal version of the program will fail and report an overflow error only if you have activated the **overflow checking** option for the file `Second_Counter` in your project window. If this option is not active, a time of 9:06:08 or later will result in a negative number. Although with a little mathematical manipulation you could produce a correct result using this negative number, it is hardly an acceptable result.

called Elapsed exceeds the maximum size of an integer variable (32,767). Changing the statement

```
var
    Hours, Minutes, Seconds, Elapsed : integer;
```

to

```
var
    Hours, Minutes, Seconds, Elapsed : longint;
```

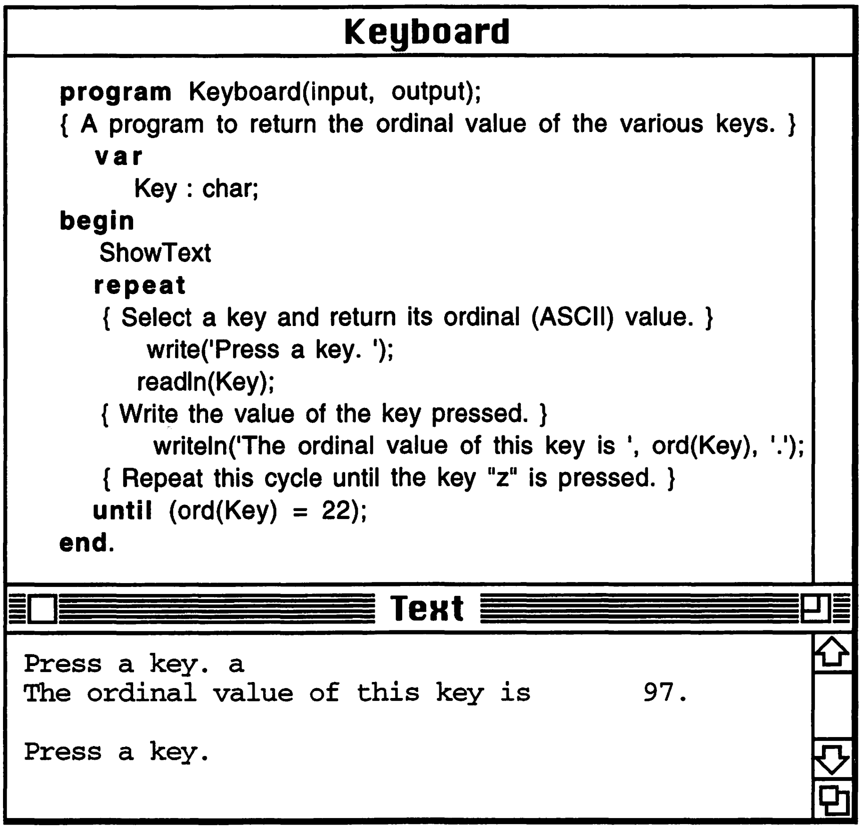
allows the program to execute with larger integer numbers and to operate with any legitimate time of day.

When reading a value for a variable that is of type integer or longint, characters are scanned from left to right. Both the read or readln statements will skip all blanks and end-of-lines (returns) that precede the digits of an integer number. On reaching a proper digit, the read statement will continue to read the sequence of digits and other characters that form a proper signed integer number. The first nondigit encountered terminates the entry of an integer value.

The ordinal data type, char, has a set of integral values that relate to the character set of the Macintosh. For example, the character *A* has an integer value of 65 (decimal). This number is also called its *ASCII value* and can be produced with the function ord(X). The function ord returns the position (ordinal number) for argument X, which must be an ordinal type such as integer, longint, char, or Boolean. A complete list of ASCII characters is given in Appendix C. For example, the program Keyboard, shown in Figure 3.16, makes use of a char-type variable. This program prompts the user to type a character from the keyboard. The computer then displays the ordinal value of the character. For the present, ignore the new commands **repeat** and **until**; they are explained in Chapter 5. The program continues to execute until the lowercase *z* key is pressed. The heart of the program is the function ord(Key), where Key is a variable declared to be of the ordinal type char.

An identifier having the data type Boolean has an ordinal value of *false* or *true*. The values *false* and *true* have ordinality, just like any other ordinal data type. Specifically, the ordinal value of *false* is 0 and the ordinal value of *true* is 1. What makes this data type unique is the limited range of its values. For example, consider the program Truth_Table, where Boolean variables are used to display a truth table. The result is a table similar to the one in Figure 3.17.

This table and the program that produces it show how Boolean variables are used with the Boolean operations **and** and **or.** The operation One **and** Two results in a value that is *true* if both values of variables One and Two are *true*. If one or both are *false*, the result of One **and** Two is *false*. The operation One **or** Two is *true* if one or both of the values of variables One and Two are *true*. If both values are *false*, the result of One **or** Two is *false*.



```
program Keyboard(input, output);
{ A program to return the ordinal value of the various keys. }
   var
      Key : char;
begin
   ShowText
   repeat
    { Select a key and return its ordinal (ASCII) value. }
        write('Press a key. ');
       readln(Key);
    { Write the value of the key pressed. }
        writeln('The ordinal value of this key is ', ord(Key), '.');
    { Repeat this cycle until the key "z" is pressed. }
   until (ord(Key) = 22);
end.
```

```
Press a key. a
The ordinal value of this key is       97.

Press a key.
```

Figure 3.16 The program `Keyboard` and its output.

Line	Variable		Operation	
	`One`	`Two`	`or`	`and`
1	*True*	*True*	*True*	*True*
2	*Tru*^	*False*	*True*	*False*
3	*False*	*True*	*True*	*False*
4	*False*	*False*	*False*	*False*

Figure 3.17 A truth table based on `Boolean` variables.

```
program Truth_Table(input, output);
{ Purpose:  demonstration of a  Boolean data type. }
   var
   { Declare two  Boolean variables. }
      One, Two :  Boolean;
begin
{ Display table headings. }
   writeln('Line     Variable      Operation');
   writeln('       One    Two      "or"    "and"');
{ Assign values for line 1. }
   One := true;
   Two := true;
{ Display table line 1 .}
   writeln('1 ', One: 8, Two: 8, One or Two: 8, One and Two: 8);
{ Assign values for line 2. }
   One := true;
   Two := false;
{ Display table line 2 .}
   writeln('2 ', One: 8, Two: 8, One or Two: 8, One and Two: 8);
{ Assign values for line 3 . }
   One := false;
   Two := true;
{ Display table line 3 .}
   writeln('3 ', One: 8, Two: 8, One or Two: 8, One and Two: 8);
{ Assign values for line 4 . }
   One := false;
   Two := false;
{ Display table line 4 . }
   writeln('4 ', One: 8, Two: 8, One or Two: 8, One and Two: 8);
end.
```

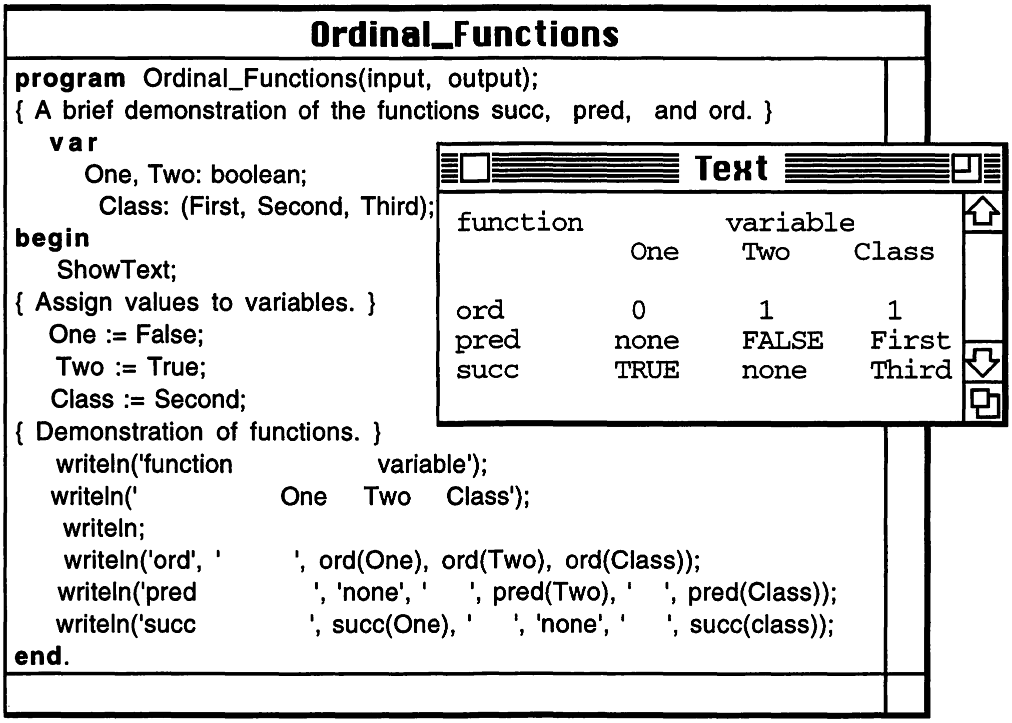
There are three important Macintosh functions that operate with any `ordinal` data type:

`ord(X)`	: This function takes as its argument an `ordinal` type represented by `X` and returns the ordinal position of this argument.
`pred(X)`	: This function takes as its argument an `ordinal` type represented by `X` and returns the ordinal predecessor of the argument.
`succ(X)`	: This function takes as its argument an `ordinal` type represented by `X` and returns the ordinal successor of the argument.

The first value in any ordinal set has no predecessor. Using the function `pred` under this circumstance will cause an error at the time of execution. Thus the expression `pred(`*`true`*`)` with a `Boolean` variable returns a value of 0 (the ordinality of the value *`false`*), while the expression `pred(`*`false`*`)` causes an error since *`false`* is the first value in this ordinal set `[`*`false`*`(0),` *`true`*`(1)]`. Likewise, the last value in this same ordinal set has no successor. The program `Ordinal_Functions`, shown in Figure 3.18, illustrates how to use these standard functions.

Three variables are declared in the program; two `Boolean` variables called `One` and `Two` and an `enumerated` variable called `Class`. (`Enumerated` variables are explained in Section 2.5.3.) These variables are assigned values as follows:

```
One := false;
Two := true;
Class := Second;
```



Ordinal_Functions

```
program Ordinal_Functions(input, output);
{ A brief demonstration of the functions succ, pred, and ord. }
   var
      One, Two: boolean;
       Class: (First, Second, Third);
begin
   ShowText;
{ Assign values to variables. }
   One := False;
    Two := True;
   Class := Second;
{ Demonstration of functions. }
    writeln('function              variable');
   writeln('              One    Two    Class');
    writeln;
    writeln('ord', '        ', ord(One), ord(Two), ord(Class));
    writeln('pred          ', 'none', '     ', pred(Two), '    ', pred(Class));
    writeln('succ          ', succ(One), '    ', 'none', '    ', succ(class));
end.
```

Text

```
function            variable
              One     Two     Class

ord           0        1        1
pred          none     FALSE    First
succ          TRUE     none     Third
```

Figure 3.18 The program `Ordinal_Functions` and its output.

The three ordinal functions are then applied to each of the variables, and the results are printed in a table. Notice that the cases that would cause a failure of the program are omitted, with the word *none* substituted in the table. For example, `pred(One)` is not included. Since the variable `One` has a value of *`false`* (with an ordinality of 0), it has no predecessor. Likewise, `succ(Two)` is omitted, since `Two` has a value of *`true`*, which has no successor. The variable `Class` produces a value for all three functions, but only because we assigned it a value from the middle of the range `(Second)`.

3.5.3 `Ordinal` Data Types: Nonstandard

In addition to these standard `ordinal` data types, there are two nonstandard types: the `enumerated` data type and the `subrange` data type. The term *enumerated* means to count off, name one by one, or list as a group. The following declaration shows some examples of the `enumerated` data type:

```
var
   Month  : ( Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,,Sep, Oct,
              Nov, Dec );
   Days_of_Week : ( Sunday, Monday, Tuesday, Wednesday, Thursday,
                    Friday, Saturday );
   First_Names : ( Rose, Mary, John, Paul, Sue, Fred, Bill );
   Alphabet : ( A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P,
                Q, R, S, T, U, V, W, X, Y, Z );
```

In this example we have declared the variables Month, Days_of_Week, First_Names, and Alphabet to be of type enumerated. These declarations indicate the possible values that each of the variables can be assigned. Through the execution of a read, readln, or assignment statement, a value can be assigned to any of our enumerated variables. The items in each list, such as (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec), have ordinal values related to their position in the enumerated list. For example, ord(Jan) is equal to 0 since Jan is first in the declaration list, while ord(Oct) is equal to 9. The other functions of the ordinal data type, pred and succ, also apply. For example, pred(Mar) is equal to Feb, and succ(Mar) is equal to Apr.

The following program, called Date, was designed around an enumerated type representing the month of the year.

```
program Date(input,output);
   var
      Month : (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,
               Nov, Dec);
      Day, Year : integer;
      Slash : char;
begin
   ShowText;
{ Prompt the user for todays date. }
   writeln(' Enter todays date using the following ');
   writeln(' format:  month/day/year,  where month');
   writeln(' is written as three characters,  day as');
   writeln(' a whole number,  and year as the last two');
   write(' digits of the current year: ');
   readln(Month, Slash, Day, Slash, Year);
   writeln;
{ Display today's date in the following format: }
{ Month - Day - 19 _ _ . }
   write('Todays Date : ');
   writeln(ord(Month) + 1 : 2, ' - ', Day : 2, ' - ', '19', Year :
           2);
end.
```

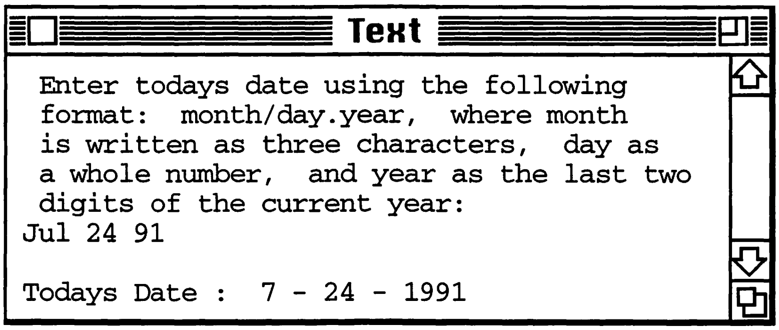
In this program we enter a date given in the form month/day/year, where month is represented by three characters, day is a two-digit number, and year is the last two digits for any year in the current century. Notice that the input statement

```
readln( Month, Slash, Day, Slash, Year );
```

reads a single character after it has read the enumerated value for Month and again reads a single character after it has read a value for Day. This simply provides a way to dispose of the slash on input. The program then displays the same date in the form "Month - Day - 19 _ _ " using the following output statement:

```
writeln(ord(Month) + 1 : 2, ' - ', Day : 2, ' - ', '19', Year :
        2 );
```

where Month is now represented by 1 plus the ordinal value of the variable Month. Why add 1 to this ordinal value? Remember that the ordinals of the enumerated values Jan through Dec are 0 through 11, respectively. The addition is necessary to correct for the proper numeric value of Month in relation to presenting the date. Figure 3.19 shows the output from Date.



```
 Enter todays date using the following
 format:  month/day.year,  where month
 is written as three characters,  day as
 a whole number,  and year as the last two
 digits of the current year:
Jul 24 91

Todays Date :  7 - 24 - 1991
```

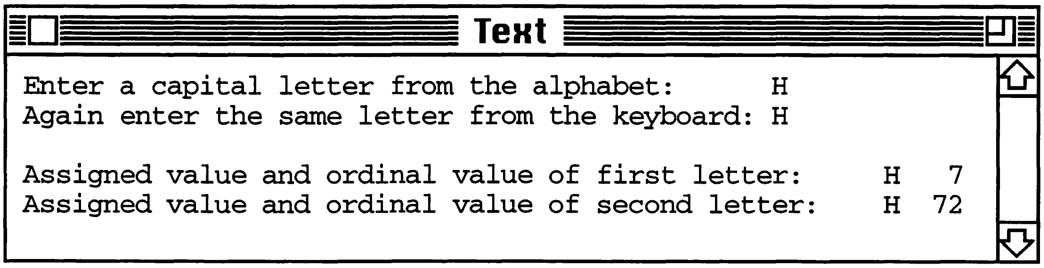
Figure 3.19 Output from the program Date.

Next, we show another short program called Enumerated_Type. This program reveals the difference between an enumerated type representing the letters of the alphabet *A* through *Z* and the standard ordinal type, char. Here the letter *H* has been entered twice. It is assigned to the variable Letter_1 and a second time to the variable Letter_2. The result of executing Enumerated_Type is shown in Figure 3.20.

The ordinal value of H when assigned to variable Letter_1 is 7, since it represents the eighth character from the left in the enumerated declaration of Letter_1. Its ordinal value when assigned to variable Letter_2 is 72, since it now represents the 73rd character in the ASCII character set supported by Macintosh and THINK Pascal. Even though these two variables are assigned values that are letters from the alphabet, they have completely different data types and are completely incompatible with each other.

```
program Enumerated_Type(input, output);
{ Purpose:  demonstration of enumerated and ordinal data types. }
   var
      Letter_1 : (A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P,
                  Q, R, S, T, U, V, W, X, Y, Z);
      Letter_2 : char;
begin
   ShowText;
{ Prompt the user for a letter of the alphabet. }
   write('Enter a capital letter from the alphabet:      ');
   readln(Letter_1);
   write('Again enter the same letter from the keyboard: ');
   readln(Letter_2);
   writeln;
{ Display the ordinal values of Letter_1 and Letter_2 .}
```

```
   writeln('Assigned value and ordinal value of first letter: ',
           Letter_1, ord(Letter_1));
   writeln('Assigned value and ordinal value of second letter:',
           Letter_1, ord(Letter_2));
end.
```



```
Enter a capital letter from the alphabet:      H
Again enter the same letter from the keyboard: H

Assigned value and ordinal value of first letter:     H   7
Assigned value and ordinal value of second letter:    H  72
```

Figure 3.20 Output from the program `Enumerated_Type`.

The `subrange` type is also an `ordinal` data type having a limited but specified range of values. For example, after declaring the variable `Month` above, we might declare other variables called `Spring`, `Summer`, `Fall`, and `Winter`:

```
var
     Month  : (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep,
               Oct, Nov, Dec);
     Spring : Apr..Jun;
     Summer : Jul..Sep;
     Fall   : Oct..Dec;
     Winter : Jan..Mar;
```

The declaration for the variable `Spring` indicates that the range of values allowed is `Apr` through `Jun`. Attempting to declare `Winter` as a `subrange` `Dec..Mar` would cause an error message at translation time, indicating that we are trying to create a `subrange` where the lower boundary `Dec` is greater than its upper boundary `Mar`. This of course is *false*, since `ord(Dec)` is 11, while `ord(Mar)` is 2.

Other `ordinal` data types can also be used to define subranges. Here are some examples:

```
var
     Street_Number : 100..199;
     Numeric_Digit : 0..9;
     Uppercase_Letter : 'A'..'Z';
     Lowercase_Letter : 'a'..'z';
     Character_Digit  : '0'..'9';
```

The variable `Street_Number` is an `integer` type whose value can only be within the range 100 to 199. During execution, an error message will appear if any attempt is made to assign a value that is out of range. The remaining variables, `Uppercase_Letter`, `Lowercase_Letter`, and `Character_Digit`, are all of type `char`, with the exception that each can only be assigned an ASCII character within its `subrange`. The variable `Uppercase_Letter` will only accept the uppercase characters *A* through Z. All other characters such as *a* through *z* and 0 through 9 are out of range for this variable. In turn `Lowercase_Letter` will only accept the lowercase characters *a* through *z*. The variable `Numeric_Digit` is different, since it will accept only a single-integer digit 0 through 9. All other numbers are considered out of range.

Remember to avoid using the names of any `enumerated` or `subrange` values as the names for other variables, for example, in the following declarations.

```
var
   Alphabet : (A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q,
               R, S, T, U, V, W, X, Y, Z );
   J : integer;
```

The variable *J* will cause an error as the program is being translated, indicating that the `enumerated` value `J` has already been declared at this level of the Pascal program.

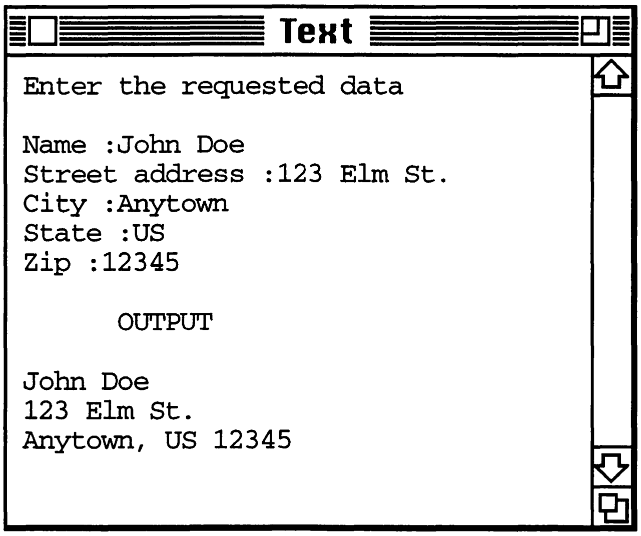
3.5.4 String Types

When we want to manipulate words or phrases (rather than numbers or characters), the appropriate data type is `string`. A `string` data type has some of the characteristics of a simple data type and some of the characteristics of a structured data type. The proper format for the declaration of a `string` variable is

```
identifier : string[n]
```

where `n` is an integer in the range 1–255, representing the maximum length of the `string`. For example, the program called `Address`, listed below, with output shown in Figure 3.21, uses the `string` variable to allow the user to enter and display text material, such as a person's name and address. This program defines a series of `string` variables: `Name, Street, City, State,` and `Zip`. For economy of program lines (not memory), each of these variables is declared to have a maximum length of 80 characters, or one full line of text. The program prompts the user to enter data for each of these variables and then displays the data in a normal address format.

```
program Address (input, output);
{ Purpose:  examples of string data types. }
   var
      Name, Street, City, State, Zip: string[80];
begin
   ShowText;
{ Input address data. }
   writeln('Enter the requested data');
   writeln;
   write('Name :');
   readln(Name);
   write('Street address :');
   readln(Street);
   write('City :');
   readln(City);
   write('State :');
   readln(State);
   write('Zip :');
   readln(Zip);
{ Output the data. }
   writeln;
   writeln('      OUTPUT');
   writeln;
   writeln(Name);
   writeln(Street);
   write(City, ', ');
   write(State, ' ', Zip)
end.
```



```
Enter the requested data

Name :John Doe
Street address :123 Elm St.
City :Anytown
State :US
Zip :12345

      OUTPUT

John Doe
123 Elm St.
Anytown, US 12345
```

Figure 3.21 Output from the program `Address`.

3.6 TYPE DECLARATIONS

In addition to the data types already discussed, you may declare types of your own, using the **type** command. This command is included in the declarations section of the program along with **const** and **var**. The type declaration part of the program begins with the reserved word **type** followed by identifiers equated to data types. Variables declared in the variable declaration part can now be declared with these programmer-defined types. For example, we can create new data types for some of our previously declared variables:

```
type
   Months_of_Year = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
                     Sep, Oct, Nov, Dec);
   Weekdays = ( Sunday, Monday, Tuesday, Wednesday, Thursday,
                Friday, Saturday );
```

```
    Letters = (A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P,
               Q, R, S, T, U, V, W, X, Y, Z );
    Spring_Months = Apr..Jun;
    Range = 100..199;
    Deposit = real;
    Age = integer;
var
    Month  :  Months_of_Year;
    Days_of_Week : Weekdays;
    Alphabet : Letters;
    Spring : Spring_Months;
    Street_Number : Range;
    Weekly_Deposit , Monthly_Deposit : Deposit;
    Age_of_Adult, Age_of_Child : Age;
```

Notice that in defining our own data type, we begin with an identifier name followed by an equal sign followed by a data type. The equal sign implies that the identifier on the left is equated with the data type on the right. As you can see by this last example, the data type can be enumerated, ordinal, or real. In our example, the variable Month is declared to be associated with a data type called Months_of_Year, and in the type declaration Months_of_Year is equated with an enumerated type. Although these types may seem to be variations on an old theme, they can be used to make a program more readable and can also save the programmer time in declaring variables. This concept is also important because Pascal on the Macintosh has many different data types that can be borrowed from the libraries such as **QuickDraw1**, **QuickDraw2**, and **SANE**. Appendix D provides a list of these predeclared data types.

The program shown below is a revision of our previous program called Date. The variation in the program is the addition of three new programmer-defined data types: Months_of_Year, Days_of_Month, and Numbers_in_Year. The latter two datatypes are declared as subrange values.

```
program Second_Date(input, output);
{ Purpose:  demonstrate the use of enumerated and subrange user-}
{           defined types. }
   type
      Months_in_Year = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
                        Sep, Oct, Nov, Dec);
      Days_of_Month = 1..31;
      Numbers_in_Year = 0..99;
   var
      Month: Months_in_Year;
      Day: Days_of_Month;
      Year: Numbers_in_Year;
      Slash: char;
begin
   ShowText;
{ Prompt the user for today's date. }
   writeln(' Enter todays date using the following ');
   writeln(' format: month/day/year where month');
   writeln(' is written as three characters, day as');
   writeln(' a whole number, and year as the last two');
   write(' digits of the current year: ');
   readln(Month, Slash, Day, Slash, Year);
   writeln;
{ Display today's date in the following format: }
{ Month - Day - 19 _ _ . }
   write('Today`s Date : ');
   writeln(ord(Month) + 1 : 2, ' - ', Day : 2, ' - ', '19', Year :
           2);
end.
```

It is important to remember that data types defined under a type declaration in Pascal are not *true* data objects like constants and variables. Even though these data types are associated with names and are equated with either a `real` or `ordinal` data type, they are not associated with values during the time the program is being translated, nor with the storage of values as the program is executed.

3.7 THINK PASCAL VERSUS STANDARD PASCAL

There are minor differences between THINK Pascal and standard Pascal regarding simple data types and the naming of identifiers. Standard Pascal allows the length of identifier names to be unlimited, while THINK and Macintosh Pascal limit the length of identifier names to 255 characters. In standard Pascal, identifier names are limited to digits and letters of the alphabet, while THINK and Macintosh Pascal allow the underscore as one of the characters of an identifier. When a compiler is implemented, exceptions are allowed, since keyboards may support character sets that are different. Some systems allow both the underscore and dollar sign ($).

How `read` and `readln` execute depends upon the implementation of the particular Pascal compiler. For example, when reading from standard input (keyboard), pressing the Return key may be sufficient to terminate an input line. In other instances, the `readln` must be executed to force an input line to be properly terminated. In some systems, the physical input buffer (the area of the terminal holding the characters being read on input) may be limited to 255 bytes (characters). Attempting to execute several `read` statements without executing a `readln` can result in the content of the input buffer being lost. Keep in mind that these can be hardware as well as operating-system dependencies.

Not all Pascal systems require a `writeln` statement to terminate a line of output to the screen. For THINK and Macintosh Pascal, a line of output is terminated if input is read, and the character string representing an input value is terminated by pressing the return key. In some systems, a `writeln` statement is important to terminate an output line and prevent the output buffer of the terminal from exceeding a fixed length and losing the characters it contains.

Standard Pascal only allows simple data types such as `real`, `integer`, and `char` to be read using the Pascal commands `readln` or `read`, and it only allows the values of `real`, `integer`, `char`, and `Boolean` variables to be written using the commands `write` and `writeln`. Both Macintosh and THINK Pascal extend input and output by allowing Pascal programs to read or write values of `enumerated` types. This includes `Boolean` types as well as programmer-defined `enumerated` types such as those given as examples in Section 3.5.3.

Both THINK and Macintosh Pascal support simple data types beyond `real`, `integer`, `Boolean`, and `char`. These include the numeric types `double`, `extended`, `longint`, and `computational`, and allow for large, precise floating-point and integer arithmetic operations. In both Macintosh and THINK Pascal, `real` types are converted to `extended` mode when `real` arithmetic operations are performed. A math coprocessor can help speed execution.

Standard Pascal does not define a `string` type. In standard Pascal, strings are manipulated through the use of a packed array of characters. The concept of a `string` type was first introduced in UCSD Pascal and has remained a standard type among Pascal compilers written for personal computers such as the Macintosh. The `string` type differs from a packed array of characters in that it implies the dynamic storage of strings. A packed array of characters uses a static storage structure where all array positions in memory are assumed to be used. While Macintosh and THINK Pascal allow character strings to be read and written for variables declared as `string` types, standard Pascal can only read character strings by continuous read operations of a single character at a time. Both allow a packed array of characters to be written by execution of a `write` or `writeln` command.

SUMMARY

This chapter began with a discussion of the general form of the Pascal program, particularly the heading. We next discussed constants and variables and how we can use them to increase the power and efficiency of the program. We then discussed the various simple data types: four `real` types (`real`, `double`, `extended`, `computational`), standard `ordinal` types (`integer`, `longint`, `char`, `Boolean`), and nonstandard `ordinal` types (`enumerated`, `subrange`). Following the discussion of numeric data types, we discussed the `string` data type, which allows the user to manipulate non-numeric data. The final discussion was devoted to the **type** command, a feature of Pascal that allows the programmer to create custom data types. Throughout the chapter, we explored the use of the input and output commands: `read`, `readln`, `write`, and `writeln`.

REVIEW QUESTIONS

1. What basic parts make up a Macintosh Pascal program?
2. What are the rules for naming a Macintosh Pascal program?
3. Are the following identifier names syntactically correct? Briefly explain why each identifier name is legal or illegal.

```
Income  Tax  Program     Circle_System
Inventory-System         PatternMaker
1234_Program             Income&Taxes
```

4. Are the special parameters `input` and `output` required in the program heading of a Macintosh or THINK Pascal program?
5. What is the purpose of the **uses** clause? How do THINK Pascal and Macintosh Pascal differ with respect to this command?
6. What is meant by the term *library* ?
7. Name the three special libraries of Macintosh Pascal. Name some of the libraries available in THINK Pascal.
8. How are comments inserted in a Pascal program?
9. What is the last statement in a Pascal program?

10. Is the body of a Pascal program bracketed by **begin** and **end;** or **begin** and **end.** ?
11. Draw a diagram representing the concept of a data object.
12. Briefly explain the difference between a constant and a variable.
13. Are the following correct constant statements?

```
const
   Constant_Value := 123.67;
   Second_Value = 34.87;
   Angle_One = pi;
   Angle_Two = 2 * pi;
   Angle_Three = Angle_One;
   Truth = true
   Message = 'This is a string.;
```

14. Is the following code syntactically correct for a Macintosh Pascal program?

```
const
   Truth = true;
const
   Angle = pi;
const
   Maximum = 9999;
   Minimum = - Maximum;
```

15. What is the logical implication of using the equal sign alone instead of := in establishing a value for a constant identifier?
16. List the constants known by Macintosh Pascal and their values.
17. What does the reserved word **var** represent?
18. Are the following declarations syntactically correct?

```
var
   Number : integer;
   Total-income ; real;
   LogicalValue :  Boolean
   Name : string;
   Number : real;
   Pi : integer;
```

19. Is the following valid in Macintosh Pascal?

```
var
   Person : string;
var
   Income : real;
var
   City_State : string;
```

20. Is the following program allowed in Macintosh Pascal or in THINK Pascal?

```
program Example(input, output);
   var
       Cost_of_Item : real;
       Sales_Tax : real;
   const
       Tax = 0.05;
begin
{ An empty example. }
end.
```

21. What is meant by the terms `input` and `output` ?
22. Explain the difference between the statements `write` and `writeln`.
23. Are the words **`write`** and **`writeln`** reserved words?
24. Use the Instant window to execute the following output commands:

```
write(  1 , ' + ' , 1 ' is ' , 2 );
write( 2 );
writeln(2);
writeln( true, false )
writeln(true,  false,  maxint,  maxlongint );
```

25. Enter and test the program titled `Display_Text` from Section 2.4.1.
26. Explain the difference between the commands `read` and `readln`.
27. When executed, how do the following three sets of Pascal commands differ?

```
write(' Enter your total income: $' );
readln( Income );

writeln(' Enter your total income: $' );
readln( Income );

write(' Enter your total income: $' );
read( Income );
writeln;
```

28. List the simple data types of Macintosh Pascal, and briefly explain what each type represents.
29. What is meant by the terms *floating-point number* and *scientific-notation* ?
30. What are `ordinal` data types?
31. List the standard `ordinal` data types of Macintosh Pascal.
32. What is the range of `longint` and `integer` data types in Macintosh and THINK Pascal?
33. Enter and test the program `Second_Counter` with the declared data objects being `integer` types. Test the same program with the declared data objects being `longint` types.
34. What is meant by the term *function* ?
35. How do the functions `ord`, `pred`, and `succ` differ?
36. What are the nonstandard `ordinal` data types in Macintosh and THINK Pascal?
37. What is the difference between an `enumerated` type and a `subrange` type?
38. Show how a variable called `Colors` can be represented as an

enumerated data type having the following possible values: red, orange, blue, green, black, white, yellow, pink, violet.

39. Declare a variable called Range_of_Income having minimum and maximum incomes of 1000 and 9999.
40. Can a constant be an enumerated type? Can a constant be a subrange type? Support your answers with some simple examples.
41. What is the difference between a char type and a string type?
42. Declare the following data objects as strings:

```
Full_Name      { a string with a maximum length of 40 characters }
Street_Address { a string with a maximum length of 30 characters }
City           { a string with a maximum length of 15 characters }
State          { a string of length 2 characters }
Zip_Code       { a string of length 11 characters }
```

43. What purpose does the type declaration serve in Macintosh Pascal?
44. Rewrite the declaration in Question 38 by introducing a programmer-defined data type called Color_Chart and redeclaring the variable Colors as a type called Color_Chart.
45. Enter and test the program shown in Figure 3.16.

PROGRAMMING EXERCISES

1. Write a Macintosh Pascal program that will perform the following steps:

   (a) Prompt for an integer, and read that number.
   (b) Prompt for a second integer, and read that number.
   (c) Prompt for a third integer, and read that number.
   (d) Compute the sum of these three numbers and assign it to Total.
   (e) Display the result of the summation to the Text window.

2. Write a Macintosh Pascal program that will perform the following steps:

   (a) Prompt the user for the radius of a circle, using a write statement.
   (b) After entering a real number using the readln command, compute the area of a circle using the formula

   ```
   area = π * radius
   ```

   where the character * represents multiplication.
   (c) Output the following information to the Text window, using writeln statements:

   ```
   Radius:            { value of radius }
   Area of Circle:    { value of area }
   ```

3. Write a Macintosh Pascal program to compute the circumference of a circle, using the formula

```
circumference = 2 * π * radius
```

Output the result by displaying the value of `Radius` and `Circumference` to the Text window. This program will require `write`, `readln`, and `writeln` statements.

4. The area of a triangle is specified by the following formula:

```
area = ( base * height )/2.0 .
```

Write a program that will perform the following steps:

(a) Prompt for the value of the base.
(b) Prompt for the value of the height.
(c) Compute the area using the given formula.
(d) Output the results of the computation to the Text window.

5. Write a program that will paint a circle in the Drawing window using the procedure `PaintCircle`. This program requires the following steps:

(a) Prompt for the center of the circle:

```
read the X_Center;
read the Y_Center.
```

(b) Prompt for the radius of the circle, and read the radius.
(c) Paint the circle in the Drawing window.

6. In the metric system of measurement, 1 inch is equal to 0.0254 meters. Write a program that prompts for distance measured in inches and computes the distance in meters, using the constant 0.0254. Output your results with the following `writeln` statement:

```
writeln(' Distance in meters: ', Distance:15:5 );
```

7. In converting Fahrenheit temperature to Celsius, we can use the following formula:

```
Celsius =  5.0 * ( Fahrenheit - 32 ) / 9.0
```

Write a program that will prompt for the temperature in Fahrenheit and display the temperature in Celsius degrees. Use the following `writeln` statement to display this result:

```
writeln(' Celsius temperature: ',  Celsius_Temperature : 7:2 );
```

Use the following checks:
212° F is equivalent to 100° C
32° F is equivalent to 0° C

8. Write a program that will prompt for total cost of items and sales-tax rate, compute the sales tax and the total cost (representing the sum of total cost of items and sales tax), and display the following to the Text window:

```
Total cost of items:    $     { value of the items }
Sales tax:                    { value of the sales tax }
_________________________
Total cost of items:    $     { value of the total cost }
```

9. Write a program to perform the following steps:

   (a) Prompt the user for the month as a `string`, and read that input using the `readln` command.
   (b) Prompt the user for the day as a `string`, and read that input using the `readln` command.
   (c) Prompt the user for the year as a four-digit integer, and read that value using `readln`.
   (d) Display today's date to the Text window, using the format month/day/year.

10. Write a Macintosh Pascal program using three different `string` variables: `Last_Name`, `First_Name`, and `Middle_Name`. Your program will require the following steps during execution:

    (a) Prompt for the first name of a person, and read this name.
    (b) Prompt for the middle name of a person, and read this name.
    (c) Prompt for the last name of a person, and read this name.
    (d) Display the person's full name to the Text window, using the following format:

    Last name , First name (blank space) Middle name

11. Repeat Exercise 10 to display the person's full name as follows:

    First name (blank space) Middle name (blank space) Last name

12. Write a program using `writeln` commands that will display the following truth table:

A	B	Not A	A and B	A or B
False	*False*	*True*	*False*	*False*
True	*False*	*False*	*False*	*True*
False	*True*	*True*	*False*	*True*
True	*True*	*False*	*True*	*True*

13. Write a program that prompts the user for an integer, and, after reading the number, displays the following:

```
pred(Number)        Number        succ(Number)

value of pred      value of     value of succ
  of number         number        of number
```

14. Write a program that prompts for the day of the week and, after the word `Sunday`, `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, or `Saturday` has been entered, displays the following to the Text window:

```
Day of the week:               { display the value of weekday }
Predecessor of the weekday:    { display pred of weekday }
Successor of the weekday:      { display succ of weekday }
```

Define the data type called `Days_of_Week`, using the following declaration:

```
type
   Days_of_Week= ( Sunday, Monday, Tuesday, Wednesday,
                   Thursday, Friday, Saturday);
```

(*Note* : Some values will cause execution errors for either the predecessor or successor functions.)

Chapter 4

Basic Arithmetic Operations, Expressions, and Assignment Statements

OBJECTIVES

After completing Chapter 4, you will know the following:

1. The distinction between an operator and an operand in a Pascal expression.
2. Pascal operators including +, –, *, /, **div**, and **mod**.
3. The nature and importance of operator precedence, including the effect of parentheses.
4. The use of the assignment statements in a Pascal program, and the meaning of the Pascal operator := .
5. How to trace a program with the help of the Observe window.
6. The Macintosh Pascal and THINK libraries of arithmetic functions, and the use of these functions in Pascal programs.

4.1 THE OPERAND AND THE OPERATOR

Consider the following short program, entitled `Addition`:

```
program Addition(input, output);
{ Purpose:  This program demonstrates the operation of addition. }
   const
      X = 10;
      Y = 5;
   var
      Z : integer;
begin
   ShowText;
{ Add the values of the constants and display their results. }
   Z := X + Y;
   writeln('The sum of X and Y is: ', Z : 3);
end.
```

This program includes the assignment statement

```
Z := X + Y
```

where the element to the right of the colon-equal sign, `X + Y`, is composed of two *operands*, `X` and `Y`, and the symbol + representing the *operator* for addition. When executed, the operator + instructs the Macintosh computer to sum the values of the constants `X` and `Y` . In this context the addition operator is called a *binary operator*, because it requires two operands to perform the operation. This statement also contains a second operator represented by the combined characters := . This operator represents the assignment of a value to a variable. It is also a binary operator, since it requires two operands during execution: the value of the expression to the right of the assignment operator and the variable to the left. When execution of the assignment statement is complete, the value of the expression on the right is assigned to the variable on the left. This results in the value of the expression being stored in memory cells addressed by variable `Z`. This program can be modified to do subtraction (–), multiplication (*), and division (/) by substituting the appropriate symbol (–, *, or /) for the addition operator in the statement `Z := X + Y`.

A *unary operator* requires only one operand. The unary arithmetic operators in Pascal are the "unary +" and the "unary –." In the assignment statement

```
N := -( B + C )
```

the minus sign (unary –) causes the value within the parentheses `(B + C)` to be negated. That is , if the value of `(B + C)` is positive, `- (B + C)` will be negative, and if the value of `(B + C)` is negative, `- (B + C)` will be positive. The operator unary + is referred to as an *identity operation*. This means that it leaves the value unchanged when applied as a unary operator. For example, the value of the expression + `(B + C)` is the same as the value of `(B + C)`. Both binary and unary operations for Macintosh and THINK Pascal are listed in Figure 4.1.

Notice that expressions such as

```
- ( + (B + C) )
+ ( - ( B + C ) )
+ ( + ( B + C ) )
- ( - ( B + C ) )
```

are syntactically valid, while expressions such as

```
- + ( B + C )
+ - ( B + C )
+ + ( B + C )
- - ( B + C )
- + B + C
+ - B + C
+ + B + C
- - B + C
```

are invalid. Combinations of operator symbols such as `- -`, `- +`, `+ +`, `+ -` are not recognized as valid unary operations. In general, the best rule is to follow each unary operator with a set of parentheses placed around the expression that the unary operator is acting on. This helps to avoid errors whenever expressions are either typed or checked.

Although many of the Macintosh and THINK Pascal arithmetic operators are represented by single-character symbols, two special arithmetic operators require short syllables: **`div`** and **`mod`**. The use of **`div`** and **`mod`** requires sensitivity to the use of data types when constructing expressions, since they are "integer-only" operators: they require operands that can only be `integer` data types. With some arithmetic operators, the mixing of values for `integer` and `real` variables can produce either `real` or `extended` results. In standard Pascal, however, **`div`** and **`mod`** produce only `integer` results.

Unary Operators (signs)		
Operator	Operation	Operand types
+	Identity	`Integer` or `real`[a]
-	Sign negation	`Integer` or `real`[a]

Binary Operators		
Operator	Operation	Operand types
+	Addition	`Integer` or `real`[b]
-	Subtraction	`Integer` or `real`[b]
*	Multiplication	`Integer` or `real`[b]
/	Division	`Integer` or `real`
`div`	Division with `integers`	`Integer`
`mod`	Modulo	`Integer`

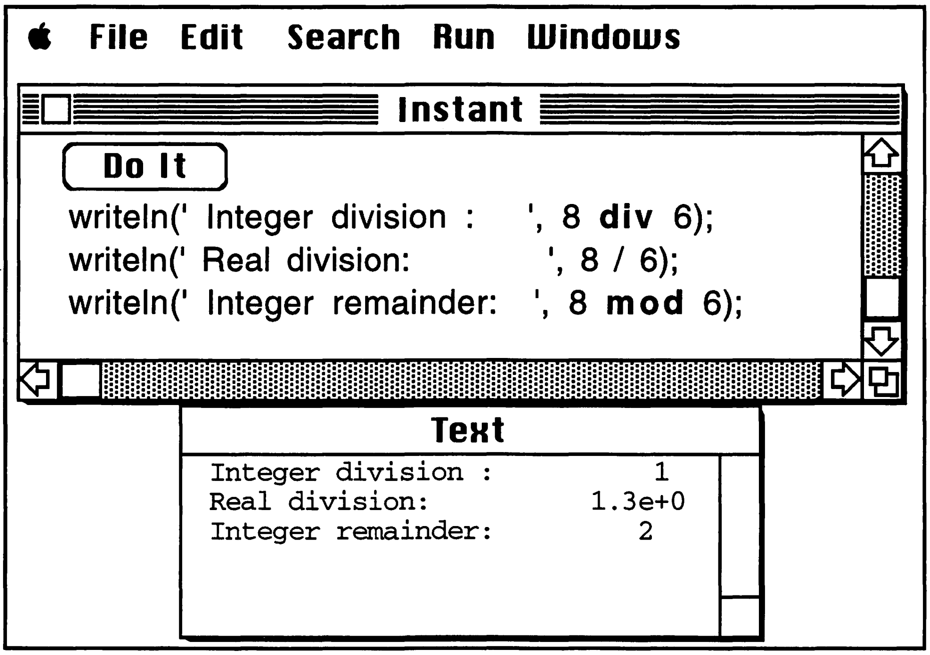
[a] The result is `real` if the operand type is `real`, and `integer` if the operand type is `integer`.

[b] The result is `real` if both operand types are `real` or if one is `real` and the other is `integer`.

Figure 4.1 The unary and binary arithmetic operators.

How do the three arithmetic operators / , **`div`** , and **`mod`** differ? We use the operator **`div`** when both operands are `integer` types, and when the result to be computed is the *quotient* of one `integer` value divided by another `integer` value. The **`mod`** operator, a division operator, yields as a result the *remainder* of one `integer` value divided by another `integer` value. For example, 5 **`mod`** 2 yields a result of 1 (5 divided by 2 yields 2 with a remainder of 1). The division operator, /, is different: it results in a `real` value whether the operands are of `integer`, `real`, or mixed data types.

As an example, look at the Text window shown in Figure 4.2 . In this example we have selected the Instant window from the menu option **Windows** (**Debug** for THINK Pascal),[1] and typed three `writeln` statements to test the differences among the three binary division operators.



```
writeln(' Integer division :    ', 8 div 6);
writeln(' Real division:         ', 8 / 6);
writeln(' Integer remainder:  ', 8 mod 6);
```

```
Integer division :        1
Real division:       1.3e+0
Integer remainder:        2
```

Figure 4.2 Application of the Instant window in Macintosh Pascal.

Note that the Program and Drawing windows can be either opened or closed. We have elected to close these windows. After typing all three statements, use the option **Check** to check for any syntax errors. If any text exists in the Text window, choosing the option **Reset** from the menu option **Run** will clear the window. To execute any of the statements that are in the Instant window, click the **Do It** button. In this case the three separate statements are checked and then executed, so that three separate lines of text are displayed in the Text window. For the `integer` constant 8 divided by `integer` constant 6, the first line of text gives the value of the quotient, the second gives the result for real division in terms of a floating-point value representing 1.333333... (a combination of the quotient as a whole number and the remainder as a fraction), and the third line displays the remainder. Notice that both the `integer` division operators **div** and **mod** give `integer` results.

[1] If you do this exercise with THINK Pascal, you will discover that the Instant window is active only when a program has been executed and halted. You may achieve this state by opening a project, executing the **Go** command, and then halting execution by clicking the spray-can icon. If your program executes too quickly to allow this, you may insert one or more stops to halt execution. The statements you test in the Instant window need not be related to the program whose execution you halt.

As a second example of using the **div** operator, consider an example Pascal program designed for computing the percentage of partisan votes in an election. When executed, this program prompts the user to enter the number of votes for Democratic candidates and the number of votes for Republican candidates, and then it computes the total partisan vote, the percentage of Democratic votes, and the percentage of Republican votes. All percentages are to be stored as whole numbers. After all computations have been performed, the total number of votes and the percentages are displayed to the Text window.

Here is the algorithm for developing this program:

```
Algorithm Election;
{ This algorithm computes the percentage of Democratic and
   Republican votes in a partisan election. }
{ List of data types:
   Democrats, Republicans, Total_Votes : integer
   Percent_Democrats, Percent_Republicans : integer }
begin
{ Prompt user for Democratic and Republican votes. }
   write 'Enter the number of Democratic votes: ' ;
   read Democrats ;
   write 'Enter the number of Republican votes: ' ;
   read Republicans ;
{ Compute the total number of votes and the percentages. }
   Total_Votes <-- Democrats + Republicans;
   Percent_Democrats <-- ( Democrats * 100 )  div Total_Votes;
   Percent_Republicans <-- ( Republicans * 100 )  div Total_Votes;
{ Display total votes and the percentages. }
   write Total_Votes ;
   write Percent_Democrats ;
   write Percent_Republicans ;
end.
```

The Pascal program for algorithm `Election` follows. Here we have added `writeln` statements to help when displaying output. In addition, the constant 100 has been replaced with an identifier called `Base`.

```
program Election(input, output);
{ Purpose:  A demonstration of the operator div. This program }
{           computes and displays the percentage of Democratic }
{           and Republican votes in a partisan election.}
   const
      Base = 100;
   var
      Democrats, Republicans, Total_Votes : integer;
      Percent_Democrats, Percent_Republicans  : integer;
begin
   ShowText;
{ Prompt user for Democratic and Republican votes. }
   write( 'Enter the number of Democratic votes: ');
   readln( Democrats);
   write( 'Enter the number of Republican votes: ');
   readln( Republicans);
{ Compute the total number of votes and the percentages. }
   Total_Votes   := Democrats + Republicans;
   Percent_Democrats := ( Democrats * Base )  div  Total_Votes;
   Percent_Republicans := (Republicans * Base)  div Total_Votes;
{ Display total votes and the percentages. }
   write('A total of ', Total_Votes : 2 );
   writeln(' votes were cast in this election');
   writeln;
   write('  The Democrats received ', Percent_Democrats : 2 );
   writeln(' percent of the vote.');
   writeln;
   write('  The Republicans received ', Percent_Republicans : 2 );
   writeln(' percent of the vote.');
end.
```

If you want slightly greater precision, use the real division operator, /, to produce `real` (decimal) results. This would require that the variables `Percent_Democrats` and `Percent_Republicans` be declared as type `real` instead of `integer`. Why? A syntax rule in Pascal is that a variable of type `integer` cannot be assigned a value of an expression of type `real`.

The program `Election_Revised` shows the changes necessary to produce percentages with decimal-fraction precision.

```
program Election_Revised(input, output);
{ Purpose:  A demonstration of the operator div. This program }
{           computes and displays the percentage of Democratic }
{           and Republican votes in a partisan election.}
   const
      Base = 100;
   var
      Democrats, Republicans, Total_Votes : integer;
      Percent_Democrats, Percent_Republicans   : real;
begin
   ShowText;
{ Prompt user for Democratic and Republican votes. }
   write( 'Enter the number of Democratic votes: ');
   readln( Democrats);
   write( 'Enter the number of Republican votes: ');
   readln( Republicans);
{ Compute the total number of votes and the percentages. }
   Total_Votes   := Democrats + Republicans;
   Percent_Democrats := ( Democrats * Base ) / Total_Votes;
   Percent_Republicans := ( Republicans * Base ) / Total_Votes;
{ Display total votes and the percentages. }
   write('A total of ', Total_Votes : 2 );
   writeln( ' votes were cast in this election');
   writeln;
   write('  The Democrats received ', Percent_Democrats : 5 : 2 );
   writeln(' percent of the vote.');
   writeln;
   write('  Republicans received ',Percent_Republicans : 5 : 2 );
   writeln(' percent of the vote.');
end.
```

The only difference between `Election` and `Election_Revised` is in the way the percentages are computed and displayed. `Election` displays percentages as whole numbers, whereas `Election_Revised` displays percentages in a decimal format with a field of five characters and two digits to the right of the decimal point.

Modulo, or **mod**, is an `integer` division operation that gives as a result the remainder of an `integer` division process. It is important to understand that special conditions apply when using this operator:

1. If `M` is not zero, `N` **mod** `M` is equivalent to `N - (N` **div** `M) * M`.
2. In THINK Pascal, `N` **mod** `M` results in an executable error if the value of `M` is zero.
3. In standard Pascal and in Macintosh Pascal, `N` **mod** `M` is in error if `M` is either zero or negative.
4. When `N` is negative and `M` is greater than 0, the **mod** operation in Macintosh Pascal is not consistent with THINK Pascal. The **mod** operation in Macintosh Pascal is only consistent when `N` is greater than or equal to 0, and `M` is greater than 0.

Here `N` and `M` are assumed to be `integer` types. The following program, `Modulo_Demonstration`, illustrates the use of the **mod** operator. The statement

```
Remainder := Dividend mod Divisor;
```

is assumed to produce an `integer` remainder for any two integers entered from the keyboard, one represented by the variable `Dividend` and the other by the variable `Divisor`. This program is useful for demonstrating the results of the three conditions just described. The modulo operator has some interesting applications, several of which are shown in Chapters 5 and 6.

```
program Modulo_Demonstration(input, output);
{ Purpose:  This program demonstrates the operator mod. }
   var
      Dividend, Divisor, Remainder : integer;
begin
   ShowText;
{ Prompt user to enter two numbers. }
   writeln('Enter a number as a dividend: ');
   readln(Dividend);
   writeln;
   writeln('Enter a number as a divisor: ');
   readln(Divisor);
   writeln;
{ Perform the mod computation and display the result. }
   Remainder := Dividend mod Divisor;
   writeln(Remainder : 2, ' is the remainder of ', Dividend : 2);
   writeln(' divided by ', Divisor : 2, ' . ');
end.
```

4.2 OPERATOR PRECEDENCE

The order in which arithmetic operations are performed is called *precedence*. In Chapter 3 we pointed out the need for precision and lack of ambiguity in writing an algorithm. To achieve this, we must be able to evaluate an expression in an unambiguous manner. For example, there must be one and only one answer to evaluating the following expression:

```
B + C * D.
```

Rules of precedence for arithmetic operators are summarized in Figure 4.3. In Macintosh and THINK Pascal, the unary operators + and – have equal operator precedence with addition and subtraction.

Operators [a]	Level of Precedence
* , / , **div** , **mod** [b]	(highest)
+ , - [c]	(lowest)

[a] Precedence rules for all operators in Macintosh and THINK Pascal are given in Figure 5.5.

[b] The multiplication and division operators have equal precedence among themselves.

[c] The addition and subtraction operators have equal precedence among themselves.

Figure 4.3 Precedence rules of arithmetic operators in Pascal.

Performing addition followed by multiplication could result in entirely different results from performing multiplication followed by addition. The rules of precedence are meant to eliminate this ambiguity. These rules require an expression to be evaluated in an order determined by the precedence of the operators involved in the expression. Otherwise, execution of operators of the same level of precedence and the operands to which they are bound is from left to right. Some of these rules are demonstrated in the program `Precedence`.

```
program Precedence(input, output);
{ Purpose:  Demonstrates the order of precedence in the }
{           evaluation of an expression. }
   var
      A,B,C,D : integer;
begin
   ShowText;
{ Assign values to the variables B, C, and D. }
      B := 5;
      C := 10;
      D := 2;
{ Evaluate the expression and display the results. }
   A := B + C * D;
   writeln('The expression 5 + 10 * 2 is evaluated as: ', A :  1);
end.
```

With the operand B assigned the value 5, C assigned the value 10, and D assigned the value 2, the value computed and assigned to A is 25 according to the rules of precedence given in Figure 4.3. Since the multiplication operator (*) has a higher precedence than the addition operator (+), the multiplication operator (*) is executed first.

However, if we substitute the expression

```
A := B / C * D;
```

for the expression given in `Precedence`, the results will change. The multiplication operator (*) and the division operator (/) have the same level of precedence, so execution is performed from left to right. In this example, the division is performed before the multiplication. If we assign the same values to the variables B, C, and D as before, the value of the expression `B / C * D` becomes 1.0. *However, before we can execute this revised program, we must make the following change:*

```
var
   A : real;
   B, C, D : integer;
```

This change is required because the division operation is `real` rather than `integer`. Without this change, the program will produce an error message informing you of an incompatibility of types.

Parentheses can be used to alter the order in which operations are performed in an expression. In the statement

```
A := B + C * D;
```

the variable A becomes 25 when the constants B, C, and D are defined as 5, 10, and 2, as in the program `Precedence`. We can change the order in which these operations are performed by using parentheses as follows:

```
A := ( B +  C ) * D;
```

Now when the expression is evaluated, the value of A becomes 30, because the operation within parentheses is executed before operations outside the parentheses.

Expressions can be nested within parentheses as long as the parentheses are balanced, that is, paired. For example, the statement

```
A := B - ( C * ( D + F ) );
```

results in the addition operation (+) being executed first, the multiplication operation (*) next, and the subtraction (–) operation last. The rule when using parentheses is that the expression contained in the innermost pair is evaluated first. Then the expression contained within the next level of parentheses is evaluated, and so forth, until execution is complete. Execution of the program `Parentheses` demonstrates that under these circumstances, A is evaluated as –45.

```
program Parentheses(input, output);
{ Purpose:  This program demonstrates the effect of parentheses }
{           on operator precedence. }
   var
      A,B,C,D,F : integer;
begin
{ Assign values to the variables A through F. }
      B := 5;
      C := 10;
      D := 2;
      F := 3;
{ Evaluation of the expression and display of the result. }
   A := B  -(C * (D + F));
   writeln('A = ', A : 1);
end.
```

4.3 EXPRESSIONS AND THE ASSIGNMENT STATEMENT

The construction of a valid statement is based on two primary rules:

1. Valid statements have one and only one variable on the left side of the statement.
2. No operators may be on the left side of the statement.

For example, the statement

```
X + Y := 12 + Z
```

is invalid because it violates both of these rules: two variables, `X` and `Y`, and the operator + are found on the left side. To make this a valid statement, we must show only one variable on the left, as in the following:

```
X := 12 + Z
```

An assignment statement has the basic form of

`Variable :=` *`expression`* `;`

where *`expression`* may be any of the following:

1. A variable
2. A numeric constant
3. *`expression`* operator *`expression`*

For example, all of the following are valid assignment statements:

```
X := Y;
X := 13;
X := Y * 8
```

The := denotes that the data object on the left side of the statement, called a *variable*, is assigned a value obtained from executing the expression. This allows statements such as

```
X  :=  X + 1;
```

In Pascal, the statement `X = X + 1`, where = is read as "is equal to," is illogical. Equality implies balance in value for both sides of an equation. The statement `X := X + 1` does not express balance. Rather, the value of 1 is added to the value of `X` when the expression on the right side of the statement is executed. The resultant value is then assigned to the variable `X` on the left side of the assignment operator (:=).

Notice that a statement such as `X = 13` represents an expression of equality and is used to establish the value of `X` as a constant. Such expressions are found in the constant declarations of a Pascal program, under the **const** heading. The program `Conversion` illustrates these points.

```
program Conversion(input, output);
{ Purpose:  This program converts a Celsius temperature, entered }
{           by the user, into a Fahrenheit temperature. The }
{           result is displayed on the screen. }
   const
      K = 32;
   var
      Celsius, Fahrenheit : integer;
begin
{ Prompt the user to enter the Celsius temperature. }
   write('Please enter a Celsius temperature: ');
   readln(Celsius);
{ Convert Celsius temperature to Fahrenheit temperature. }
   Fahrenheit := ((Celsius * 9) div 5 + K);
{ Display the results.}
   writeln('The Fahrenheit equivalent is: ', Fahrenheit : 2);
   writeln;
end.
```

This program illustrates the use of both = and := in assigning values to data objects. The constant used in the conversion of a Celsius temperature to a Fahrenheit temperature is 32. Thus the constant `K` is declared and its value equated with 32. This statement uses the =, since an equality is being established.

```
const
   K = 32;
```

Later in the program the actual conversion is accomplished in the following statement:

```
Fahrenheit := ((Celsius * 9) div 5 + K);
```

The result is `integer`, since `Fahrenheit` is declared as an `integer` type, but the program can compute `real` values by replacing **`div`** with / and by declaring both `Celsius` and `Fahrenheit` to be `real`.

As an added example, let us develop both an algorithm and a program for reversing the digits of an `integer` number. Assume that we want a program to prompt the user for a four-digit `integer` number and, after this number has been entered, display the digits in reverse order. For example, the user would enter a number such as 9753, and the program would display the value 3579 . In writing both the program and the algorithm, let us make some basic assumptions:

1. We assume that our initial number is represented by the digits $d_1d_2d_3d_4$. The new number will be represented by the digits $d_4d_3d_2d_1$.
2. Our initial number must always be positive.
3. All individual digits must be between 1 and 9; 0 is not allowed.

The following steps represent an initial solution:

1. Read a four-digit `integer` number entered from the keyboard.
2. Extract each of the four digits from the initial number.
3. Create a new number from the four separate digits.
4. Write the value of the new number.

The second and third steps are ambiguous and must be refined. For the second step, let us assume that our digits will have unique names: `Digit_1`, `Digit_2`, `Digit_3`, and `Digit_4`. To obtain the leftmost digit, we need only compute the following step:

```
Digit_1 <-- Initial_Number div 1000
```

To compute the second digit from the left, we remove the first digit on the left by using one of two statements:

```
Remainder <-- Initial_Number mod 1000
```

or

```
Remainder <-- Initial_Number - ( Digit_1 * 1000 )
```

We will choose the former and leave the latter as an exercise. The result is a remainder representing a new number having only three digits: $d_2d_3d_4$.

Next we extract the remaining digits by repeating the same steps but with `Initial_Number` replaced by `Remainder`. The following is a refinement of our initial algorithm:

```
Algorithm Reverse_Digits;
{ Reverse_Digits reverses the digits of a four-digit integer
      number. }
{ List of variables:
   Initial_Number, New_Number, Remainder : integer
   Digit_1, Digit_2, Digit_3, Digit_4 : integer }
begin
{ Enter a four-digit integer number from the keyboard. }
   write 'Type a four-digit integer number: ';
   read Initial_Number;
{ Extract each of the four digits from number. }
   Digit_1 <-- Initial_Number div 1000;
   Remainder <-- Initial_Number mod 1000;
   Digit_2 <-- Remainder div 100;
   Remainder <-- Remainder mod 100;
   Digit_3 <-- Remainder div 10;
   Digit_4 <-- Remainder mod 10;
{ Create the new four-digit number. }
   New_Number <-- (Digit_4 * 1000) + (Digit_3 * 100) +
                               (Digit_2 * 10) + Digit_1;
{ Display the value of our new number. }
   write New_Number
end.
```

To test our newly defined algorithm and see if it is functional, we can trace its steps by establishing a trace table, as shown in Figure 4.4. At the top of the table is a list of variables, and to the left is a simple counter called `Step`. Each time a step is evaluated, we record the changes to any of the values of variables. (Figure 4.6 presents the Pascal program, but without comments.)

Step	Initial_ Number	Digit_1	Digit_2	Digit_3	Digit_4	Remainder	New_ Number
1	8642						
2		8					
3						642	
4			6				
5						42	
6				4			
7						2	
8					2		2468

Figure 4.4 Trace table for the algorithm `Reverse_Digits`.

In writing an assignment statement, we must always consider the issue of compatibility between the data types of an expression and a variable. In Macintosh Pascal, data-type compatibility is checked when the program is executing, not at the time of translation. Thus you will not find assignment-compatibility errors by choosing the option **Check** from the **Run** menu. Figure 4.5 shows a list of rules for all of the data types discussed in Chapter 2. The typical error message for assignment compatibility is "An incompatibility between types has been found." Understand that Pascal does not allow an `integer` variable to be assigned a `real` value. Any attempt to use such an assignment statement will produce a syntax error.

For an assignment statement of the form *variable* := *expression*, the value of the data type for *expression* is assignment-compatible with the data type of *variable* if any of the following is true:

1. *Variable* and *expression* are identically declared types.
2. *Variable* and *expression* are `real` types, and the value of *expression* is within the range of possible values for *variable.*
3. *Variable* is is a `real` type, and *expression* is an `integer` type.
4. *Variable* and *expression* are compatible `ordinal` types, with the value of *expression* being within the range of possible values for *variable.*
5. *Variable* is a `char` type and the value of *expression* is a `string` type having a length of 1.
6. *Variable* is a `string` type with a maximum size of *n* characters, and the value of *expression* is a `string` type having a length less than or equal to *n* .
7. *Variable* is a `string` type, and *expression* is a `char` type.

Figure 4.5 Assignment compatibility rules for Macintosh and THINK Pascal.

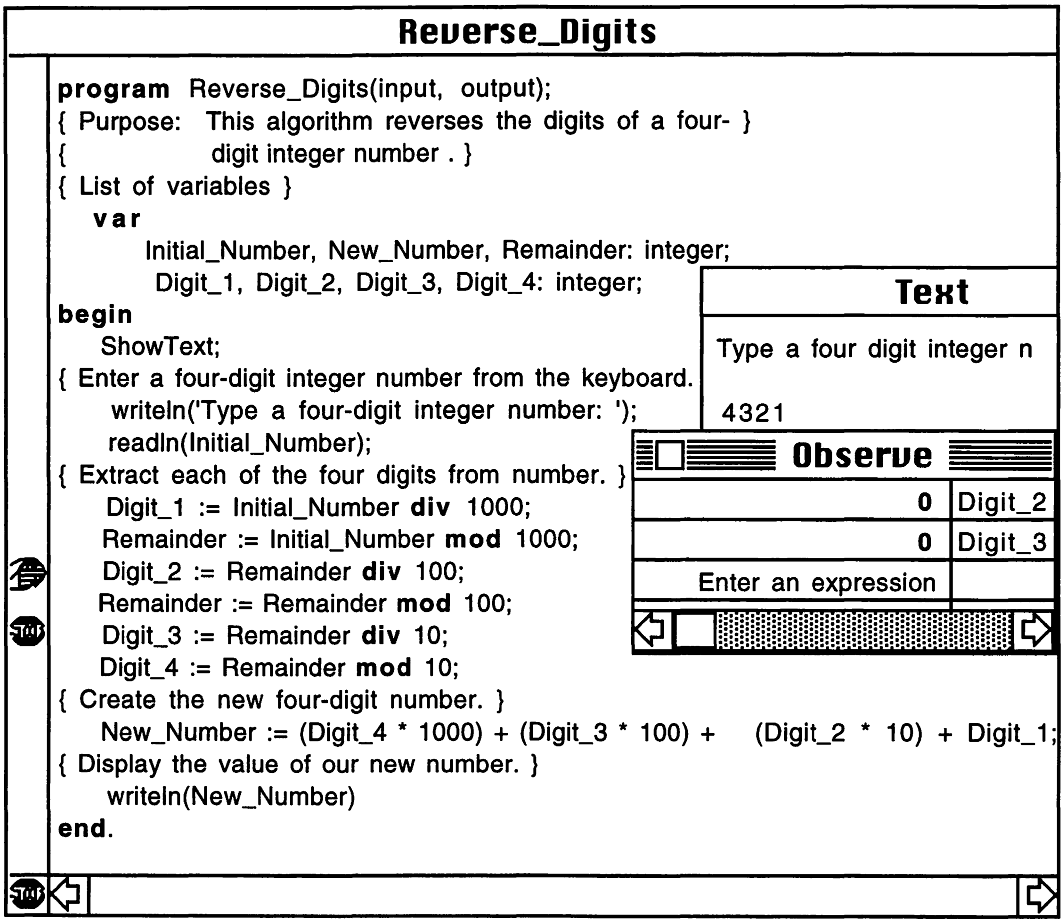
4.4 USING THE OBSERVE WINDOW TO TRACE A PROGRAM

The use of trace tables in testing an algorithm can be supplemented in Macintosh Pascal by use of the Observe window. Figure 4.6 shows our example of `Reversed_Digits`, which uses the Observe window to monitor the value of the variable `Remainder`. This window is primarily helpful in tracing values of intermediate variables and expressions. That is, variables such as `Remainder` contribute to generating the output of a program but are not themselves displayed as output. This is particularly important with respect to the values, variables, and expressions that affect the Drawing window.

To use the Observe window, open the Program window and select **Stops In** from the **Run** menu. Next, place a stop to the left of the program line after the last line you want to execute. In Figure 4.6 we placed a stop after the two statements for the variable `Remainder` by moving the cursor into the zone to the left of the program line, and then clicking the mouse button. After the stops are placed, open the Observe window from the **Windows** menu.[2] In Figure 4.6 we typed the word `Remainder` to the right of the prompt `Enter an expression`. After you press the Return key, the prompt `Enter an expression` appears, and the system waits for the next expression. When you have entered all expressions, open all windows necessary for observation. As Figure 4.7 shows, we only require that the Text window be opened for observing the prompt and for entering a four-digit `integer` number. Then select either the option **Go** or **Go-Go** from the **Run** menu. When the program reaches a stop sign, it updates the boxes to the left of the expressions in the Observe window. Figure 4.7 shows the value for `Remainder` after the program encounters the first stop sign. The program will continue to execute if

[2] The Observe window is located under the **Debug** menu in THINK Pascal.

Go-Go was selected. Otherwise, select **Go** or **Go-Go** to continue execution. By using the Observe window, you can see a real-time display of the intermediate values for expressions or variables that control output but are not normally visible as part of the output. This can be an important tool for debugging.



```
program Reverse_Digits(input, output);
{ Purpose:  This algorithm reverses the digits of a four- }
{             digit integer number . }
{ List of variables }
   var
      Initial_Number, New_Number, Remainder: integer;
       Digit_1, Digit_2, Digit_3, Digit_4: integer;
begin
   ShowText;
{ Enter a four-digit integer number from the keyboard.
    writeln('Type a four-digit integer number: ');
    readln(Initial_Number);
{ Extract each of the four digits from number. }
    Digit_1 := Initial_Number div 1000;
    Remainder := Initial_Number mod 1000;
    Digit_2 := Remainder div 100;
    Remainder := Remainder mod 100;
    Digit_3 := Remainder div 10;
    Digit_4 := Remainder mod 10;
{ Create the new four-digit number. }
    New_Number := (Digit_4 * 1000) + (Digit_3 * 100) +   (Digit_2 * 10) + Digit_1;
{ Display the value of our new number. }
    writeln(New_Number)
end.
```

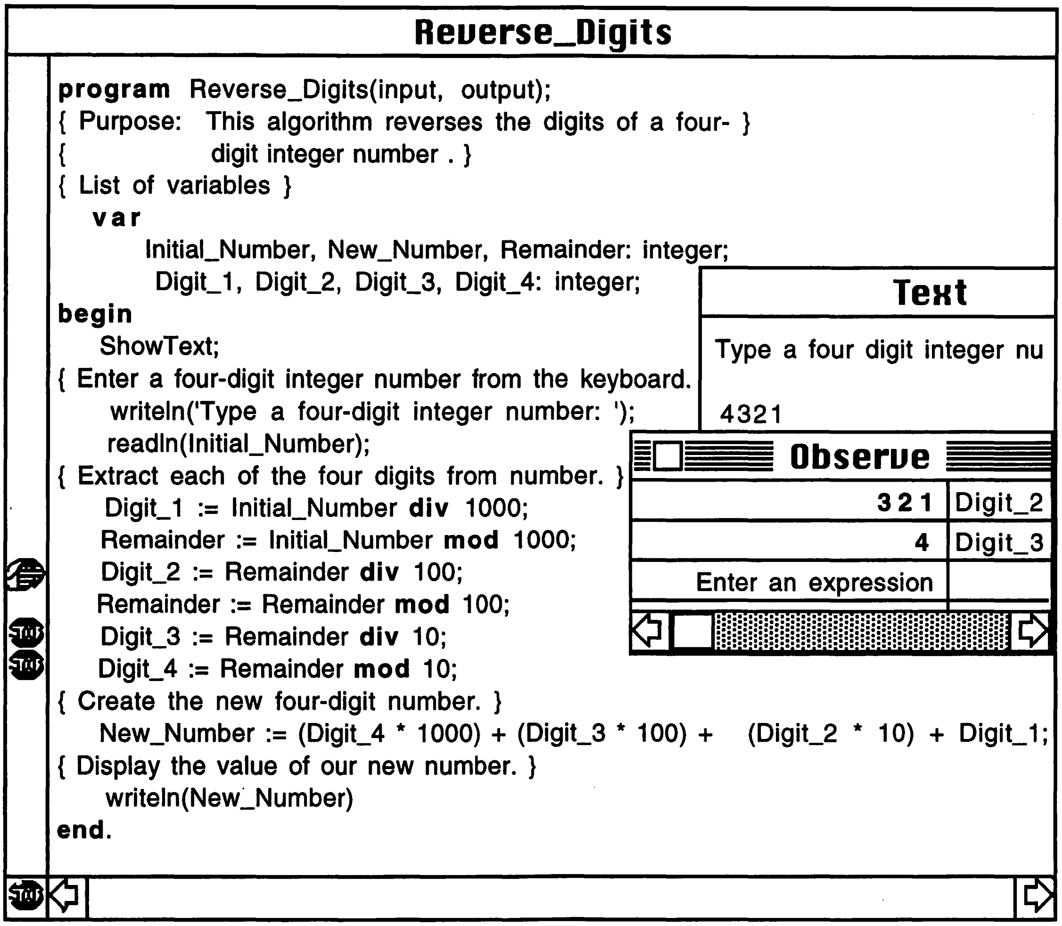
```
Type a four digit integer n

4321
```

```
                 0  Digit_2
                 0  Digit_3
```

Figure 4.6 Using the Observe window to trace a program.

4.5 ARITHMETIC FUNCTIONS

Macintosh Pascal provides a set of library functions for use with arithmetic expressions. Remember that a library is a collection of programs that a programmer can use to ease the work of writing a program. Figure 4.8 lists the arithmetic functions available in Macintosh and THINK Pascal. In relation to the precedence level of operators, functions are evaluated before any other part of the arithmetic expression.



```
Reverse_Digits
program Reverse_Digits(input, output);
{ Purpose: This algorithm reverses the digits of a four- }
{           digit integer number . }
{ List of variables }
  var
     Initial_Number, New_Number, Remainder: integer;
     Digit_1, Digit_2, Digit_3, Digit_4: integer;
begin
  ShowText;
{ Enter a four-digit integer number from the keyboard.
    writeln('Type a four-digit integer number: ');
    readln(Initial_Number);
{ Extract each of the four digits from number. }
    Digit_1 := Initial_Number div 1000;
    Remainder := Initial_Number mod 1000;
    Digit_2 := Remainder div 100;
    Remainder := Remainder mod 100;
    Digit_3 := Remainder div 10;
    Digit_4 := Remainder mod 10;
{ Create the new four-digit number. }
    New_Number := (Digit_4 * 1000) + (Digit_3 * 100) +  (Digit_2 * 10) + Digit_1;
{ Display the value of our new number. }
    writeln(New_Number)
end.
```

```
Text
Type a four digit integer nu

4321
```

```
Observe
                 321 | Digit_2
                   4 | Digit_3
 Enter an expression |
```

Figure 4.7 Executing the first stop in `Reverse_Digits`.

When using these standard arithmetic functions, one must be sensitive to data types. In some cases, a function requires an argument of a given type. For example, `odd(X)` requires an argument of the `integer` type, and the trigonometric functions and natural log functions may use either `real` or `integer` arguments. In addition, the data type of the value returned by the function is important. The `odd(X)` function returns a `Boolean` type, while the natural log function returns an `extended` type. The argument of a function is the portion of the expression that the function operates on, and it is enclosed in parentheses. In the statement

```
Y := sin(X);
```

`X` is the argument of the function `sin`. The value that this function returns is the sine of the angle `X`. For example, if `X` is 90°, then `Y` is assigned the value 1.

Function	Argument Data Type	Returns Data Type	Brief Description of Function
odd(X)	Integer	Boolean	Returns *true* if X is odd.
abs(X)	If X is integer If X is real	Longint Extended	Returns the absolute value of X.
sqr(X)	If X is integer If X is real	Longint Extended	Returns the square of X.
sqrt(X)	If X is integer If X is real	Longint Extended	Returns the square root of X.
ln(X)	Integer or real	Extended	Returns the natural logarithm of X.
exp(X)	Integer or real	Extended	Returns the exponential of X.
sin(X)[a]	Integer or real	Extended	Returns the sine of X.
cos(X)[a]	Integer or real	Extended	Returns the cosine of X.
arctan(X)	Integer or real	Extended	Returns the arctangent of X (in radians).
round(X)	Real	Longint	Converts a real value to a longint value (rounds to the nearest whole number).
trunc(X)	Real	Longint	Converts a real value to a longint value (rounds down to a whole number).
random	None	Integer	Returns a random integer in the range -32768 to 32767.

[a] X represents an angle measured in radians.

Figure 4.8 Library of arithmetic and related functions for Macintosh and THINK Pascal.

We can include a function as part of an expression, but it must have a specific argument. For example, using the Law of Cosines, we can compute the unknown length of side A of a triangle, given the other two sides, B and C , and the angle opposite the unknown side. The Law of Cosines is expressed as follows:

$$A^2 = B^2 + C^2 - 2BC\cos(X)$$

The side A is given by

$$A = \sqrt{(B^2 + C^2 - 2BC\cos(X))}$$

Figure 4.9 illustrates the Law of Cosines using a triangle with sides B = 2 and C = 4 and an angle of 45° opposite side A.

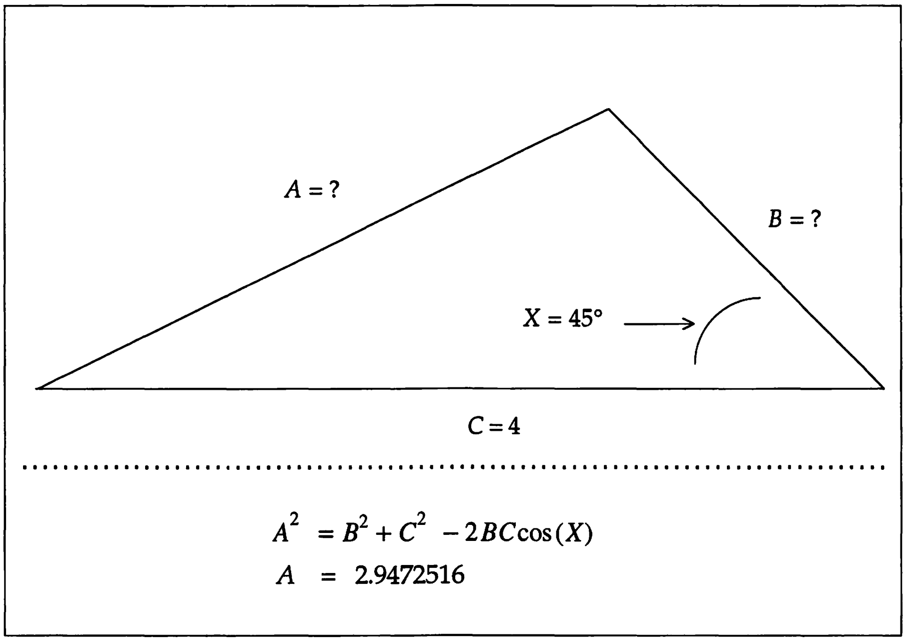


Figure 4.9 Using the Law of Cosines to determine the length of the unknown side of a triangle.

Translating the last expression into a Pascal statement yields

```
A := sqrt(sqr(B) + sqr(C) -2 * B * C * cos(X));
```

using three arithmetic functions from the Macintosh Pascal library: `sqrt`, `sqr`, and `cos`. When this expression is evaluated, the functions `sqr(B)`, `sqr(C)`, and `cos(X)` will be evaluated first, followed by multiplication between the `integer` constant 2 and the value of B, followed by multiplication of the resulting value of `2 * B` and the value of C, followed by multiplication between the resulting values of `2 * B * C` and `cos(X)`, followed by addition between the resulting values of `sqr(B)` and `sqr(C)`, followed by subtraction between the resulting values of `sqr(B) + sqr(C)` and `2 * B * C * cos(X)`, followed by the square-root operation.

The program `Triangle` uses this equation to compute an unknown side of a triangle, given two sides and an opposite angle. In the program `Triangle`, the lengths of the known sides are entered at the keyboard, and the opposite angle is provided as a constant. Since this angle is given in degrees, and the cosine function requires that the angle be in radians, we must convert the angle from degrees to radians before executing the cos function. This is accomplished with the statement

```
Z := X * (pi / 180.0);
```

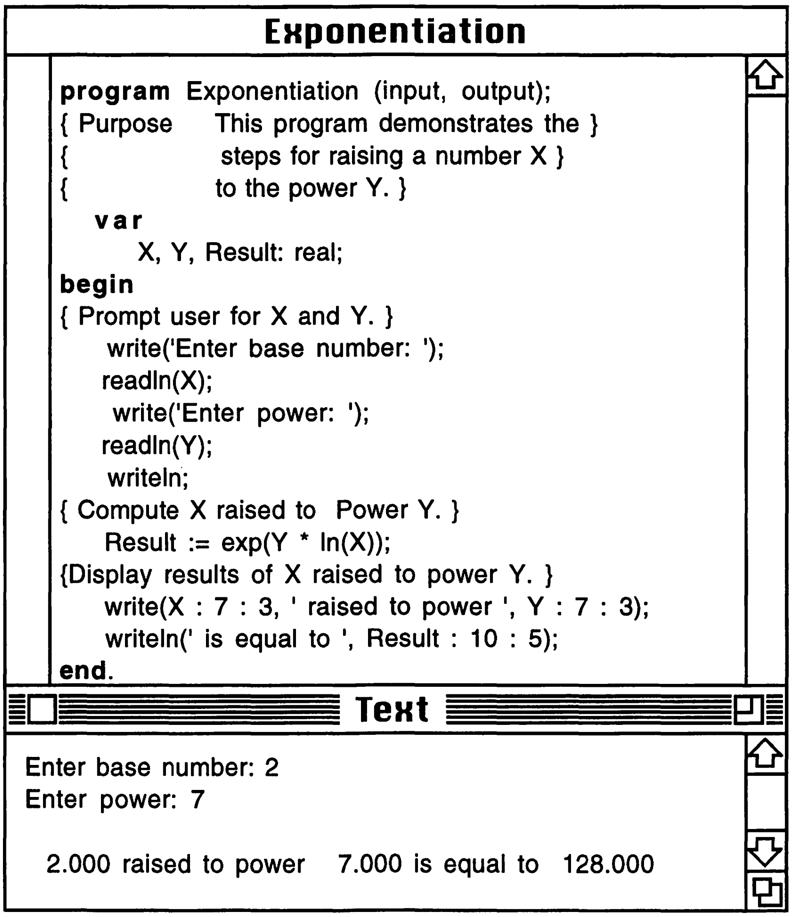
where X is the angle given in degrees, and pi is a constant representing the value of π. No declaration for pi is required, because it is represented internally in Macintosh Pascal as a real constant with the value 3.141592653... When executed, the program displays the angle, the lengths of the known sides, B and C , and the length of the unknown side, A .

```
program Triangle(input, output);
{ Purpose:  This program demonstrates the use of library }
{           arithmetic functions. Computes the unknown side }
{           of a triangle using the Law of Cosines. }
   const
{ The known angle opposite side A is assumed to be in degrees. }
      X = 45.0;
   var
{ The sides of the triangle and a temporary variable Temp. }
      A, B, C, Temp : Real;
begin
{ Prompt the user to enter the length of sides B and C. }
   writeln;
   write('Please enter a length for side B. ');
   readln(B);
   writeln;
   write('Please enter a length for side C. ');
   readln(C);
   writeln;
{ Convert the angle from degrees into radians. }
   Temp  := X * (pi / 180.0);
{ Compute the length of the unknown side called A. }
   A := sqrt( (sqr(B) + sqr(C) ) - (2 * B * C * cos(Temp) ) ) ;
{ Display the angle and the lengths of sides A, B, and C. }
   writeln('Angle X = ', X : 1 : 1, ' degrees');
   writeln;
   writeln('Side A = ', A : 1 : 2);
   writeln('Side B = ', B : 1 : 2);
   writeln('Side C = ', C : 1 : 2);
   writeln;
end.
```

The next example shows how two arithmetic functions, exp and ln, can be used to raise a number to a power. Most Pascal translators have no arithmetic operator for raising a value X to a power Y, when Y has a value other than 2. The expression X^Y can also be written as $e^{Y \ln(X)}$. In Pascal this is represented by the expression

```
exp(Y * ln(X)).
```

We can use the ln function as long as the value of X is greater than zero. Figure 4.10 shows a program for testing this expression. Notice that the result will always be real, since the functions ln and exp return values that are extended (real).



```
program Exponentiation (input, output);
{ Purpose    This program demonstrates the }
{             steps for raising a number X }
{            to the power Y. }
  var
     X, Y, Result: real;
begin
{ Prompt user for X and Y. }
    write('Enter base number: ');
   readln(X);
    write('Enter power: ');
   readln(Y);
    writeln;
{ Compute X raised to  Power Y. }
    Result := exp(Y * ln(X));
{Display results of X raised to power Y. }
    write(X : 7 : 3, ' raised to power ', Y : 7 : 3);
    writeln(' is equal to ', Result : 10 : 5);
end.
```

```
Enter base number: 2
Enter power: 7

  2.000 raised to power   7.000 is equal to  128.000
```

Figure 4.10 Ex cution of the program Exponentiation.

This programming trick is useful when approaching problems such as computing compound interest. For example, the program Compounded_Interest computes the accumulated value A, given the principal P, interest i, and period n, using the following equation:

$$A = P(1 + i)^n$$

```
program Compounded_Interest(input, output);
{ Purpose:  This program computes the compounded interest given }
{           principal, period, and rate of interest. }
  var
     Principal, Accumulated_Value : real;
```

```
      Interest, Period : integer;
begin
{ Enter principal, interest, and saving period. }
   write(' Enter principal: ');
   readln( Principal );
   write(' Enter interest rate: ');
   readln( Interest );
   write(' Enter years for compounding: ');
   readln( Period );
{ Compute accumulated savings. }
   Accumulated_Value := Principal * exp(Period * ln(1+Interest) );
{ Display accumulated savings. }
   writeln( ' Accumulated Savings: ', Accumulated_Value );
end.
```

The last program in this chapter, Rounding_Test, is designed to demonstrate three additional functions. The first is a random-number function that returns an integer in the range of –32768 to 32767. The other functions, trunc and round, convert a real number to a longint number by either truncating or rounding the fractional part of the number. In Rounding_Test we work with a real number called Selected_Number, which is produced by executing the following statement:

```
Selected_Number := random / 13;
```

To ensure that the selected number has a reasonable fractional part, the number generated by the function random is divided by the constant 13, using real division.

```
program Rounding_Test(input, output);
{ Purpose:  This program compares the round and trunc functions.}
   var
      Truncated_Number, Rounded_Number : longint;
      Selected_Number : real;
begin
{ Select a random number. }
   Selected_Number := random / 13;
{ Truncate and round the selected number. }
   Truncated_Number := trunc(Selected_Number);
   Rounded_Number := round(Selected_Number);
{ Display the number in original, truncated, and then }
{ rounded form. }
   writeln('The random number is     ', Selected_Number : 14 : 8);
   writeln('The truncated form is ', Truncated_Number);
   writeln('The rounded form is   ', Rounded_Number);
end.
```

4.6 STANDARD PASCAL VERSUS THINK PASCAL

While Standard Pascal supports scalar types such as integer, real, Boolean, and character, THINK and Macintosh Pascal have an added set of numeric types that includes the real types double (double-precision real), single (single-precision real, which with the Macintosh is equivalent to real), comp (short for *computational*), and extended. The type computational represents a real type where the exponent for e is always zero. The only extended integer type in THINK Pascal is longint, representing a double-precision integer. The range of values can vary among personal computer Pascal systems. In Turbo Pascal real uses 6 bytes for storage, while single precision requires only 4 bytes.

In either THINK or Macintosh Pascal, the basic arithmetic operations +, –, *, **div**, and **mod** will always result in a longint type if one of the operands is integer and the other is longint, or if both operands are longint. If one of the operands is a real type, the result will always be an extended type. With respect to the division operator /, the result is always extended. For the negation operation – or the identity operation +, if the operand is a longint, the result is a longint, and if the operand is any real type, the result is an extended type. This implies that programs compiled in Pascal may execute arithmetic operations in longint, an integer operand being converted to longint when necessary, or in extended mode if one or both operands are real type and extended arithmetic is required.

THINK Pascal allows the type of an expression and its value to be cast (coerced) into another type by execution of a type-casting expression. This is performed by applying an expression of the form

```
type( expression )
```

where type represents the data type to which the expression is to be converted, and the expression within the brackets is the operand that is being converted. The result of the type-casting expression is the value associated with the casting type. The following simple program demonstrates this concept. Here a loop is used to convert the value of a character into an integer, add 32 to the integer variable called Character_Value, and then convert this integer value back into a character type.

```
program Output_Letters (output);
{ Purpose:  This program demonstrates type casting.}
var
   Upper_Case_Letter: char;
   Lower_Case_Letter: char;
   Character_Value: integer;
begin
{ Open the text window for viewing output to the screen.}
   ShowText;
{ Display a header for uppercase and lowercase letters. }
   writeln('    Uppercase Letter        Lowercase Letter');
   writeln(' ----------------------------------------');
{ Convert capital letters A-Z to lowercase letters a-z }
   for Upper_Case_Letter := 'A' to 'Z' do
      begin
         Character_Value := integer(Upper_Case_Letter);
         Lower_Case_Letter := char(Character_Value + 32);
         writeln(Upper_Case_Letter : 12, Lower_Case_Letter : 23);
      end;
end.
```

Note that the ability to type-cast in Pascal is not as flexible as in the C language. While an `integer` value can be cast into a `Boolean` type, a `real` value cannot be cast into an `integer` type. If the type-casting is permitted for ordinal values, the conversion may either involve the extension or truncation of the original value if the internal storage size is changed. For nonordinal values, the internal representation of the expression being cast must be the same as the internal representation of the cast type, since the storage size of a nonordinal expression cannot be changed by a type-cast. Type-casting is not supported in Macintosh Pascal.

SUMMARY

This chapter has presented the principal elements necessary for writing arithmetic expressions in Macintosh and THINK Pascal. It introduced the concepts of operators and operands and the rules of operator precedence. The assignment of values to variables in Pascal expressions was discussed. Finally, we illustrated the use of library functions in a Macintosh or THINK Pascal expression, and presented a table of the basic mathematical functions.

REVIEW QUESTIONS

1. What is the purpose of the reserved words **`const`** and **`var`**?
2. Explain briefly the difference between a binary operator and a unary operator.
3. What is meant by the term *operand*?
4. What is the result of using the binary operator **`div`**?
5. What is the result of using the binary operator **`mod`**?
6. List the other binary arithmetic operators in Macintosh and THINK Pascal.
7. What happens in Pascal if you attempt to use `real` data types with the binary operators **`div`** and **`mod`**? Try this with both Macintosh and THINK Pascal.
8. Use the Instant window to see if there is any difference in the result from executing the two expressions 7 **`div`** 5 and `7 / 5`.
9. After reviewing the rules for computing `M` **`mod`** `N`, use the Instant window to show an example for each case.
10. Is the assignment operator := a binary operator?
11. What is meant by the term *operator precedence*?
12. For the arithmetic operators in Pascal, list the operator precedence from highest to lowest. What operators have equal operator precedence?
13. How do parentheses change the order of operator precedence?
14. Are parentheses considered operators?
15. Which of the following expressions have correct syntax?

(a) $-+(A*-B)$ (b) $+(A-(B-C))$

(c) $-(--A+B)$ (d) $-(A+(B-(C-D)))$

16. Why is the statement $X = X + 1$ illogical in algebra but acceptable as an assignment statement, `X := X + 1`, in Pascal?
17. Why is the symbol = used for constant declarations instead of := ?
18. If we had only the **`div`** operator and the **`mod`** operator did not exist, how would you write the proper assignment statements and expressions to compute the quotient and remainder for an `integer` number `M` divided by another `integer` number `N` ?
19. What is the difference between the Observe window and the Instant window?
20. What is meant by the term *program library*?
21. What is meant by the term *function*?
22. Without using any of the Pascal arithmetic functions, write valid Macintosh and THINK Pascal expressions for the following algebraic expressions:

    (a) $\dfrac{A+B^2}{C-4D}$

    (b) $AX^4 + BX^3 + CX^2 + DX + 1$

    (c) $\dfrac{base * height}{2}$

    (d) $\{[(EX+F)X+G]X+H\}X+1$

23. Write valid assignment statements for the following algebraic identities:

    (a) $force = mass * acceleration$

    (b) $force = G\dfrac{m * m'}{r^2}$

    (c) $wavelength = \dfrac{h}{mass * velocity}$

24. List all the arithmetic functions in Macintosh Pascal.
25. Write Pascal assignment statements, using the arithmetic function `sqrt` for the following algebraic identities:

    (a) $frequency = \dfrac{1}{2\pi}\sqrt{k/mass}$

    (b) $velocity = \sqrt{g * radius}$

    (c) $time = 2\pi\sqrt{(L\cos(\theta)/g)}$

(d) $velocity = \sqrt{(V_x{}^2 + V_y{}^2)}$

whereV_x is the x velocity and V_y is the y velocity

26. Write an assignment statement for converting angle `A` in degrees into angle `R` in radians.
27. Write an assignment statement for converting angle `R` in radians into angle `A` in degrees.
28. How can the exponentiation operation be performed in Pascal?
29. What special conditions must you consider when using the function `ln`?
30. Using the `ln` and `exp` functions, write assignment statements for the following two equations:

(a) $volume = \frac{4}{3}\pi r^3$

(b) $radius = \left(\frac{3}{4}\frac{volume}{\pi}\right)^{1/3}$

31. Determine values for the following functions:

(a)	`round`(−3.6)	(b) `round`(0.987)	
(c)	`round`(0.4498)	(d) `trunc`(−3.6)	
(e)	`trunc`(0.987)	(f) `round`(0.4498)	
(g)	`trunc`(54.567)	(h) `round`(54.567)	
(i)	`trunc`(pi)	(j) `round`(pi)	
(k)	1 + `trunc`(pi ∗ 100/ 3.0)		
(l)	1 + `round`(pi ∗ 100/ 3.0)		

32. What is the range of values for the expression `abs(random)`?
33. What is the range of values for the expression `- abs(random)`?
34. Write an expression using the function `random` and the operator **mod** for computing a random number between 0 and 100.
35. Write an expression for computing a random number between −1 and −1000.
36. What are the values for the following expressions?

(a) `odd`(−56) (b) `odd`(56) (c) `odd` (99)
(d) `odd`(0) (e) `odd`(2)
(f) `odd`(1 + `trunc`(pi ∗ 100/ 3.0))

37. Using the functions `sin` and `cos`, write an assignment statement for computing the tangent of an angle.
38. If *A* is declared as `longint` and `B` is declared as `integer`, are the following assignment statements valid?

(a) A := B; (b) B := A;

39. If C is declared `real` and D is declared `integer`, are the following assignment statements valid?

    (a) C := D; (b) D := C;

40. What is the order of operations when the following Pascal expressions are evaluated?

    (a) $(A+B)/(A/D-E)+G$

    (b) $(U/D+F)/4.0-H$

    (c) $A+B/A/D-E+G$

41. Use the `sqr` function to write Pascal expressions for the following arithmetic expressions.

    (a) $(A+B^2)^3$ (b) $(X^2+Y^2+Z^2)^4$

    (c) $(A+B)^{2^{3^4}}$ (d) $(W^2+2WU+U^2)^5$

    (e) $AX^4+BX^3+CX^2+DX+1$

    (f) $[A^2(B^2+C^2-A^2)+B^2(A^2-B^2+C^2)+C^2(A^2+B^2-C^2)]$

42. Using the Instant window, show that `ord(succ(X))` is equal to `ord(X) + 1` for some value of `X` (`X` can be `enumerated`, `integer`, or `char`).
43. Using the Instant window, show that `ord(pred(X))` is equal to `ord(X) - 1` for some enumerated value of `X`.
44. Given that `pi` is a `real` constant recognized by Macintosh and THINK Pascal, what is the difference between the execution of the statement `writeln(pi : 15)` and `writeln(pi:15:13)`?
45. What happens if Macintosh or THINK Pascal attempts to execute the `sqrt` function with an argument that is negative? Use the Instant window to show the result.
46. If `arctan(X)` represents angle `Y`, how can the `arctan` function be used to compute the value of `arcsin(X)`?
47. If `arctan(X)` represents angle `Y`, how can the `arctan` function be used to compute the value of `arccos(X)`?
48. If the tangent of 45° is 1, use the Instant window to check if the value of `tan(arctan(1))` represents an angle of 45°. If `arctan(1)` represents an angle of 45°, use the Instant window to verify that `arctan(tan( X ))` is equivalent to an angle of $\pi/4$ radians.
49. Assuming that a Pascal expression can only be written using the binary operators * and + , convert the following expressions into Pascal code, and count the number of multiplications required by each expression.

Which Pascal expression requires more multiplications? How many additions are required by each Pascal expression?

(a) $AX^4 + BX^3 + CX^2 + DX + 1$

(b) $\{[(AX + B)X + C]X + D\}X + 1$

PROGRAMMING EXERCISES

The following exercises require a basic knowledge of high school algebra and trigonometry. Since programming sometimes requires the execution of algebraic expressions and trigonometric functions, the following exercises are introduced to encourage the writing of short programs as well as the solution of simple problems.

1. Write a Pascal program that will perform the following steps:

   (a) Prompt for the volume of a cylinder, and enter a value.
   (b) Prompt for the height of a cylinder, and enter a value.
   (c) From the formula

$$\texttt{Volume\_Cylinder} = 2 * \pi * \texttt{Radius}^2 * \texttt{Height},$$

   compute the required radius using the function `sqrt`.

   (d) Display the following values to the Text window: volume of the cylinder, height of the cylinder, and radius of the cylinder.

2. The program `Triangle` can be modified in several ways to explore the Law of Cosines and to apply functions in expressions. Modify the program `Triangle` so that the user can change the angle `X` by entering the desired angle from the keyboard.

3. Write a Pascal program that will perform the following steps:

   (a) Prompt for the volume of a cone, and enter a value.
   (b) Prompt for the height of a cone, and enter a value.
   (c) From the formula

$$\texttt{Volume\_Cone} = \pi * \texttt{Diameter}^2 * \texttt{Height} / 12,$$

   compute the required radius using the function `sqrt`.
   (d) Display the following values to the Text window: volume of the cone, height of the cone, and radius of the cone.

4. Rewrite the program `Conversion` for converting a Fahrenheit temperature to Celsius.

5. Write a program that will prompt the user for angle `X`, compute the tangent of `X` using the functions `sin` and `cos`, and display the result with appropriate headings to the Text window.

6. Write a program that will take an `integer` number from input and report whether the number is odd or even.

7. Write a program that will take a `real` number from input represented by the variable named `X`, and compute the arctangent, arcsine, and arccosine for `X`. After computing all three values in terms of degrees rather than radians, display to the Text window the results with appropriate headings.

8. Assume that it is necessary to compute the value X^Y, where `X` is greater than zero. Since Pascal has no operator for performing exponentiation, rewrite the expression X^Y as $e^{Y \ln(X)}$. Write a Pascal program that will prompt for the values of `X` and `Y`, and use the functions `ln` and `exp` to compute `X` raised to the power `Y`. Assume that `X` and `Y` are `real`, and that appropriate headings are to be displayed.

9. Write a program that will select a random number greater than or equal to 1 and then display this number and the natural logarithm of the number. Include appropriate headings in the display.

10. The common logarithm of a number can be given by the relationship

$$\log_{10} m = \frac{\ln m}{\ln 10} \text{ where m > 0.}$$

Write a program that prompts for the value of *m*, computes the common logarithm of *m* using the function `ln`, and displays the result using appropriate headings.

11. The following approach can be used to compute the area of a triangle when only the lengths of the sides are known. Write a Pascal program necessary to evaluate the following equation:

$$\frac{[A^2(B^2+C^2-A^2)+B^2(A^2-B^2+C^2)+C^2(A^2+B^2-C^2)]^{\frac{1}{2}}}{4}$$

as the area of a triangle, where *A* , *B* , and *C* are the lengths of the sides of the triangle.

12. The function `chr(X)` takes the `integer` value of `X` and returns the ASCII character. In Macintosh Pascal the `integer` value used in memory to represent the ASCII value of `A` is 65. The lowercase letter `a` is represented by the `integer` value of 97. Each lowercase letter of the alphabet is given by the relationship `ord(Capital_Letter) + 32`. Write a Pascal program that prompts for a capital letter `A`

through Z and after the letter has been entered displays the following two messages along with the appropriate letters: `Uppercase letter:` and `Lowercase letter:`. This program will require the use of functions `chr` and `ord`.

13. Enter the following program. (The **while-do** loop is discussed in Chapter 5.) The program computes the predecessor and successor values of `Number`, using the Pascal library functions `pred` and `succ`. As you may have noticed, it has no `write` or `writeln` statements for displaying values. Insert stops at the proper points, and use the Observe window to monitor the values of `Number`, `Predecessor`, and `Successor` as the program is executed.

```
program Pred_Succ(input, output);
{ Purpose:  Compute the predecessor and successor values of }
{           Number. }
  var
    Predecessor, Successor, Number : integer;
begin
   Number := 0;
   while true do
      begin
      { Compute the predecessor and successor of Number. }
         Number := Number + 1;
         Predecessor := pred(Number);
         Successor := succ(Number);
      { Display the values of Predecessor, Number, and Successor }
      { in the Observe window rather than the Text window. }
      end;
end.
```

The exercises that follow require you to write an algorithm and establish a trace table for validating your algorithm. After you have eliminated all possible errors from your algorithm, convert your algorithm into a Pascal program and execute it. If any errors occur, be sure your algorithm is correct before making any changes to your program.

14. (a) Write an algorithm that accepts from input five temperatures measured at 6 a.m., 9 a.m., 12 n., 3 p.m., and 6 p.m. and then computes the mean of these five temperatures as the average daytime temperature. The algorithm will then output mean daytime temperature and each of the five daytime temperatures.
    (b) Trace the algorithm with several examples.
    (c) Write a Pascal program for this algorithm, displaying all information to the Text window.

15. (a) Write an algorithm that will take as input two values: the radius of a circle and an angle representing a counterclockwise rotation in degrees from the x axis shown in Figure 4.11, and compute the point (x, y).

(b) Trace the algorithm with several examples.
(c) Write a Pascal program to display the point (x , y).

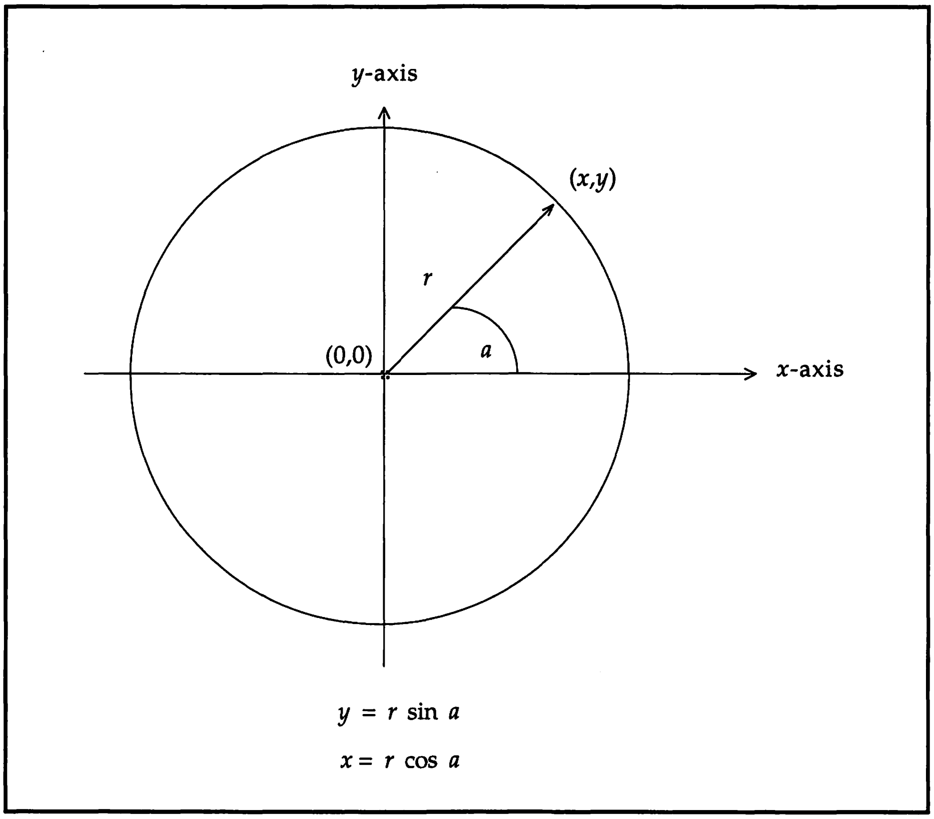


Figure 4.11 Defining a point on the circumference of a circle in relation to the radius and angle of rotation.

16. (a) Using the **mod** function, write an algorithm that prompts the user for a seven-digit (`longint`) number, then separates each of the seven digits, computes the mean of the seven digits, the product of the seven digits, and a seven-digit (`longint`) number with all of the digits reversed. The algorithm then displays the original number, the mean as a `real` value, the product as a `longint` value, and the number with digits reversed, using appropriate headings.
(b) Once you have completed the algorithm, construct a trace table, and trace your algorithm for three different numbers.
(c) Convert your algorithm into a Macintosh Pascal program and test your program using the input values from part (b).
(d) Use the Observe window to trace the values of variables in your Pascal program.

17. (a) Using `char` data types instead of `integer` data types, write an algorithm that prompts the user for a seven-digit `integer` number, then reads each of the seven digits by assigning each digit as a character to a variable. The algorithm is then to display the original number and the number with all digits reversed, using appropriate headings.
    (b) Once you have completed the algorithm, construct a trace table, and trace your algorithm for three different numbers.
    (c) Convert your algorithm into a Macintosh Pascal program, and test your program using the input values from part (b).
    (d) Use the Observe window to trace the values of variables in your Pascal program.

18. (a) Write an algorithm that will convert a speed given in terms of miles per hour into feet per second. Have your algorithm also provide a list of distances traveled in feet over the following time frames: 1 second, 5 seconds, 10 seconds, 15 seconds, 30 seconds, and 1 minute.
    (b) Once you have completed the algorithm, construct a trace table, and trace your algorithm for three different speeds.
    (c) Convert your algorithm into a Macintosh Pascal program, and test your program using the input values from part (b).
    (d) Use the Observe window to trace the values of variables in your Pascal program.

19. (a) Write an algorithm that will take the speed (velocity) of a moving car and compute the distance traveled in 15 minutes, 30 minutes, 45 minutes, 1 hour, 1 hour 30 minutes, and 2 hours. Assume that the velocity is represented in terms of miles per hour.
    (b) Once you have completed the algorithm, construct a trace table, and trace your algorithm for three different speeds.
    (c) Convert your algorithm into a Macintosh Pascal program, and test your program using the input values from part (b).
    (d) Use the Observe window to trace the values of variables in your Pascal program.

20. The common logarithm $\log_{10} X$ can be approximated by using the equation:

$$\log_{10} X = a_1 t + a_3 t^3 + a_5 t^5 + a_7 t^7 + a_9 t^9$$

where $t = (X - 1)/(X + 1)$, and X must be greater than 0.3162277 and less than 3.162277. Write an algorithm that will prompt for a value X within the range just specified; then, using both the approximation and the method from Exercise 10, show that the absolute difference between these values is less than or equal to 10^{-7}. The following are values for constants a_1, a_3, a_5, a_7, and a_9 :

$a_1 = 0.868591718$	$a_7 = 0.094376476$
$a_3 = 0.289335524$	$a_9 = 0.191337714$
$a_5 = 0.177522071$	

Translate your algorithm into a Pascal program, and execute the program to verify the statements.

Chapter 5

Basic Control Instructions for Looping and Branching

OBJECTIVES

After completing Chapter 5, you will know the following:

1. The Pascal pretest iteration loop command `while-do`.
2. The Pascal post-test iteration loop `repeat-until`.
3. The Pascal conditional expressions.
4. Branching in Pascal; the use of the one-way selector, `if-then`; the two-way selector, `if-then-else`; and the multiway selector, the `case` statement.
5. Nesting of loops by placing loops within loops in a Pascal program.
6. The use of `Boolean` operators to create compound conditions for use in branching statements.
7. The use of the `for` statement as an iteration statement controlled by a simple counter.

5.1 PROBLEM ANALYSIS AND TRACING

Before developing an algorithm and converting it into a computer program, one should always analyze the problem. The following briefly reviews the steps for this analysis:

1. Provide a precise definition of the problem.
2. Determine whether the problem has already been solved.
3. List all of the information required as input.
4. List all of the information required as output.
5. Begin with an initial set of steps as a first approximation of a solution.

6. Refine the steps, so they are precise and explicit.
7. Trace each step of the solution with known information.

The result of this process is an *algorithm* defining a finite number of unambiguous steps to the solution of a problem. Once the algorithm has been translated into a specific Pascal program, it should always be tested to determine if it provides correct output. As you learned in Chapter 4, specific output from a program can be checked relative to specific input(s) through the use of a trace table, an Observe window, and the inclusion of `write` statements. Every Pascal program should produce correct output for proper input data. The algorithm and its equivalent Pascal program should also be able to identify and reject improper input data if it occurs.

This chapter introduces the basic principles of programming with loops and branches, the Pascal commands for implementing these principles, and how they are applied to developing structured algorithms and programs. We emphasize the **`repeat-until`** and **`while-do`** commands because they require thinking in terms of the properties of loops. The branching constructs **`if-then`**, **`if-then-else`**, and **`case`** are examined during a discussion of one-way, two-way, and multiway selection. In relation to these loop and selection processes, conditional expressions are introduced and represented through relational and Boolean expressions. The **`for`** command, an additional loop statement, is presented as a command for controlling loops where simple counters are required. We compare various loop commands in later sections of the chapter.

5.2 CONTROL STRUCTURES FOR LOOPS

A loop represents a segment of an algorithm or program designed to be executed repeatedly. It reduces the length of an algorithm or program by allowing statements to be reused rather than rewriting them several times. A loop begins with the execution of an initial statement followed by other executable statements, and eventually returns to the initial statement. Every loop must be capable of terminating either before execution or after execution has been completed. Termination of a loop depends on a specified condition or *test*.

Applications of loops depend on the specific problem being solved and how the algorithm is to be implemented. Loops have one important purpose: they iterate the execution of a statement or a group of statements. Without the loop, a programmer would have to write repetitive code—an inefficient use of time and energy.

There are three basic types of loops: pretest iteration loops (**`while-do`**), post-test iteration loops (**`repeat-until`**), and in-test iteration loops. These types differ only with respect to where a test is applied to terminate execution of the loop. In this chapter we examine only the pretest and post-test iteration loops, because Pascal does not support any control construct for an in-test iteration loop. The syntax that follows describes pretest and post-test iteration loops:

```
while  condition  do                 repeat
{ body of pretest iteration          { body of post-test
   loop }                               iteration loop }
                                     until  condition
```

Flow diagrams providing semantic meaning for both control constructs are shown in Figure 5.1.

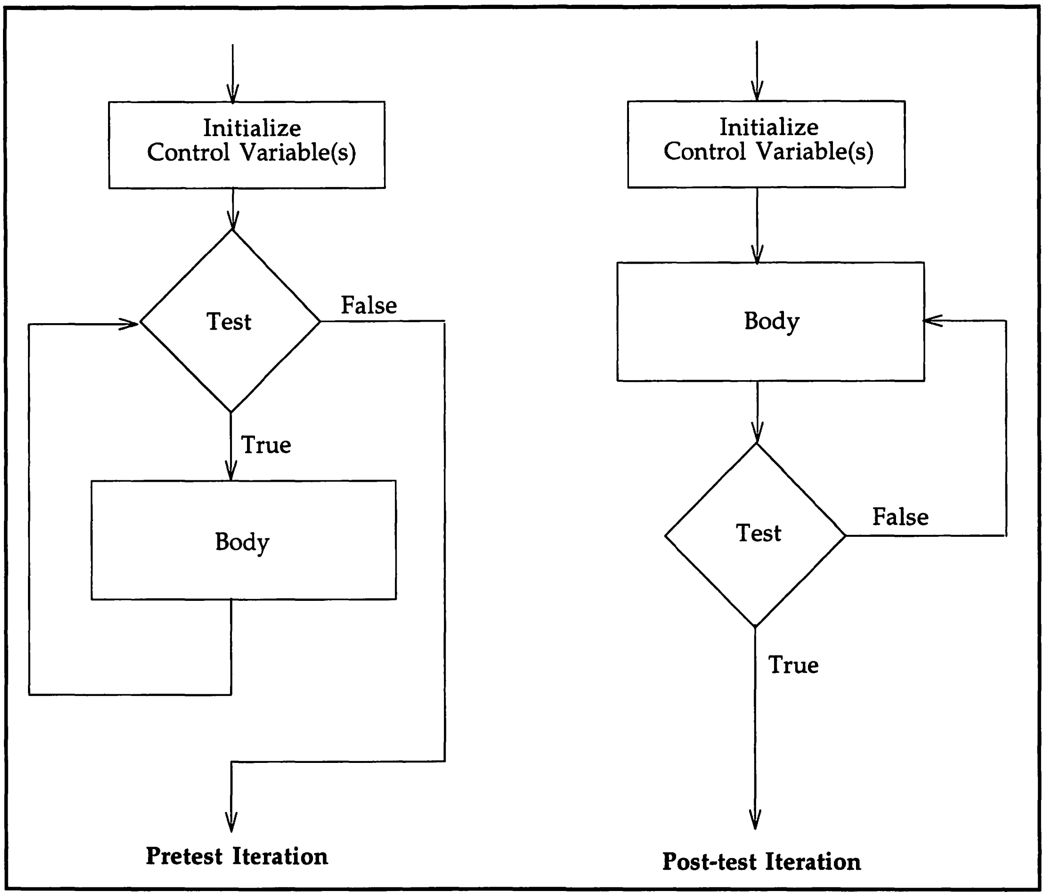


Figure 5.1 Flow diagrams for the pretest iteration (`while-do`) and post-test iteration (`repeat-until`) loops.

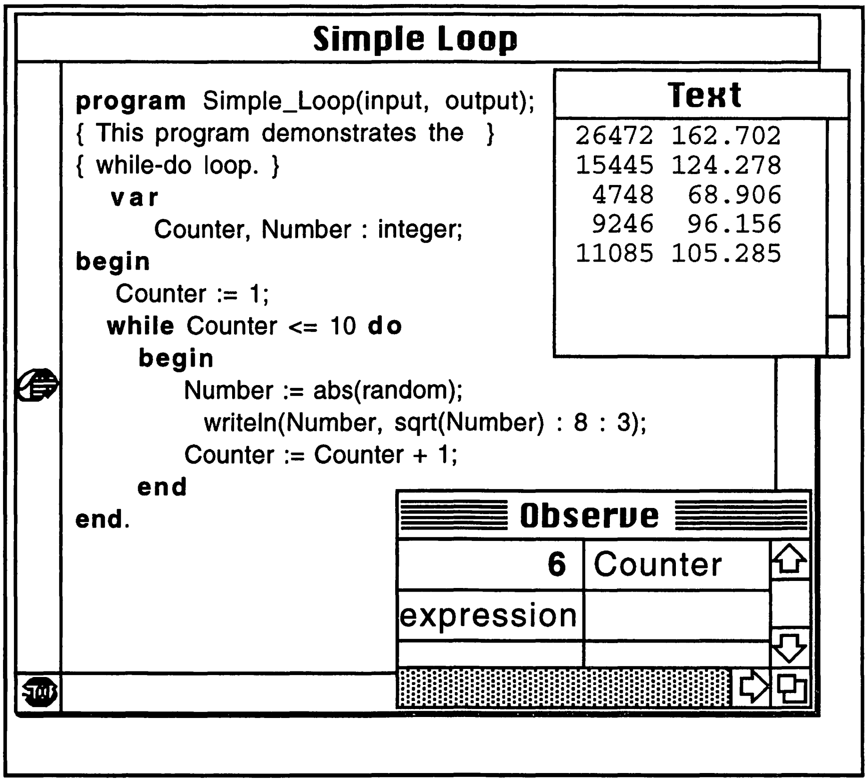
5.2.1 Pretest Iteration Loops

In pretest iteration, the condition is tested first, and if the condition is *true*, the body of the loop is executed. On returning to the beginning of the pretest loop, the condition is again tested, and if it is *true*, the body of the loop is executed again; if the condition is *false*, execution of the loop is terminated.

In Pascal the pretest iteration loop is represented by the following syntax:

```
while condition do
   statement ;
```

Often referred to as the **`while-do`** control construct, both the words **`while`** and **`do`** are reserved and must be used only within the context of writing a loop. For example, consider the program titled `Simple_Loop` in Figure 5.2. As you can see, this program begins execution by assigning an initial value of 1 to a variable called `Counter`. When the **while-do** statement is executed, the condition `Counter <= 10` (`Counter` less than or equal to 10) is tested. Because the value of `Counter` is initially 1, the test is *true* and the body of the loop is executed.



```
program Simple_Loop(input, output);
{ This program demonstrates the }
{ while-do loop. }
   var
      Counter, Number : integer;
begin
   Counter := 1;
   while Counter <= 10 do
     begin
        Number := abs(random);
          writeln(Number, sqrt(Number) : 8 : 3);
        Counter := Counter + 1;
     end
end.
```

```
26472 162.702
15445 124.278
 4748  68.906
 9246  96.156
11085 105.285
```

6	Counter
expression	

Figure 5.2 A simple example of a **while-do** loop.

In this example the body of the loop contains three statements. The first chooses a random number using the library function `random`. The library function `abs` forces the value of the random number to be positive. The second statement displays the value of the random number and the square root using the library function `sqrt`. Because the variable `Counter` controls the number of times that the body is executed, its value is modified by execution of the assignment statement

```
Counter := Counter + 1;
```

By using the Observe window and a strategically inserted stop, you can trace the values of the control variable as the loop is executed. Without the presence of the assignment statement for incrementing the value of the `Counter` by 1, the value for `Counter` would remain unchanged, and the value of the condition `Condition <= 10` would always be *true* . This would result in the loop continuing to execute forever, unless you choose the option **Pause** from the menu bar (or, in the case of THINK Pascal, you click the spray can).

You should also understand that the body of the **while-do** loop may never be executed if the condition is *false* when first evaluated. For instance, if we had typed

```
Counter := 11;
```

nothing would be displayed in the Text window, because the condition `Counter <= 10` would be *false*, and the loop would not execute.

As a second example, let us develop an algorithm where a set of positive numbers is to be entered from the keyboard and summed. When a negative value is entered, the algorithm stops summing and computes the average value by dividing the sum of the positive numbers entered by a total count. Only one input and one output are required: the next number to be read from the keyboard and the average value of all positive numbers, respectively. Here are the steps necessary for computing the sum and average:

```
Algorithm Average_Input;
begin
{ Initialize two variables: Partial_Sum and Total_Count. }
            Partial_Sum <-- 0.0;
            Total_Count <-- 0;
{ Prompt the user to enter the first number from the keyboard. }
            write ('Enter your first number: ');
            read ( Number );
{ Continue to enter numbers and compute partial sums while the
  value of Number is positive. }
      while { Number is positive } do
         begin
            Partial_Sum <-- Partial_Sum + Number;
            Total_Count <-- Total_Count + 1;
         { Enter next number from the keyboard. }
            write ( 'Enter your next number: ');
            read (Number);
         end;
{ Compute the average value. }
                  Average <-- Partial_Sum /  Total_Count;
{ Display the average value. }
                  write  (' Average value: ',  Average)
end.
```

In this algorithm, `Total_Count` is an `integer` type, and the remaining variables are `real`. Figure 5.3 shows a trace table for the algorithm `Average_Input`. Each step represents the execution of either an assignment statement or a test for the condition `Number is positive`. Placing an explicit column in the table for any condition helps us to understand when a condition succeeds or fails as the algorithm is being traced. The next program shows the Pascal code for executing this particular algorithm.

```
program Average_Input(input, output);
{ Purpose:  This program computes the average value of positive }
{           numbers entered from the keyboard. }
   var
      Number,  Average,  Partial_Sum :  real;
      Total_Count :  integer;
```

```
begin
   ShowText;
{ Initialize two special control variables: Partial_Sum and }
{ Total_Count. }
   Partial_Sum := 0.0;
   Total_Count := 0;
{ Prompt the user for entering the first number from the }
{ keyboard. }
   write( 'Enter your first number: ');
   readln( Number );
{ Continue to enter numbers while they are positive. }
   while  Number >= 0  do
      begin
         Partial_Sum := Partial_Sum + Number;
         Total_Count := Total_Count + 1;
         write( 'Enter your next number: ' );
         readln( Number );
      end;
{ Compute the average value. }
    Average  := Partial_Sum/ Total_Count;
{ Display the average value. }
   writeln  (' Average value: ',  Average:10:3 )
end.
```

Step	Number	Test If Number Is Positive	Partial_Sum	Total_Count	Average
1			0.0		
2				0	
3	4.2				
4		True			
5			4.2		
6				1	
7	3.8				
8		True			
9			8.0		
10				2	
11	5.2				
12		True			
13			13.2		
14				3	
15	-5				
16		False			
17					4.4

Figure 5.3 Trace table for the algorithm `Average_Input`.

In Pascal the body of a **`while-do`** loop can be a simple assignment statement, a `read` statement, a `write` statement, another **`while-do`** statement, a **`repeat-until`** statement, a branching statement, or a compound statement. If the body of a **`while-do`** loop is to have several statements, the following syntax is necessary:

```
while  condition  do
   begin
      statement1;
      statement2;
          :
      statementn
   end;
```

Any statements nested within a pair of **begin-end** brackets are referred to as a *compound statement.* Note that in Pascal a semicolon *does not have to precede an* **end**. Without the **begin-end** brackets to make a compound statement, only the first statement in a group of statements is the body of the loop. For example, if we had written the following code for the loop in Average_Input,

```
while  Number >= 0  do
   Partial_Sum := Partial_Sum + Number;
Total_Count := Total_Count + 1;
write( 'Enter your next number: ' );
readln( Number );
```

only the first statement, Partial_Sum:= Partial_Sum + Number, would be executed as the body of the loop. The remaining statements would not be executed, because a new value for Number could not be entered, and the condition Number >= 0 (Number greater than or equal to zero) would always be *true*, because the initial value would never change. The best advice when writing the body of a **while-do** loop is always to use a compound statement. This helps to avoid any confusion when adding or deleting statements from the body of a loop.

5.2.2 Post-test Iteration Loops

In the post-test iteration loop, the body of the loop is executed before the condition is tested. At the end of the loop, if the condition is *false*, execution of the body of the loop is repeated. In Pascal the **repeat-until** loop has the following syntax:

```
repeat
   statement1;
   statement2;
       :
   statementn;
until condition;
```

Notice that a compound statement is not required when the body of the loop has several statements in sequence. A compound statement does not change the semantic meaning of the program during execution. In the program Average_Input, the following **repeat-until** loop can be used in place of the **while-do** loop:

```
repeat
```

```
   Partial_Sum := Partial_Sum + Number;
   Total_Count := Total_Count + 1;
   write( 'Enter your next number: ' );
   readln( Number );
until  Number < 0 ;
```

As you can see, the primary difference between **while-do** and **repeat-until** is in the position of the condition. In the **repeat-until** loop, the body of the loop is executed at least once before the condition is tested. In the **while-do** loop, the condition is tested first, and if it is initially *false*, the body of the loop will not be executed.

When writing loops, it is important that you remember the following basic rules:

1. What represents the body of a loop? What are the steps that require repeated execution when the loop is employed? If you cannot answer these questions, further problem analysis is required.
2. What test or condition will allow the loop to continue execution? Where is the test to be performed? You must understand the condition for terminating loop execution and where the test is to be positioned. In some instances we may want the body of the loop to be executed at least once; in other cases we may not want the body of the loop to be executed at all if a condition is not met.
3. Every loop requires at least one variable for control. In some cases several variables can control the number of times the body of a loop is executed. In our example `Average_Input`, there is only one control variable, named `Number`. As long as the value of `Number` remains positive, the body of the loop is executed.
4. In the body of the loop, one or more statements can modify the control variable. In our example, `Simple_Loop`, the control variable `Counter` is modified by an assignment statement in which the value of `Counter` is incremented by 1. In `Average_Input`, the `read` statement modifies the value of the control variable `Number`, allowing the user to enter a new value from the keyboard. Entry of a negative value terminates the execution of the loop. As you can see from `Average_Input`, not all loops are controlled by counters.
5. The control variable may be required to have an initial value before entering the body of a loop. In `Simple_Loop`, this step is performed by an assignment statement. In `Average_Input`, an initial value for `Number` is necessary before the partial sum can be computed.

When writing loops, keep these rules in mind. Failure to observe them can lead to an unsuccessful program.

5.3 CONDITIONAL EXPRESSIONS

The conditions or tests in Pascal are often referred to as *conditional expressions*. When evaluated, a conditional expression will have either the `Boolean` value *true* or the `Boolean` value *false*. Conditional expressions, like the arithmetic expressions discussed in Chapter 4, have specific operators and operands. Two sets of operators can be used in conditional expressions: relational and `Boolean` (logical).

In Pascal a relational expression is represented by the following syntax:

```
operand   relational operator   operand
```

The values of the left and right operands must be type-compatible. If they are not, a syntax error results in THINK Pascal, and an execution error in Macintosh Pascal.

Figure 5.4 shows the relational operators, their meanings, and the required operand types. The result of a relational expression is always `Boolean`; that is, a relational expression will always evaluate to a `Boolean` value that is either *true* or *false*. In relation to the arithmetic operators, relational operators have the lowest operator precedence and are executed last. This means that operands that are arithmetic expressions will be evaluated before the relationship is tested. Keep in mind that a relational expression can only have a single *relational* operator. For example, the expression

```
A + B <= C - G / H
```

is syntactically correct, whereas the expression

```
A <> B = C < D
```

is syntactically incorrect. Along with the relational operators, there are three `Boolean` operators: **not**, **and**, and **or**. These three operators are discussed at length in Section 5.6.

Operator	Operation	Operand Types	Type of Result
=	Equal	Compatible simple types, `pointer`, `set`, `String`, `packed-string` types	`Boolean` (*true* or *false*)
<>	Not equal		
<	Less than	Compatible simple types, `String` and `packed-string` types	
>	Greater than		
<=	Less than or equal		
>=	Greater than or equal		
<=	Subset of	Compatible set types	
>=	Superset of		
in	Member of	Compatible set types	

Figure 5.4 The relational operators and their types.

Figure 5.5 shows all of the operators and their precedence level. The Boolean operator **not**, which negates a `Boolean` value (changes the truth value of its argument), has the highest operator precedence. The Boolean operator **and** has an operator precedence equal to the "multiplying" operators, and the Boolean operator **or** has precedence equal to the "adding" operators. Notice that the relational operators have the lowest operator precedence. The @ operator is used with pointers, and its use is discussed in Chapter 12. The operator **in** is used only with "set" operands.

Operators	Level of Precedence	
@, not	1	(highest)
*, /, div, mod, and	2	
+, -, or	3	
=, <>, <, >, <=, >=, in	4	(lowest)

Figure 5.5 Operator precedence for arithmetic, Boolean, and relational operators.

5.4 CONTROL STRUCTURES FOR BRANCHING

5.4.1 The One-Way Selector

When writing an algorithm, you may wish to test a condition in order to determine the flow of execution. For example, suppose a commission of $50 will be paid to a salesperson having gross sales above $2500, and a $100 commission will be paid for gross sales above $5000. This situation requires that we test if gross sales are above $2500 and then test if gross sales are above $5000. If either test fails, the salesperson receives no commission.

There are three basic branching commands for developing an algorithm: **if-then**, **if-then-else**, and **case**. The **if-then** statement has the following syntax in Pascal:

```
if condition then
   statement;
```

where *statement* can be a `read`, `readln`, `writeln`, or a `write` statement, an assignment statement, a compound statement, a command for looping, or even another branching statement. This statement is often referred to as the ***then** clause*. When this branching command is executed, the condition is tested and, if *true*, the Pascal system branches to the **then** clause and continues execution. After completing execution of the **then** clause, the Pascal system continues execution with the statement following the **then** clause. If the condition fails, the Pascal system jumps to the statement following the **then** clause and continues execution. This type of branching command is often referred to as a *one-way selector*, because only one selection can be made if the condition is *true*.

As an example, we can compute the salesperson's commission in the situation described above with the following steps:

```
Commission := 0.0;
if Gross_Sales > 2500 then
```

```
   Commission := 50.0;
if Gross_Sales > 5000 then
   Commission := Commission + 50.0;
```

As you can see, the variable Commission is initially assigned a value of zero. The first **if-then** command is executed by testing the condition Gross_Sales > 2500. If this condition is *true*, the value of Commission is assigned a value of 50.0; otherwise, the value for Commission remains unchanged. Whether the first condition is *true* or *false*, the second **if-then** command is executed by first testing the condition Gross_Sales > 5000. If this condition is *true*, the present value of Commission is incremented by 50.0; otherwise, the value for Commission remains unchanged.

We can save some steps during execution if we note that the second test will not be performed if the first step fails. We can save the step of the second test (but only if the first step fails) by nesting the second **if-then** command within the **then** clause of the first **if-then** command. The following statements show these steps:

```
Commission := 0.0;
if Gross_Sales > 2500 then
   begin
      Commission := 50.0;
      if Gross_Sales > 5000 then
         Commission := Commission + 50.0;
   end;
```

In these steps, the **then** clause of the outer **if-then** command is represented by a compound statement composed of an assignment statement and another **if-then** command. If the first condition fails, the **then** clause for the outer **if-then** command is never executed. If *true*, the value of Commission is modified, and the second **if-then** statement is executed. The latter statements are preferred when writing a well-structured algorithm, because if Gross_Sales is less than $2500, it is only logical that we need not test whether the value of Gross_Sales is greater than $5000.

5.4.2 The Two-Way Selector

The branching command **if-then-else** is referred to as a *two-way selector*. Here is the Pascal syntax for this command:

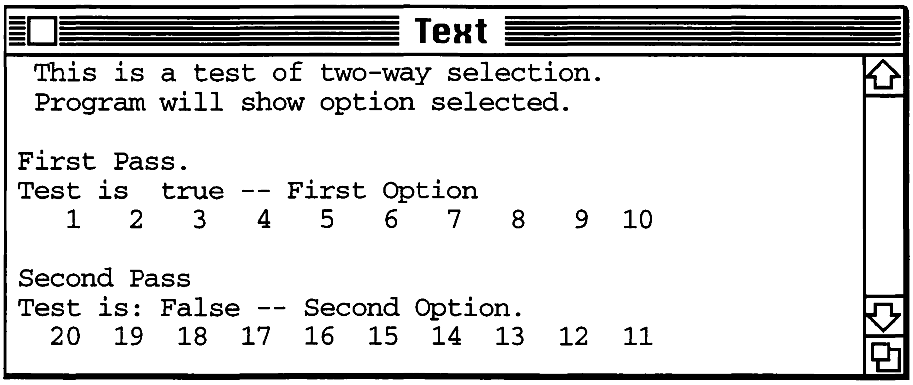
```
if condition then
   statement₁
else
   statement₂;
```

where either statement can be a read, readln, write, or a writeln statement, an assignment statement, a compound statement, a looping command, or another branching command. In Pascal both the **if-then** and **if-then-else** commands end with a semicolon. It is syntactically incorrect to place a semicolon before the keyword **else**. For the two-way selector just shown, $\texttt{statement}_1$ is referred to as the ***then clause*** and $\texttt{statement}_2$ as the ***else clause***. When executed, the condition is tested and, if *true*, the Pascal system branches to $\texttt{statement}_1$ to continue execution. If the condition fails, the Pascal system branches to statement$_2$, the **else** statement, to continue execution. After either the **then** clause or the **else** clause has been executed, the Pascal system continues execution with the statement following the **if-then-else**.

The following program applies two-way selection. In this program the Boolean variable Test is assigned a value *true* if the random number is found to be odd and *false* if even. If the variable Test is *true* , the values of the variable Index are displayed in ascending order; otherwise, they are displayed in descending order.

```
program Two_Way_Selection(input, output);
{ Purpose:  This program is a demonstration of using a two-way }
{           selector where the then clauses and else clauses are }
{           represented by compound statements. }
   var
      Test : Boolean;
      Index : integer;
begin
   ShowText;
   writeln(' This is a test of two-way selection. ');
   writeln(' Program will show option selected. ');
   writeln;
   writeln('First Pass.');
   Test := odd(random);
   if Test then { random number is odd }
      begin
         writeln('Test is: ', Test, ' -- First Option.');
      { Display values of Index starting at 1 and ascending to }
      { a value of 10. }
         Index := 1;
         repeat
            write(index : 4);
            Index := Index + 1;
         until Index > 10;
         writeln;
      end
   else { random number is even }
      begin
         writeln('Test is: ', Test, ' -- Second Option.');
      { Display values of Index starting at 10 and descending }
      { to a value of 1. }
         Index := 10;
         repeat
            write(Index : 4);
            Index := Index - 1;
         until Index = 0;
         writeln;
      end;
   writeln;
   writeln('Second Pass');
   Test := odd(random);
   if Test then { random number is odd }
      begin
         writeln('Test is: ', Test, ' -- First Option.');
      { Display values of Index starting at 11 and ascending }
      { to a value of 20. }
         Index := 11;
         repeat
            write(Index : 4);
            Index := Index + 1;
         until Index > 20;
         writeln;
      end
   else { random number is even }
      begin
          writeln('Test is: ', Test, ' -- Second Option.');
      { Display values of Index starting at 20 and descending }
      { to a value of 11. }
            Index := 20;
            repeat
               write(Index : 4);
               Index := Index - 1;
            until Index < 11;
            writeln;
      end;
end.
```

We can perform a trace of this program with the assistance of the program output shown in Figure 5.6 It is clear from this figure how compound statements can be used to create blocks of statements for execution in an **if-then-else** construct.



```
 This is a test of two-way selection.
 Program will show option selected.

First Pass.
Test is  true -- First Option
   1   2   3   4   5   6   7   8   9  10

Second Pass
Test is: False -- Second Option.
  20  19  18  17  16  15  14  13  12  11
```

Figure 5.6 Program output from `Two_Way_Selection`.

As an additional example, consider an algorithm for counting the number of partisan votes in a group of Democrats and Republicans. When executing, this program will ask the user to enter 1 for a Democratic vote and 2 for a Republican vote. If the number entered is neither 1 nor 2, the algorithm will respond with the message `VOTER FRAUD!` As each vote is cast, the algorithm will count the number of Democratic and Republican votes as well as the total number of votes. After a vote is cast, the user will be asked if he or she wishes to enter another vote with the prompt `Press C to continue, Q to Quit`. If the response is not the character `C`, the algorithm stops counting votes and displays a short report on the percentage of votes cast. If any fraudulent votes have been cast, the algorithm will also display the percentage of fraudulent votes. Input to our algorithm will be a vote entered from the keyboard, and output will be the percentage of Democratic votes, percentage of Republican votes, and percentage of fraudulent votes.

Here are the initial steps in solving our problem:

1. Initialize the three counters to zero: `Democratic_Votes`, `Republican_Votes`, and `Total_Votes`.
2. Repeat entering a vote, checking to see if the vote is Democratic, Republican, or fraudulent, until the response to the prompt is not the character `C`.
3. Compute the percentage of Democratic, Republican, and fraudulent votes.
4. Display the percentage of Democratic, Republican, and (if any) fraudulent votes.

Here is a refinement of our algorithm with a list of variables:

```
Algorithm Vote_Counter;
   { List of variables:
      Democratic_Votes, Republican_Votes, Total_Votes, Vote :
                                                        integer
      Percentage_Democrats, Percentage_Republicans,
      Percentage_Fraudulents : real
      Response : Char }
begin
{ Initialize three counters: Democratic_Votes, Republican_Votes,
  and Total_Votes. }
   Democratic_Votes <-- 0;   Republican_Votes <-- 0;
   Total_Votes <-- 0;
{ Repeat entering a vote, checking to see if the vote is
  Democratic, Republican, or fraudulent, and continue this until
  the response to continue voting is not the character 'C'. }
      repeat
      { Prompt the user for the next vote. }
         write( 'Please enter the next vote. ');
      { Prompt the user for either a Democratic or Republican vote
        by typing 1 for Democratic,  2 for Republican.  }
         read(Vote);
      { Count the number of Democratic and Republican votes. }
         if Vote = 1 then
            Democratic_Votes <-- Democratic_Votes + 1
         else
            if Vote = 2 then
               Republican_Votes <-- Republican_Votes + 1
         else
            write('VOTER FRAUD!' );
      { Modify the variable Total_Votes. }
```

```
        Total_Votes <-- Total_Votes + 1;
        write(' Press C to continue, Q to quit: ');
        read( Response );
     until  Response <> 'C';
  { Compute the percentage of Democratic and Republican votes
    multiplied by a scaling factor of 100. }
     Percentage_Democrats <-- ( Democratic_Votes * 100 ) /
                                                   Total_Votes;
     Percentage_Republicans <-- ( Republican_Votes * 100) /
                                              Total_Votes;
     Percentage_Fraudulents<-- 100.0 - ( Percentage_Democrats +
                                        Percentage_Republicans );
     { Display the percentage of Democratic, Republican, and, if
       any, fraudulent votes. }
        write( Percentage_Democrats );
        write( Percentage_Republicans );
        If  Percentage_Fraudulents > 0.0  then
           write( Percentage_Fraudulents );
end.
```

Here is the Pascal program Vote_Counter:

```
program Vote_Counter(input, output);
{ Purpose:  This example Illustrates the application of a }
{           one-way selection, a  two-way selection process, and }
{           the application of a repeat-until loop. }
   const
      Scale = 100;
   var
      Democratic_Votes, Republican_Votes: integer;
      Total_Votes, Vote : integer;
      Percentage_Democrats, Percentage_Republicans : real;
      Percentage_Fraudulents : real;
      Response : Char;
begin
   ShowText;
{ Initialize three counters: Democratic_Votes, Republican_Votes, }
{ and Total_Votes. }
   Democratic_Votes := 0;
   Republican_Votes := 0;
   Total_Votes := 0;
{ Repeat entering a vote, checking to see if the vote is }
{ Democratic,  Republican, or fraudulent, and continue this }
{ until the response to the prompt is not the }
{ character 'C'. }
   repeat
   { Prompt the user for the next vote. }
      writeln;
      write   ( 'Please enter the next vote.');
      writeln(' Type "1"<cr> for a Democratic vote, ');
      write('  "2"<cr> for a Republican vote. ');
      readln(Vote);
   { Count the number of Democratic and Republican votes. }
      if Vote = 1 then
         Democratic_Votes := Democratic_Votes + 1
      else
         if Vote = 2 then
            Republican_Votes := Republican_Votes + 1
         else
            begin
               writeln;
               write ('VOTER FRAUD!' );
               writeln
            end;
         { Modify the variable Total_Votes. }
         Total_Votes := Total_Votes + 1;
         writeln;
         write(' Press C to continue, Q to quit: ');
         read( Response );
         writeln;
   until  Response <> 'C';
{ Compute the percentage of Democratic and Republican votes. }
   Percentage_Democrats := ( Democratic_Votes * Scale ) /
                                                Total_Votes;
   Percentage_Republicans := ( Republican_Votes * Scale) /
                                                Total_Votes;
   Percentage_Fraudulents := 100.0 - ( Percentage_Democrats +
                                    Percentage_Republicans );
{ Clear the Text window with command Page. Display percentage }
{ of Democratic, Republican, and any fraudulent votes.  }
   Page;
   writeln(' Report on the election results: ');
   writeln;
   writeln ('Democratic Votes: ',Percentage_Democrats:4:2,' %');
   writeln ('Republican Votes: ',Percentage_Republicans:4:2,' %');
   if  Percentage_Fraudulents > 0.0  then
      writeln('Fraudulent Votes: ',Percentage_Fraudulents:4:2,'%')
end.
```

Before ending our discussion of this example, we should mention two minor points. First, we used a `read` statement instead of a `readln` statement to enter a value for `Response`. We then followed with a `writeln` statement. This approach allows a single character to be entered from the keyboard without having to press the `Return` key. The `writeln` statement terminates the print line displayed to the Text window. Second, the special library routine `Page` is used to clear the Text window of any previously displayed text before displaying any new text.

5.4.3 Multiway Selection

The third type of branch command is the **case** command, which allows multiway selection, beyond that allowed by the two-way selector. In Pascal the **case** command is represented by the following syntax:

```
case expression of
   label1 : statement1 ;
   label2 : statement2 ;
      •        •
      •        •
   labeln : statementn ;
otherwise
   statement
end;
```

Here are the special rules for using the **case** command.

1. The expression, sometimes referred to as a *selector,* can be of a standard `ordinal` type, such as an `integer`, `char`, or `Boolean`; an `enumerated` type; or a `subrange` type. It cannot be a `longint` type or a `real` type, because an `enumerated` set of reals is uncountable.
2. The labels, sometimes referred to as *case constants*, must be type-compatible with the selector.
3. Although case labels must be distinct, they do not have to be ordered.
4. A case constant can be represented by a list of labels, separated by commas.
5. Statements following the labels can be `read`, `readln`, `write`, `writeln`, assignment, compound, branching, or loop constructs.
6. The term **otherwise** followed by a statement is referred to as the **otherwise** clause.
7. When the **case** statement is executed, the selector is evaluated, and if one of the labels has a value equal to that of the selector, the Pascal system branches to that label and continues by executing the corresponding statement. If no label exists with a value equal to that of the selector, the Pascal system branches to the **otherwise** clause and executes the corresponding statement.
8. With Macintosh Pascal an execution error will occur, halting the program, if there is no label with a value equal to that of the selector and the **otherwise** statement is missing.
9. A **case** statement must end with the reserved word **end** followed by a semicolon.

The following examples illustrate some applications of the **case** statement:

```
Background := abs( random ) mod 5;
```

```
case Background of
   0 :  Pat := white;
   1 :  Pat := black;
   2 :  Pat := gray;
   3 :  Pat := ltgray;
   4 :  Pat := dkgray;
end;
```

The **otherwise** clause is not necessary in this example, because the value of the selector, Background, will always have a remainder in the range 0 through 4 from execution of the **mod** operator.

```
case Selector of
   's'  :  Value := sin( Angle );
   'c'  :  Value := cos( Angle );
   't'  :  Value := sin( Angle )/cos( Angle );
otherwise
   Value := ( exp( Angle ) + exp( -Angle ) ) / 2
end;
```

In this example, the selector is a char type and, when evaluated, will cause the **case** statement to branch to either label 's', 'c', or 't' if equal to one of the three, and to the **otherwise** clause if not.

```
read( Letter );
case Letter of
   'a', 'e', 'i', 'o', 'u' :  Vowel_Count := Vowel_Count + 1;
   'A', 'E', 'I', 'O', 'U' :  Vowel_Count := Vowel_Count + 1;
otherwise
   { null statement }
end;
```

In this last example, Letter is assumed to be of type char. If the value of this selector is neither an uppercase nor lowercase vowel, the **otherwise** clause is evaluated. Not having this **otherwise** clause could result in an error at execution time for Macintosh Pascal.

The **case** statement is sometimes convenient for performing table lookups; for example, consider the following problem. The Zeta-Data Merchant Company sells many Brand X products to customers who prefer to buy Brand X. The president of Zeta-Data has decided to use the Macintosh for computing the total cost for any customer buying a quantity of Brand X. The unit price of Brand X appears in the table shown below.

The sales tax charged to each customer is 5% of the cost of the items, with a shipping charge of $3 if the quantity ordered is less than 300 units. There is no shipping charge if the quantity ordered is greater than 299.

Before defining an algorithm, let us consider how we can use a **case** statement to determine the proper unit price for computing the cost. If we represent each entry in the table by a row number, starting with 1 and ending with 6, we can define a selector that, when given the quantity of the order, can provide a value that is 1, 2, 3, 4, 5, or greater.

Quantity of Brand X	Unit price
1 - 99	0.97
100 - 199	0.91
200 - 299	0.85
300 - 399	0.75
400 - 499	0.67
500+	0.51

One approach is to let the selector use the expression `(Quantity_Ordered` **`div`** `100) + 1`. Notice that if the quantity ordered is less than 100, the value of this expression is 1, because the value of `Quantity_Ordered` **`div`** `100` is 0. If the quantity ordered is between 100 and 199, the value of the expression is `2`, because the value of `Quantity_Ordered` **`div`** `100` now evaluates to `1`.

Here is an algorithm for this problem:

```
Algorithm Table_Lookup;
begin
   ShowText;  {Display the text window }
{ 1. Prompt the user for quantity ordered. }
   write( ' Quantity ordered: ' );
   read( Quantity_Ordered );
{ 2. Check that quantity ordered is greater then zero. }
   while  Quantity_Ordered <= 0  do
      begin
      { Display message indicating improper data. }
         write (' ******* IMPROPER DATA -PLEASE REENTER ******' );
         write { ' Quantity ordered: ' );
         read( Quantity_Ordered )
      end;
{ 3. Perform a table lookup to compute cost. }
   case  ( Quantity_Ordered div 100 ) + 1 of
      1 :   Unit_Price <-- 0.97;
      2 :   Unit_Price <-- 0.91;
      3 :   Unit_Price <-- 0.85;
      4 :   Unit_Price <-- 0.75;
      5 :   Unit_Price <-- 0.67;
      otherwise Unit_Price <-- 0.51
   end;
{ 4. Compute total cost of purchase. }
   Cost <-- Quantity_Ordered * Unit_Price;
   Sales_Tax <-- Cost * .05;
   Shipping_Cost <-- 0.00;
   If Quantity_Ordered < 300 then
      Shipping_Cost <-- 3.00;
   Total_Cost <-- Cost + Sales_Tax + Shipping_Cost;
{ 5. Report total cost for billing customer. }
   write( Cost, Sales_Tax, Shipping_Cost, Total_Cost )
end.
```

In this algorithm the variables `Cost`, `Sales_Tax`, `Total_Cost`, and `Unit_Price` are `real`, and `Quantity_Ordered` is of type `integer`. The program based on this algorithm follows. Added expressions are included in `writeln` statements to enhance the display of information to the Text window.

```
program Table_Lookup(input, output);
{ Purpose:  This program computes the unit price of items sold, }
{           including sales tax and shipping cost. }
   var
      Quantity_Ordered, Counter : integer;
      Cost,Sales_Tax,Shipping_Cost,Total_Cost,Unit_Price : real;
begin
   ShowText;
{ Prompt the user for quantity ordered. }
   write(' Quantity ordered: ');
   readln(Quantity_Ordered);
   writeln;
{ Check that quantity ordered is greater then zero. }
   while Quantity_Ordered <= 0 do
      begin
      { Display message indicating improper data. }
         write(' ******* IMPROPER DATA - PLEASE REENTER ******');
         writeln;
         write(' Quantity ordered: ');
         readln(Quantity_Ordered);
         writeln;
      end;
{ Perform a table lookup to compute cost. }
      case (Quantity_Ordered div 100) + 1 of
         1 :   Unit_Price := 0.97;
         2 :   Unit_Price := 0.91;
         3 :   Unit_Price := 0.85;
         4 :   Unit_Price := 0.75;
         5 :   Unit_Price := 0.67;
         otherwise
               Unit_Price := 0.51
      end;
{ Compute total cost of purchase. }
   Cost := Quantity_Ordered * Unit_Price;
   Sales_Tax := Cost * 0.05;
   Shipping_Cost := 0.00;
   if Quantity_Ordered < 300 then
      Shipping_Cost := 3.00;
   Total_Cost := Cost + Sales_Tax + Shipping_Cost;
{ Report total cost for billing customer. }
   Counter := 1;
   while Counter <= 30 do
      begin
         write('-');
         Counter := Counter + 1
      end;
   writeln;
   writeln(' Cost of items : ', Cost : 10 : 2);
   writeln(' Sales tax ___ : ', Sales_Tax : 10 : 2);
   writeln(' Shipping cost : ', Shipping_Cost : 10 : 2);
   writeln(' .............. ___________ ');
   writeln(' Total cost _ : $', Total_Cost : 10 : 2);
   writeln;
   Counter := 1;
   while Counter <= 30 do
      begin
         write('-');
         Counter := Counter + 1
      end;
end.
```

Could this program be written without a **case** statement? The answer is yes, by using a series of nested **if-then-else** statements. For example, if we know that the quantity ordered is greater than zero, we can specify that **if** the quantity ordered is less than 100, the unit price is 97 cents; **else if** the quantity ordered is less than 200, the unit price is 91 cents; **else if** the quantity ordered is less than 300, the unit price is 85 cents; **else if** the quantity ordered is less than 400, the unit price is 75 cents; **else if** the quantity ordered is less than 500, the unit price is 67 cents; **else** the unit price is 51 cents, because at this point the quantity ordered must be greater than 499. The Pascal code that could replace the **case** statement in the program Table_Lookup follows:

```
if  Quantity_Ordered < 100  then
   Unit_Price := 0.97
else
   if  Quantity_Ordered < 200  then
      Unit_Price := 0.91
    else
      if  Quantity_Ordered < 300  then
         Unit_Price := 0.85
       else
         if  Quantity_Ordered < 400  then
            Unit_Price := 0.75
         else
            if  Quantity_Ordered < 500  then
               Unit_Price := 0.67
            else
               Unit_Price := 0.51;
```

Sometimes a Pascal program involving nested **if-then-else** statements gives rise to syntactic ambiguity. For example, does the statement

```
if condition₁ then
   if condition₂ then
      statement₁
   else statement₂;
```

mean that the **else** clause with statement$_2$ is attached to **if** condition$_1$ **then**, or to **if** condition$_2$ **then**? In Pascal the rule states that the last **else** clause must always be attached to the closest **if-then**. This means that statement$_2$ above serves as an **else** clause for **if** condition$_2$ **then**. A clearer way of saying this is

```
if condition1 then
   begin
      if condition2 then
         statement1
      else  statement2
   end;
```

The following Pascal code represents the case where the **else** clause is attached to the first **if-then**:

```
if condition1 then
   begin
      if condition2 then
         statement1
   end
else  statement2 ;
```

You should use compound statements to help you understand the nesting levels of both branching and looping commands. In some instances, compound statements are necessary.

5.5 NESTED LOOPS

A pretest or post-test loop can be placed within the body of another loop. Consider the following example:

```
N := 1;
while N <= 10 do
   begin
      J := 1;
      repeat
         writeln( J, sqr(J), sqrt(J) );
         J := J + 1;
      until J > N;
      N := N+ 1
   end;
```

In this example, the variable N controls the outside loop represented by the **while-do** construct. The body of this outside loop is composed of three statements: an assignment statement initializing the control variable J to 1, a **repeat-until** construct, and a second assignment statement incrementing the value of the control variable N. The inner loop represented by the **repeat-until** construct is executed N times for each execution of the outside loop.

As a second example, consider the following code for trapping a vote that is neither Democratic nor Republican:

```
repeat
{ Prompt the user for the next vote. }
   repeat;
      repeat;
         writeln;
         write  ( 'Please enter the next vote.');
         writeln(' Type "1"<cr> for a Democratic vote, ');
         write('  "2"<cr> for a Republican vote. ');
         readln(Vote);
      until Vote > 0;
   until Vote <3;
{ Count the number of Democratic and Republican votes. }
   if Vote = 1 then
      Democratic_Votes <-- Democratic_Votes + 1
   else
      if Vote = 2 then
         Republican_Votes <-- Republican_Votes + 1;
      { Modify the variable Total_Votes. }
   Total_Votes <-- Total_Votes + 1;
   writeln;
   write (' Press C to continue, Q to quit: ');
   readln ( Response );
until  Response <> 'C';
```

Notice that the innermost nested **repeat-until** loop having the condition `Vote > 0` traps a fraudulent vote if the value entered is less than or equal to zero, and the outer nested **repeat-until** loop having the condition `Vote < 3` traps execution if the value of `Vote` is greater than or equal to 3. Though we can nest loops in a variety of ways, the structure of the loops used here can trap inappropriate input that would otherwise disable or crash the program. Computing the percentage of fraudulent votes and reporting these results in the program `Vote_Counter` is no longer necessary, so we can remove all references to variables or statements involving fraudulent votes.

5.6 BOOLEAN OPERATORS AND COMPOUND CONDITIONS

In addition to relational operators, Pascal supports `Boolean` operators for constructing complex `Boolean` expressions. We can often use `Boolean` operators to construct compound conditionals in which we need to test two or more conditions.

Figure 5.7 shows the three `Boolean` operators supported in THINK and Macintosh Pascal. `Boolean` operators can only be used in expressions that will return a `Boolean` value, that is, a value of either *true* or *false*.

Operator	Operation	Operand Types	Type of Result
or	Disjunction	`Boolean`	`Boolean`
and	Conjunction	`Boolean`	`Boolean`
not	Negation	`Boolean`	`Boolean`

Figure 5.7 List of the `Boolean` operators in Pascal.

A table showing the results of executing a `Boolean` operator appears in Figure 5.8. This table assumes that the variables `A` and `B` represent conditional expressions and that they have been evaluated before the `Boolean` operation is performed. Parentheses surround the expressions to emphasize the fact that these operators have higher operator precedence than relational operators and are either higher than or equal to arithmetic operators in operator precedence. The **or** operator says that when either one or both operands are *true*, the result is *true*. The **and** operator says that the result is *false* unless both of its operands are *true*.

Operands		Expression	Result
A	B		
False	*False*	(A or B)	*False*
False	*True*	(A or B)	*True*
True	*False*	(A or B)	*True*
True	*True*	(A or B)	*True*
False	*False*	(A and B)	*False*
False	*True*	(A and B)	*False*
True	*False*	(A and B)	*False*
True	*True*	(A and B)	*True*
False		(not A)	*True*
True		(not A)	*False*

Figure 5.8 Truth table for `Boolean` operators **and**, **or**, and **not**.

The `Boolean` operator **not** negates the value of a conditional expression. That is, if a conditional expression evaluates to a value *true*, then **not** *true* is *false*. If the conditional expression evaluates to a value *false*, then **not** *false* is *true*. The `Boolean` operators in conjunction with the relational operators allow the Pascal programmer to specify a wide range of conditions, through conditional expressions. These expressions can be used to control selection (one-, two-, or multiway) and the operation of the two principle loop structures. With control structures and loops, we can develop complex algorithms to solve nontrivial problems.

For an example where a compound conditional is used, consider the Pascal program `Package`:

```
program Package(input, output);
{ Purpose:  This program demonstrates the evaluation of a }
{          compound conditional. The program determines if a }
{          package is to be  shipped provided it has acceptable }
{          characteristics. }
   var
   Weight, Perimeter : real;
begin
   ShowText;
{ Request user to enter the package characteristics: weight }
{ and perimeter. }
   write('Package weight? ');
   readln(Weight);
   write('Package size? ');
   readln(Perimeter);
   writeln;
{ Examine the package to determine if it is acceptable. }
{ If package exceeds either a weight of 50 pounds or a }
{ perimeter of 200 inches, then the package is too big. If }
{ the weight is equal to or less than 50 pounds and the }
{ perimeter is equal to or less than 200 inches, then }
{ the package is acceptable for shipping. }
   if (Weight > 50.0) or (Perimeter > 200.0) then
      writeln('Package too BIG! ')
   else
      writeln('Package acceptable. ');
end.
```

Notice that the **if-then-else** statement uses the compound conditional `(Weight > 50.0)` **or** `(Perimeter > 200.0)`. If either the weight of the package is greater than 50 pounds or its perimeter is greater than 200 inches, or both, the message `Package too BIG!` is displayed; **Else**, if both of these conditions are *false*, the message `Package acceptable` is displayed.

We can express the same logical conditions in other ways. For example, each of the following branch statements performs the actions discussed above:

```
if (Weight <= 50.0) and (Perimeter <= 200.0) then
   writeln('Package acceptable. ')
else
   writeln('Package too BIG! ');
```

or

```
if  not( (Weight <= 50.0) and (Perimeter <= 200.0)   ) then
   writeln('Package too BIG! ')
else
   writeln('Package acceptable. ');
```

or

```
if  not ( (Weight > 50.0) or (Perimeter > 200.0) ) then
   writeln('Package acceptable. ')
else
   writeln('Package too BIG! ');
```

5.7 ITERATIONS REQUIRING SIMPLE COUNTERS

Pascal supports two loop commands that can employ simple counters for counting either up or down to a limit. For counting up to a limit, there is the **for-to** loop, with the following syntax:

```
for Control_Variable := Initial_Expr to Final_Expr do
   statement ;
```

where *statement* can be any of the types of statements previously studied as well as another **for** loop, and where the Control_Variable can be a standard ordinal type such as an integer, char, or Boolean; an enumerated type; or a subrange type. In the context of this construct, *statement* represents the body of the loop. When the **for** loop is executed, the initial and final expressions are evaluated only once with Control_Variable being assigned the value of Initial_Expr. At this point the value of Control_Variable is compared with that of Final_Expr and, if it is less than or equal, the body of the loop is executed. After execution of the body of the loop, the control variable is assigned the value succ(Control_Variable), and the condition between Control_Variable and Final_Expr is evaluated again to see if the body of the loop can again be executed. Execution of the **for** loop ends when the value of Control_Variable exceeds the value of Final_Expr. Values for both the initial and final expressions must be type-assignment-compatible with that of the control variable.

Here is a revised example of the program Electric_Bill from Chapter 1:

```
program Electric_Bill_Revised(input, output);
{ Purpose:  Example of a program using a for loop where the }
{           control variable is an enumerated type. }
   var
      Total_Consumption, Total_Cost, Consumption, Cost : integer;
      Average_Consumption, Average_Cost : real;
      Month : (January, February, March, April, May, June, July,
                  August, September, October, November, December);
begin
   ShowText;
{ Initialize the totals for consumption and cost. }
   Total_Consumption := 0;
   Total_Cost := 0;
{ Repeat entry of consumption and cost data for January }
{ through December. }
   for Month := January to December do
      begin
      { Enter data from the keyboard. }
         write('Enter consumption for month of ', Month, ' : ');
         readln(Consumption);
         write('Enter cost: ');
         readln(Cost);
      { Compute the partial sums. }
         Total_Consumption := Total_Consumption + Consumption;
         Total_Cost := Total_Cost + Cost;
      end;
```

```
{ Compute the average values of consumption and cost.}
   Average_Consumption := Total_Consumption / 12;
   Average_Cost := Total_Cost / 12;
{ Display the results. }
   writeln;
   write('Average monthly consumption: ');
   writeln( Average_Consumption : 7 : 2);
   writeln('Average monthly cost: ', Average_Cost : 6 : 2)
end.
```

In the revised program, the **repeat-until** loop has been replaced with a **for** loop. In this case the control variable for the **for** loop is a nonstandard enumerated type called Month. The **for** loop is executed 12 times, with the control variable Month being assigned an initial value of January. On reaching the end of the loop, the control variable Month is assigned succ(Month). On executing the loop for the last time, the value for Month is December. Notice that each time the loop is executed, the value for Month is displayed as part of a message prompting the user to enter data.

The second form of the **for** statement is given by the following syntax:

```
for  Control_Variable := Initial_Expr downto Final_Expr do
   statement;
```

When the **for** loop is executed, the initial and final expressions are evaluated only once, with Control_Variable being assigned the value of Initial_Expr. Here the loop is executed if the value of Control_Variable is greater than or equal to the value of Final_Expr. When the body of the loop has been executed, Control_Variable is assigned the value pred(Control_Variable).

What happens if we try to change the value of the final expression by executing an assignment or read statement from within the body of the loop? Remember that when the **for** loop is executed, the initial and final expressions are evaluated only once. Although it may appear that the value of the final expression has been changed, such changes have no effect on the number of iterations of the **for** loop. Once the **for** loop has completed execution, the value of the control variable is undefined. An error can occur at execution time if any attempt is made to change the value of the control variable from within the body of the loop. **For** loops are different from the previously discussed loops in that there is no explicit statement for changing the value of the control variable. The control variable is either incremented or decremented by internal code generated by the translator. In addition, there is no explicit expression for testing a condition. The code for testing is generated by the translator when interpreting the **for** loop. The following example compares code for displaying 30 dashes across the Text window by using a **while-do** loop and a **for** loop:

```
Counter := 1;
while Counter <= 30 do                 for  Counter := 1 to  30 do
   begin                                  write('-');
      write('-');                      writeln;
      Counter := Counter + 1
   end;
writeln;
```

Assuming that `Counter` is declared an `integer`, the value of `Counter` will be 31 after the **`while-do`** loop has been executed, but the value of `Counter` is unpredictable after the **`for`** loop has been executed.

The **`for`** loop has another interesting property. If the control variable is declared as a `subrange`, it will successfully execute even though on completing the last iteration of the loop, we assume that the control variable has been assigned a successor or predecessor value that is out of range. For example, if `Counter` is declared as a `subrange` type `1..30`, execution of the above **`while-do`** statement will be halted when it attempts to assign a value of 31 to `Counter`. As for the **`for`** loop, it successfully completes execution. To convince yourself, try these two loops with the variable `Counter` declared as a `subrange` type `1..30`.

5.8 PROBLEM ANALYSIS: DEVELOPING AN ALGORITHM REQUIRING BRANCHING AND LOOPING CONSTRUCTS

To examine the application of loops and branches in problem analysis, let us consider a game called `Beanpicker`. `Beanpicker` is an old game that had its origins before the development of computers. It can be played with any set of discrete objects such as stones, beans, or other small items. The game has some interesting properties that are useful for examining the development of an algorithm and the application of loops. `Beanpicker` has many variations in rules, but basically you begin with a fixed number of objects (beans), and each player is permitted to pick up a limited but variable number of beans. This selection process continues with the players taking turns until all of the beans are gone. The goal of the game is to select the beans so that your opponent is left with nothing to pick up during his or her last turn. The rules of our version allow players to select no more than 5 beans and no fewer than 1 during each turn. The game begins with 36 beans in the pool.

The process of problem analysis was discussed in Section 5.1. The end product of this process is an algorithm that can be translated into specific Macintosh Pascal instructions. Recall that every algorithm must have the following properties:

1. Finiteness
2. Definiteness
3. Input data objects
4. Output data objects
5. Effectiveness

Let us develop an algorithm for the Macintosh Pascal version of the game described above. First, let us apply the seven steps listed in Section 5.1.

1. Define the problem. The problem is to develop a version of the game in which the Macintosh is one player and the user the other. An explicit set of rules is necessary for the game to be properly played. A mathematical expression is required for defining an operational strategy that allows either player to win. The algorithm must also be able to determine whether the user (player) or computer (Macintosh) is the winner.
2. Determine if the problem has already been solved. Although this problem has been solved using other computers and languages, we will

assume that the program has not been implemented using Macintosh Pascal.

3. List the information required as input. Required input from the user is the number of beans to pick. The user is also asked whether to continue with another game. A random selection in the range of 1–5 (the number of beans it will select) is required from the Macintosh.
4. List the information required as output. Required output is the number of beans picked up by the Macintosh, the number picked up by the player, and the number of beans remaining in the pool. At the end of the game, the program displays the winner.
5. Determine the initial steps for defining a solution.
   (a) Display the title and rules. Set the initial number of beans at 36. (b) Mac picks 1–5 beans. (Mac will always go first.) (c) Display the number of beans remaining. (d) The player picks 1–5 beans. (e) Display the number of beans remaining. (f) Test if number of beans remaining = 0. If not, repeat Steps (b) to (e). (g) Report the winner. (The last selection was made by the winner.)
6. Refine and elaborate the steps of the algorithm `Beanpicker` and construct the loops necessary to complete the process.

```
begin
   (a) Provide an introduction by displaying title
       of game and rules.
   (b) Accept input from user regarding whether he
       or she wishes to play.
   (c) Test if input from Step (b) was 'yes'. If
       not, the algorithm terminates execution.
   (d) While { Continuation = 'yes' } do Steps (e)
       through (n).
   (e) Number_of_Beans <-- 36; { initialize bean
       pool }
   repeat through Step m
   (f) { Let the computer pick a number of beans. }
       if Number_of_Beans = 36 then
          MAC_Pick <-- abs(random) mod 6
       else
          MAC_Pick <-- Number_of_Beans mod 6
       { The minimum number of beans picked by the
          computer must be least 1. }
       if MAC_Pick = 0 then MAC_Pick <-- 1
       {Display number beans picked by computer.}
       write MAC_Pick
   (g) { Reduce number of beans in pool of beans. }
       Number_of_Beans <-- Number_of_Beans -
                                     MAC_Pick
       { Display the number of remaining beans. }
       write Number_of_Beans
   (h)  { Determine if the computer has won. }
       if Number_of_Beans = 0 then {Mac wins}
          write 'Mac wins!'
       else {begin else clause}
```

```
        (i) { The player now picks a number of beans. }
            read Player_Pick
        (j) { Modify the number of remaining unpicked
            beans. }
            Number_of_Beans <-- Number_of_Beans -
                                          Player_Pick
        (k) { Display the number of remaining beans. }
               write Number_of_Beans
        (l) { Determine if player wins. }
            if Number_of_Beans = 0 then {player wins}
               write 'You win!'
              { end else clause }
        (m) { Game ends when number of remaining beans
            is zero. }
            until Number_of_Beans = 0;
        (n) { Check to see if player wants to play again
            or quit. }
            read Continuation
            { Game returns to Step (d) to see if player
            wishes to continue. }
            { end while-do loop }
    end. { end of program }
```

Step (e) is where the game process begins. It also introduces the steps whereby the algorithm is made an effective player. Because there are 36 beans when the game is initiated, and because the algorithm and the player are limited to choosing from 1–5 beans, there is only one winning strategy. If the player who picks second always chooses a number of beans that, when added to the number of beans picked by the first player is 6, the second player will win.

In tracing the algorithm, the number of beans in the pool for the algorithm to pick from is 36, 30, 24, 18, 12, and 6, if the human player follows the winning strategy. With 6 remaining beans for the algorithm to choose from, and with the algorithm picking at least 1, the human player need only pick whatever beans remain to win.

The player who picks first can win only if the second player does not know or fails to follow the winning strategy. However, should the first player know the winning strategy and the second player not follow the winning strategy, the first player can gain the advantage.

If the algorithm picks second, having it use the winning strategy always makes it the winner. In our version, however, the algorithm is instructed to pick first, randomly. If the human player knows and follows the winning strategy, he or she will win. It is only when the human player deviates from the winning strategy that the algorithm can win. The statement

```
MAC_Pick <-- Number_of_Beans mod 6
```

produces the winning strategy for the algorithm because it is the basis for the algorithm's second and subsequent picks.

Let us refine Step 6 in our problem analysis by writing more precise steps. After this refinement, the conversion from the algorithmic notation to Pascal will easily follow. Detailed code for prompting the user and reporting results is left to the Pascal program.

Our primary interest here is to understand and be able to trace the algorithm to see that it is functional.

```
Algorithm Beanpicker;
{ The purpose for this algorithm is to provide a Macintosh Pascal
   version of the game Beanpicker. }
begin
{ Display the title and rules of the game. }
   write { title };
   write  { rules };
{ Check to see if player wants to play. }
   write  ( 'Do you wish to play? ');
   read ( Continuation };
{ Begin outer loop. }
   while Continuation = 'yes' do
      Number_of_Beans <-- 36;
      repeat
      { Computer selects a number of beans from the pool. }
         if Number_of_Beans = 36 then
            MAC_Pick <-- abs(random) mod 6
         else
            MAC_Pick <-- Number_of_Beans mod 6;
      { The computer must pick at least one bean. }
         if MAC_Pick = 0
            then MAC_Pick <-- 1;
      { Display the number of beans picked by the computer. }
         write ( MAC_Pick);
      { Determine number of beans remaining in pool. }
         Number_of_Beans <-- Number_of_Beans - MAC_Pick;
      { Display number of beans in the pool. }
         write  (Number_of_Beans)
      { Determine if the computer has won. }
         if Number_of_Beans = 0 then
            { report that Mac wins }
         else { player can choose beans from the pool }
            begin
            { When reading the player's choice, check the
             player's entry to exclude a value greater
             than five and less than one. }
               repeat
                  write (' Your pick: ');
                  read ( Player_Pick );
               until (Player_Pick > 0) and (Player_Pick < 6);
            { Reduce the number of beans chosen by the player. }
               Number_of_Beans <-- Number_of_Beans - Player_Pick
            { Display the number beans in the pool of beans. }
               write Number_of_Beans
            { Determine if the player has won. }
               if Number_of_Beans = 0 then
                  {report that the player wins }
            end
      until Number_of_Beans = 0 ;
   { Game ends when number of beans is zero. }
   { Check to see if player wants to play again or quit. }
      write { prompt to continue the game };
      read ( Continuation )
   end { while-do loop }
end.
```

What remains is to translate the Beanpicker algorithm into the Pascal program Beanpicker. The final form adds further detail in such areas as the presentation of the rules and the reporting of the winner. Where it is convenient, **for** loops are used for simple repetition.

```
program Beanpicker(input, output);
{ Purpose:  This program allows the user to play a game of skill }
{           where the player is pitted against the computer, and }
{           the computer wins most of the time. }
   var
      Number_of_Beans, Player_Pick, MAC_Pick, Index : integer;
      Continuation : string [3];
begin
   ShowText;
{ Introduction to the rules and the title of the game. }
   for Index := 1 to 5 do
      writeln;
   writeln('                     ***** BEANPICKER *****');
   for Index := 1 to 5 do
      writeln;
   write('There are 36 black beans. You may pick up one (1) to ');
   write(' five (5) beans; the object of the game is to be the
         last');
   write(' one to pick up beans, leaving no beans for your  ');
   writeln('opponent -- the Mac. Mac picks up first.');
   for Index := 1 to 5 do
      writeln;
{ Ask if player wants to play. If not, program will end. }
   write('If you wish to play, type "yes".  ');
   write('CAUTION...Anything but lowercase yes will quit
      BEANPICKER! ');
   write(' If you wish to quit, press the <Return> key. ');
   readln (Continuation);
{ Continue the game as long as player wishes. If not program }
{ passes to the end. }
   while Continuation = 'yes' do
      begin
         for Index := 1 to 3 do
            writeln;
      { Initialize number of beans at 36.  }
         Number_of_Beans := 36;
      { Begin the game by repeating the following steps. }
         repeat
      { Computer selects a number of beans from the pool. }
         writeln('MAC`S PICK');
         if Number_of_Beans = 36 then
            MAC_Pick := abs(random) mod  6
         else
            MAC_Pick := Number_of_Beans mod 6;
         { The minimum number of beans picked by the computer }
         { must be at least one. }
         if MAC_Pick = 0 then
            MAC_Pick := 1;
         { Display number of beans picked by the computer. }
         writeln('Mac picks up: ', MAC_Pick : 1);
      { Compute the number of remaining beans. }
         Number_of_Beans := Number_of_Beans - MAC_Pick;
      { Display the number of beans in the pool. }
         write('The number of beans remaining is: ');
         writeln( Number_of_Beans : 1);
      { Determine if the computer has won. }
         if Number_of_Beans = 0 then
            begin
               writeln('>>>>> MAC WINS! <<<<< ');
               writeln
            end
         else { player can choose beans from the pool }
            begin
            { When reading the player's choice, check the }
            { player's entry to exclude a value greater than }
            { five and less than one. }
               repeat
                  writeln('YOUR PICK');
                  write('Please pick up 1 to 5 beans. ');
                  readln(Player_Pick);
               until (Player_Pick > 0) and (Player_Pick < 6);
            { Compute the number of remaining beans. }
               Number_of_Beans :=
               Number_of_Beans - Player_Pick;
            { Display the number of beans in the pool. }
               write('The number of beans remaining is: ');
               writeln(Number_of_Beans : 1);
            { Determine if the player has won. }
               if Number_of_Beans = 0 then
                  writeln('>>>>> YOU WIN! <<<<< ');
               writeln;
            end;
         { Game ends when the number of beans is zero. }
      until Number_of_Beans = 0;
   for Index := 1 to 4 do
      writeln;
 { Check to see if player wants to play again or quit. }
   write('Do you wish to play again?  Type "yes".  ');
   write('CAUTION...Anything but lowercase "yes" will' );
```

```
      write(' quit BEANPICKER! ');
      write(' If you wish to quit, press the <Return> key. ');
      readln(Continuation);
   end; {end while }
   writeln;
   writeln('Thank you for playing BEANPICKER.  Have a nice
      day!');
end.
```

Beanpicker, based on a strategy using modular division through execution of the **mod** operator, is an example that uses nested loops such as **repeat-until**, **while-do**, and **for**. The outer loop represented by a **while-do** construct controls the playing of the next game, based on the player entering the string yes to the prompt asking if he or she wants to continue. The first inner loop, using a **repeat-until** construct, continues play until the number of beans in the pool is zero. When this occurs, the winner of the game is displayed. Once the program leaves the inner loop, the player is prompted with a message asking if he or she wants to continue by playing another game. If the player types the response yes, the game is repeated. A **repeat-until** loop nested within the first inner loop assures that the player cannot pick a set of beans that is either greater than 5 or less than 1. If the player chooses a value that is out of range, the program simply repeats execution of this loop by prompting the user and requesting an entry of beans. The player cannot leave the execution of this loop until a proper value is entered.

5.9 THINK PASCAL VERSUS STANDARD PASCAL

THINK and Macintosh Pascal have adopted the **otherwise** extension for the **case** statement found in some other Pascal compilers. The **otherwise** clause that is supported by THINK and Macintosh Pascal is not standard for Pascal. Without the clause, Pascal compilers have not been clear on what actions are taken if the **case** selector has a value that is not one of the **case** labels. In some compilers the program will continue to execute as if the **case** list had fallen through, as though an empty statement were specified and executed. Other compilers will flag the situation by raising an execution error. By setting the check option R (range checking) for any file unit that is in the Project window, the THINK Pascal compiler will check any **case** selector for being out of range and generate a bug dialog window indicating that the **case** selector is out of range if the value of the selector fails to match any of the **case** labels.

Macintosh Pascal requires that either the **otherwise** clause be used, or the **case** selector have a value that is one of its **case** labels. Not having a value raises an execution error. Unfortunately, Macintosh Pascal has no option to remove any range-checking generated by the translator.

THINK Pascal supports an additional feature with **case** labels that is not a part of Standard Pascal, nor is it supported by Macintosh Pascal. This feature involves the use of range labels for **case** labels in **case** statements. For example, consider the Pascal code required for performing the table lookup that follows:

```
case ( Quantity_Ordered div 100) + 1 of
   1 :   Unit_Price := 0.97 ;
   2 :   Unit_Price := 0.91 ;
   3 :   Unit_Price := 0.85 ;
   4 :   Unit_Price := 0.75 ;
   5 :   Unit_Price := 0.67 ;
   otherwise
         Unit_Price := 0.51 ;
end;
```

This code is somewhat awkward, because it requires a case expression that will generate an integer value ranging from 1 through 5. This same example can be written in THINK Pascal using range labels. Rather than requiring a formula that will generate the value 1 if `Quantity_Ordered` is in the range 1 through 99, the actual range can be applied as a **case** label, and the **case** expression is now limited to the `integer` variable `Quantity_Ordered`. The following shows the example with range labels used as **case** labels:

```
case  Quantity_Ordered  of
   1 ..    99 :   Unit_Price := 0.97 ;
   100 .. 199 :   Unit_Price := 0.91 ;
   200 .. 299 :   Unit_Price := 0.85 ;
   300 .. 399 :   Unit_Price := 0.75 ;
   400 .. 499 :   Unit_Price := 0.67 ;
   otherwise
                  Unit_Price := 0.51 ;
end;
```

THINK Pascal also supports additional commands that are not part of either Macintosh Pascal or Standard Pascal. These include the predefined commands `cycle`, `leave`, and `halt`. The command `cycle` forces the next iteration of an enclosing **while**, **repeat**, or **for** statement in the context of the program. The following code demonstrates this command by generating 10 random numbers but only computes the square root of positive numbers. The computation of the square root of a negative value is skipped by forcing the program to execute the command `cycle`:

```
{ Display a header for number and square root of number. }
   writeln('  Number    sqrt(Number)');
{ Display two columns of numbers.}
   for Count := 1 to 10 do
   begin
      Number := random;
      if ( Number < 0 ) then
         begin
            writeln( Number : 7 );
            cycle;
         end
      else
         writeln( Number : 7, sqrt(Number):20:5 );
   end;
```

The command `leave` allows the program to break out from an enclosing **while**, **repeat**, or **for** statement and continue with execution following any one of these statements. The following is an example of an infinite loop, in which the loop continues to execute as long as the value of `Number` is positive. If `Number` is negative, the program leaves the loop and continues execution with the statement that follows the while statement:

```
while true do
   begin
      Number := random;
      if ( Number < 0 ) then
         leave
      else
         writeln( Number : 7, sqrt(Number):20:5 );
   end;
```

The commands `cycle` and `leave` are similar in concept and execution to the commands `continue` and `break`, respectively, found in the programming language C.

The command `halt` halts the execution of a THINK Pascal program. While this command may appear useful, it can be confusing when a program is halted by executing this command if the command itself is deeply nested within the program or within several programmer-defined units.

In executing `Boolean` expressions, THINK Pascal supports the short-circuit `Boolean` operators `&` and `|`. The operator & is similar to the **and** operator in that it will return a `Boolean` value *true* or *false*. It is different from the **and** operator in that if the left operand of the `&` operation is *false*, the right operand is never evaluated, because the logical operation of *false* with the logical **and** of any other logical value is always *false*. For example, in the `Boolean` expression

```
( Player_Pick > 0 )  &  ( Player_Pick < 6 )
```

if `Player_Pick` is not greater than 0, the second test `Player_Pick < 6` is never executed. It is only tested if the first test `Player_Pick > 0` is *true* .

The `Boolean` operator | is similar to the **or** operation in that it also returns a `Boolean` value that is either *true* or *false* . It is different from the **or** operator in that if the left operand of | is *true* , it does not execute the right operand, because the logical operation of *true* with the logical **or** of any other logical value is always *true*. Only when the left operand of | is *false* will the right operand be tested. These two short-circuit operators are useful where we want to minimize the evaluation of `Boolean` expressions.

Before ending this section, we must make a few comments on using the function `random`. Although `random` is not a standard Pascal function, both Macintosh and THINK Pascal support it and allow random integers to be generated from –32,767 to +32,767. There is a difference in how each of these translators generates a set of random numbers when the function is used. For example, the program in Figure 5.2 generates a different set of random numbers each time it is executed under Macintosh Pascal. In THINK Pascal, this program always generates the same set of random numbers, because a special `longint` global variable called `randSeed` (short for random seed) is initialized to 1 each time the program begins execution. This global variable is important to programs compiled in THINK Pascal, because it represents the seed for generating the next random number using the function `random`. Given that it is assigned the same initial value each time the program `Simple_Loop` begins execution, the set of random numbers is always predictable. This side effect can be eliminated by assigning a random value to `randSeed` before calling upon the function `random`.

The following program uses a general utility procedure called GetDateTime to assign an initial value to randSeed. The procedure GetDateTime returns the current date and time, but as a longint number given in the number of seconds since midnight, January 1, 1904. It is useful for initializing randSeed with an initial value because for any instant of time, the procedure GetDateTime always provides a different value for its argument. This program does not translate under Macintosh Pascal.

```
program Simple_Loop_Modified (input, output);
{Purpose:   This program demonstrates the step needed to }
{           initialize the seed variable randSeed with a random }
{           number. This program will only execute under THINK }
{           Pascal.}
var
   Counter, Number: integer;
begin
{ Initialize the global variable randSeed with a random number }
   GetDateTime(randSeed);
{ Generate 10 random numbers using the function random. }
   Counter := 1;
   while Counter <= 10 do
      begin
         Number := abs(random);
         writeln(Number, sqrt(Number) : 8 : 3);
         Counter := Counter + 1;
      end;
end.
```

SUMMARY

This chapter introduced the basic principles of loops and branching and the commands supported by Pascal to develop and execute structured programs. In particular, we examined the post-test iteration loop, referred to as **repeat-until**, and the pretest iteration loop, referred to as **while-do**. They are important because they reinforce the properties of loops by requiring the use of explicit statements for initialization and modification of control variables, and explicit expressions for testing. The branching constructs **if-then**, **if-then-else**, and **case** were also examined for the purpose of illustrating one-way, two-way, and multiway selection. In the context of looping and branching, relational or Boolean expressions were also introduced. These expressions employ either relational or Boolean operators. Compound conditionals can be defined by combining several relational expressions with Boolean operators.

The **for** command, an additional loop statement for simple counters, was discussed. We also noted that although the **for** loop is not a post-test iteration loop, its control variable is never out of range.

REVIEW QUESTIONS

1. List the seven steps in the analysis of a problem.
2. What is meant by the term *algorithm* ?
3. What properties must every successful algorithm have?
4. Define the purpose for having a loop within a programming language.
5. What are the three basic types of loops?
6. What are the basic properties that all loops should have?
7. Why is the **while-do** construct referred to as a *pretest iteration loop*?
8. What is a compound statement?
9. Can compound statements be nested? Can you think of a test to support your answer?
10. Why is the **repeat-until** construct referred to as a *post-test iteration loop*?
11. Explain the difference between a **while-do** loop and a **repeat-until** loop.
12. If a **repeat-until** construct did not exist, how could a **while-do** loop be used to represent its actions?
13. What is the purpose of having a **for** loop if control constructs such as **while-do** and **repeat-until** are available?
14. What is the purpose of having a conditional expression? What value can a conditional expression have once it has been executed?
15. List the relational operators supported by Macintosh and THINK Pascal.
16. In relation to the arithmetic operators, what is the operator precedence level of the relational operators?
17. Assuming all variables to be `real`, what is the order of operations when the following expressions are evaluated?

    (a) `A - B / C >= D + E`
    (b) `Z <= sqrt(sqr(X) + sqr(Y))`
    (c) `C * (D + E) = (F + G) / G`

18. Assuming all variables to be `real`, which of the following expressions are syntactically correct in Macintosh and THINK Pascal?

    (a) `A + B <= F - Y > H + J`
    (b) `A - B < = U + J`
    (c) `A < B < G < H`
    (d) `(A + B) > R - J`

19. If `Flag` is a `Boolean` variable, why is the `Boolean` expression unnecessary in the following **if-then** statement? How could it be written?

```
if Flag = true then
   writeln(' The truth wins over evil. ');
```

20. What is the difference between the relational operator = , the use of the symbol = in a constant declaration, and the assignment operator := ?
21. Which of the following constant declarations are syntactically correct?

```
const
   Truth = true;
   Falsehood = false;
   Max_Value = 9999;
   Min_Value = -9999;
   Relation = Min_Value < Max_Value;
   Never_True = ( Max_Value = Min_Value );
```

22. List the three types of control constructs supported by Macintosh Pascal for branching.
23. What is the difference between the **if-then** and **if-then-else** constructs?
24. What purpose does the **case** statement serve in relation to one-way and two-way selection?
25. In the syntax rules for the control constructs **while-do**, **repeat-until**, **if-then**, **if-then-else**, **for**, and **case**, the term *statement* occurs. What types of statements can *statement* represent?
26. If a **while-do** construct did not exist, how could the conditional construct **if-then** and loop construct **repeat-until** be used to emulate a **while-do** loop?
27. What is meant by the term *compound conditions* ?
28. List the three Boolean (logical) operators supported by Macintosh and THINK Pascal.
29. What is the purpose of a truth table?
30. What is meant by the term Boolean *expression* ?
31. Is a relational expression a Boolean expression?
32. Is a relational expression a compound condition?
33. In relation to arithmetic and relational operators, what is the operator precedence for Boolean operators?
34. What special care should be taken when writing a compound conditional in Pascal?
35. Assuming all variables to be real, which of the following conditional expressions are syntactically correct?

(a) **not** A < B + C
(b) (A + C < L) **and** ((G - H < L) **or** (A = B))
(c) A + B < C **or** D
(d) **not** ((A < B) **and** (C + D > 999.99))

36. Are the following two conditional expressions logically equivalent when executed?

(a) **not** ((A < B) **or** C = D))
(b) (A >= B) **and** (C <> D)

37. Replace the following two **if-then** statements with only one **if-then** statement:

```
if A < B then
```

```
   if B < C then
      writeln(' success ' );
```

38. Replace the following **if-then-else** statement with a **case** statement:

```
if  ( A + B ) < ( C - D )  then
   A := ( A + B ) / 2.0
else
   C := ( C + D ) / 2.0;
```

Hint: The **case** statement needs only two labels, `true` and `false`, with the selector represented by a relational expression.

39. If Macintosh Pascal did not support an **otherwise** clause in its **case** statement, what control construct along with the **case** statement would be necessary to prevent the **case** statement from failing?
40. What is wrong with the following Pascal program? Try it with both Macintosh and THINK Pascal.

```
program Sample_One;
   var
      Counter : 1..100;
begin
   Counter := 1;
   while Counter <=100 do
      begin
         writeln( Counter );
         Counter := succ( Counter )
      end;
end.
```

41. Correct the program in Question 40, using a **for** loop, so that it can complete execution without errors.
42. Leaving `Counter` declared as `subrange` 1 . . 100, and assuming that the **for** loop construct did not exist in Pascal, how could the **while-do** loop be written so that the program would successfully execute?

PROGRAMMING EXERCISES

Many of following exercises require problem analysis, including the definition of a refined algorithm, with trace tables for hand testing. Programs are easier to write when an algorithm is developed and tested with several different examples. If you discover errors in your program when it is being checked or executed, return to your algorithm to see if the error lies with the syntax and/or semantic actions of the algorithm. If it does, correct the algorithm first, test again by hand before correcting the program, and again execute the program. These steps may at first seem to consume too much time in developing a program, but with practice they will save many hours of guessing if a program fails to execute. Developing a refined algorithm and testing by hand will often lead to fewer programming problems as you become more proficient. Practice is the key to successful programming.

1. A computer program is required to compute the average of a set of negative numbers. Here is an initial set of steps specifying an algorithm for this problem:

```
(a) Initialize summation and counter to 0.
(b) Prompt and enter a first number from the keyboard.
(c) While the value of Number is negative, do the following:
        (i) Add the value of the number to the summation.
        (ii) Increment the value of the counter.
        (iii) Prompt and enter the value of the next number.
(d) Compute the average of all negative numbers entered.
(e) Report on the total count and the average value.
```

   First, refine the algorithm by choosing formal variable names `Sum`, `Number`, `Counter`, and `Average`. Then establish a trace table, and check that your algorithm functions for several sets of numbers. What happens in your algorithm if the first number entered is negative? Finally, write a Pascal program using the Text window to view input and output.

2. Instead of using a **while-do** loop, write the algorithm and program in Exercise 1 using a **repeat-until** loop.

3. A program is needed to compute the average of both positive and negative numbers. Here are the initial steps specifying an algorithm for the problem:

```
(a) Initialize summation and counter to 0.
(b) Prompt the user to see if any numbers are to be entered.
(c) While the response is yes, do the following:
        (i) Prompt the user for the next number.
        (ii) Add the value of the number to the summation.
        (iii) Increment the counter.
        (iv) Prompt the user to see if he or she wishes to
                continue entering numbers.
(d) Compute the average of all numbers entered.
(e) Report on the total count and the average value.
```

   First, refine the algorithm by choosing formal variable names `Sum`, `Number`, `Counter`, `Average`, and `Response`. Then establish a trace table, and check that your algorithm functions for several sets of numbers. What happens in your algorithm if no numbers are entered? Finally, write a Pascal program using the Text window to view input and output.

4. Modify the algorithm and the program in Exercise 3 so that the response for entering another number can only be `YES` or `yes` to continue and `NO` or `no` to cease.

5. How could you modify Exercise 3, so that the set of numbers to be averaged would be picked randomly rather than by having the user enter values from the keyboard, as would the count for the number of elements in the set?

6. The following is an algorithm for determining the largest value for a set of numeric values entered from the keyboard:

```
Algorithm Largest_Value;
begin
{ Initialize the variable Largest. }
   Largest <-- 0;
   repeat
   { Enter the next number from the keyboard. }
      write( ' Enter next number: ');
      read( Number );
      if Largest < Number then
         Largest <-- Number;
   { Prompt user to see if another number should be entered. }
      write( ' Are additional numbers to be checked? ');
      read( Response );
   until Response = 'No';
        { Report on the largest number. }
end.
```

(a) Refine the algorithm, and establish a trace table for testing the algorithm.

(b) Convert your algorithm into a Pascal program, and apply the tests from Step (a) to validate your program.

7. Modify the algorithm and program in Exercise 6 to compute both the largest and smallest value of a set of numbers entered from the keyboard. Name the algorithm and program `Largest_and_Smallest`.

8. (a) Write an algorithm to determine if a triangle is equilateral, isosceles, or scalene, according to the following rules:

Equilateral triangle: All sides are equal.
Isosceles triangle: Two sides are equal.
Scalene triangle: All sides are unequal.

Assume that you are given three sides. The algorithm should report the type of triangle and the reason for its choice.

(b) Establish a trace table for your algorithm, and test it with several sets of data.

(c) Write a Pascal program for your algorithm.

9. A computer program is necessary for computing a letter grade from a numeric grade. Here are the ranges of numeric grades and letter grades:

Numeric grade	Letter grade
90–100	A
80–89	B
70–79	C
60–69	D
00–59	F

(a) Write an algorithm using nested conditional branch **if-then-else** constructs to compute a letter grade. Remember that on input, the value of the numeric grade can never be negative. If the value entered from the keyboard is improper, have your algorithm provide a message indicating this error and again prompt the user to enter the proper value. Continue doing this until the value entered is a positive grade.
(b) Establish a trace table, and use several different grades to test your algorithm.
(c) Write a Pascal program using your algorithm, and apply the test data in Part (b) to validate your program. Be sure you test for all grade levels.

10. Rewrite the algorithm and program in Exercise 9 using a **case** statement instead of nested branching constructs. Use only one **if-then-else** statement. Be sure to test your algorithm and program for the numeric grade 100.

11. (a) Write an algorithm that performs the following steps:
    (i) Select a random `integer` between 0 and 40.
    (ii) Prompt the user by explaining that a player must guess the number selected by the machine.
    (iii) Ask the player to enter a guess.
    (iv) After the player has entered a number, check that the value entered is between 0 and 40. If not, repeat Steps (iii) and (iv).
    (v) Tell the player that the value entered is too large, too small, or proper.
    (vi) If the player has failed to guess, return to Step (iii).
    (vii) Once the player has guessed the proper value, display the number of guesses.
    (viii) Prompt the player for repeating the game. If response is yes, the program returns to Step (iii). If no, the algorithm terminates execution.

(b) Establish a trace table, and test your algorithm with several examples.
(c) Write a Pascal program for your algorithm, and apply the examples of Part (b) as tests.
(d) How can the algorithm and program be modified to display the message `getting warmer` as the player gets closer to the correct value, and the message `getting colder` as the player gets farther from the correct value?

12. The Kilo-Watt Electric Company has set the following electric rates for its residential customers:

	Residential Service:
$2.78	minimum customer charge per month plus
$0.05764	per kwh for the first 100 kwh used per month
$0.05306	per kwh for the next 300 kwh used per month
$0.04878	per kwh for all in excess of 400 kwh used

(a) Write an algorithm that prompts the user for the following information:

Customer name
Street address
City and State
Zip code
Present reading
Previous reading

This algorithm then computes the electric bill based on the rates, including a 3% charge for school tax based on the total cost for the monthly service. If the present reading is less than the previous reading, have your program terminate with a message `Improper meter readings. Present reading less then previous reading.` If the present reading is greater than or equal to the previous reading, the following must be displayed to the Text window:

```
   Customer Name:
 Street Address:
    City & State:
        Zip code:
________________________________________________
Present reading:   Previous reading:    Kwh used:

Cost for power: ............... $
School tax (3%): ..............
                                          ___________
Total cost: ................... $
________________________________________________
```

(b) Write a Pascal program for the algorithm, using several values of meter readings. Be sure to compare these with hand calculations. If a printer is connected to your Macintosh, choose the option for printing the data from the Text window to the printer.

13. A quadratic equation is represented in the following form:

$$ax^2 + bx + c = 0$$

where x is unknown. The roots (values for x for which the quadratic equation is identically zero) are given by the following equations:

$$x_1 = \frac{-b + \sqrt{(b^2 - 4ac)}}{2a}$$

$$x_2 = \frac{-b - \sqrt{(b^2 - 4ac)}}{2a}$$

provided a is nonzero. The term $b^2 - 4ac$ is referred to as the *discriminant.* The following rules exist for the number and type of roots:

If a is zero, and b and c are not zero, then there exists only one root given by $x = -c / b$. If a is not zero, then there exist two real roots if the discriminant is positive, and two complex roots if the discriminant is negative. If the discriminant is negative, then the value

$$-b / (2a)$$

is referred to as the real part, and

$$(abs(b^2 - 4ac))^{1/2} / (2a)$$

as the imaginary part.

(a) Write an algorithm for computing the roots of a quadratic equation that accepts from input the values of a , b , and c.
(b) Write a Pascal program for the algorithm in Part (a) that can display one of the following outputs:

(i) No solution if a and b are both zero.
(ii) One real root and its value if a is zero and b is not zero.
(iii) Two real roots if the discriminant is positive, and their values.
(iv) Two complex roots if the discriminant is negative, and their values, using the following format:

```
x  =  real part  +  I  imaginary part
 1

x  =  real part  -  I  imaginary part
 2
```

14. This problem requires both **div** and **mod** operations. Assume that a program is needed to teach people how to count exact change. This program will perform the following steps:

    (a) Provide the user with a message that the program will choose a value for the cost of items. This value is randomly chosen and must be a number expressed in terms of dollars and cents (no fractions of pennies).
    (b) Ask the user to enter a payment for computing change. If the value entered is smaller than the cost, it will continue prompting the user until a proper value is entered. The payment cannot involve any fractions of pennies.
    (c) Compute the exact change by displaying the following output:

```
                          Cost of item(s): $
                              Amount paid: $

                 Exact change to the customer:

                        Number of $20 bills:
                        Number of $10 bills:
                         Number of $5 bills:
                         Number of $1 bills:
                         Number of quarters:
                            Number of dimes:
                          Number of nickels:
                          Number of pennies:
```

    Write an algorithm for the problem, and then transform your algorithm into a Pascal program. Check with several values entered by the customer. How could this exercise be modified, so that the user would first have to enter t. ` exact change for bills and coins, then have the program display the information with the user's values next to the computer's computation? Whenever the user types the wrong value for change, the program must flag this with the message WRONG VALUE.

15. Let us assume we need to show a trace of the Beanpicker program. Modify Beanpicker, so the output displayed to the Text window is similar to the screen dump shown in Figure 5.9. *Hint*: Replace the statements

```
writeln('YOUR PICK');
write('Please pick up 1 to 5 beans. ');
readln(Player_Pick);
```

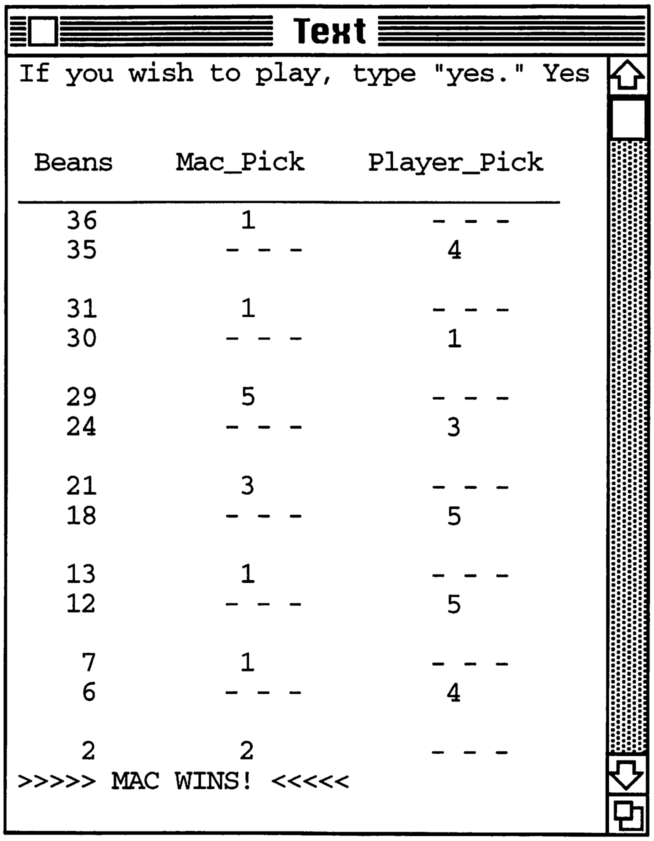

```
If you wish to play, type "yes." Yes


 Beans     Mac_Pick     Player_Pick
 ____________________________________
   36         1             - - -
   35       - - -             4

   31         1             - - -
   30       - - -             1

   29         5             - - -
   24       - - -             3

   21         3             - - -
   18       - - -             5

   13         1             - - -
   12       - - -             5

    7         1             - - -
    6       - - -             4

    2         2             - - -
>>>>> MAC WINS! <<<<<
```

Figure 5.9

with `Player_Pick := abs(random) mod 6`. Execute your modified program several times to convince yourself that before the last two picks, there are always 6 beans in the pool.

16. (a) Write an algorithm that will display the values of `sin`, `cos`, and `tan` for angles in the range $-\pi/2$ to $+\pi/2$. Your algorithm must prompt for an initial angle and a final angle. This angle must be within the specified range. If not, prompt to reenter these two values. The stepping (incremental) value must also be entered and must always be positive. This implies that the initial angle is always assumed to be greater than the final angle. Where the value of `tan` cannot be displayed, have your algorithm display either the term `+INFINITY` or `-INFINITY`. You can determine these angles by writing a short test program displaying the angle and the tangent of the angle. When an overflow error message appears, you will have an estimate for the angle.
    (b) Convert your algorithm into a Pascal program, and test your program for several sets of initial and final angles.

17. (a) Write an algorithm that will count the number of vowels in a lineof characters. Assume that the user will be prompted to enter a line of characters. A line is assumed terminated when it ends with a period. This algorithm must be capable of counting vowels in both uppercase and lowercase as well as calculating the percentage of vowels in the total number of characters. The algorithm must display a report in the following format:

Report on number of vowels:

```
Total number of characters:
    Vowel     Number     Percentage
________________________________________________

      A
      E
      I
      O
      U
________________________________________________
```

(b) Test your algorithm, using several different sentences.

(c) Convert the algorithm into a Pascal program. Use the set of tests from Part (b) to check your program.

18. The XYZ Screw and Nut Company has the following unit-price chart for screws or nuts purchased in large quantities:

Quantity	**Unit Price ($)**
1–65	0.098
66–105	0.087
106–229	0.081
230–399	0.079
400–499	0.075
500–799	0.071
800–999	0.065
1000 +	0.050

The shipping charge is based on the following schedule:

Total Cost of Parts ($)	**Shipping Rate** ($)
1–10	1.05
11–20	1.55
21–30	1.75
31–40	1.85
41–50	1.95
51–60	2.05
61–70	2.15
71–80	2.25
80 +	2.50

(a) Write an algorithm that will prompt for the name, street address, city, state, zip code, and quantity ordered, and will compute the total billing cost from the sum of the total cost of parts and the shipping charge. Output should appear as follows:

```
------------------------------------------------

     <<<<<   XYZ Screw and Nut Company >>>>>

          Name:
Street Address:
  City & State:
      Zip code:

================================================

Quantity  Parts Ordered:
          Cost of parts:        $
          Shipping cost:

________________________________________________

             Total cost:        $
------------------------------------------------
```

Be sure that your algorithm does not try to compute with a negative quantity entered for the number of parts ordered.

(b) Select several values to test your algorithm.
(c) Write a Pascal program for your algorithm. Test your program with the test data given in Part (a).

19. The factorial of a number *N* is defined as

```
1 * 2 * 3 * · · · * (N  - 1) X N.
```

For example, the factorial of 6 is

```
1 * 2 * 3 * 4 * 5 * 6 = 720.
```

(a) Write an algorithm that will accept from input an `integer` number and compute the factorial of this number as a `real` value. Special conditions exist for computing the factorial:

(i) Factorial is always computed for an `integer` number.
(ii) Factorial is always computed for a positive number.
(iii) The factorial of 0 is 1, and the factorial of 1 is 1.

(b) Write and test a Pascal program for Part (a).

20. A company has established a hiring policy for employing new engineers. It has decided to enter the following qualifications into a computer program, so that on examining an application, the program can determine if the candidate should be invited for an interview:

Field	Degree	Years of experience	Age
Electrical	BS	1–2	21–23
Mechanical	BS	3	24–27
Chemical	BS	1–3	21+
Electrical	MS	5	27–35
Mechanical	MS	5	27–38
Electrical	PhD	0–5	26+
Mechanical	PhD	5–8	36+

(a) Write an algorithm for qualifying candidates on the basis of the table. The algorithm can use both **`if-then-else`** and **`case`** statements. You must choose a sufficient number of test cases to show that the algorithm is functional. If a candidate does not qualify, provide some reason for rejection.

(b) Write a Pascal program for the algorithm. Use the test cases to show that the program is functional.

Chapter 6

Basic Graphic and Mouse Commands

OBJECTIVES

After completing Chapter 6, you will know the following:

1. The purpose of the QuickDraw1 library.
2. The routines needed for drawing simple lines.
3. The routines needed for drawing geometric patterns such as circles, rectangles, ovals, and rectangles.
4. The routines for allowing input from the mouse.
5. The commands for setting the size of Text and Drawing windows and for showing these windows on the screen.

6.1 QUICKDRAW LIBRARY

The Macintosh computer is a true graphics machine. All of its screen displays are provided by a complex graphical system embedded within the firmware of the computer and supported by a collection of special programs referred to as *routines* or *procedures*. A procedure is a set of commands, written in a high-level language such as Macintosh Pascal, for performing a specific task. A collection of several such procedures is known as a program library. For Macintosh Pascal there are two special libraries for graphics and text: `QuickDraw1` and `QuickDraw2`. There are 75 procedures in `QuickDraw1` and 70 procedures in `QuickDraw2`. Some of them are discussed in this chapter; some of the remaining procedures and functions are discussed later in the book. (All of these other procedures and functions are discussed in the earlier edition of this book.)

`QuickDraw1` is a Macintosh Pascal library containing various constants, types, variables, and procedures that may be borrowed by a Pascal program. They are made available to a Macintosh Pascal program by including the library name in the **uses** clause, as shown in the following program lines:

```
uses
   QuickDraw1;
```

This clause follows the program statement and precedes all other declared identifiers local to a Macintosh Pascal program. Because the `QuickDraw1` library is frequently used, it is not mandatory to include this statement in Macintosh Pascal programs. We will make a practice of including the **uses** clause, even when it is not required, to emphasize the application of a library within a Pascal program.

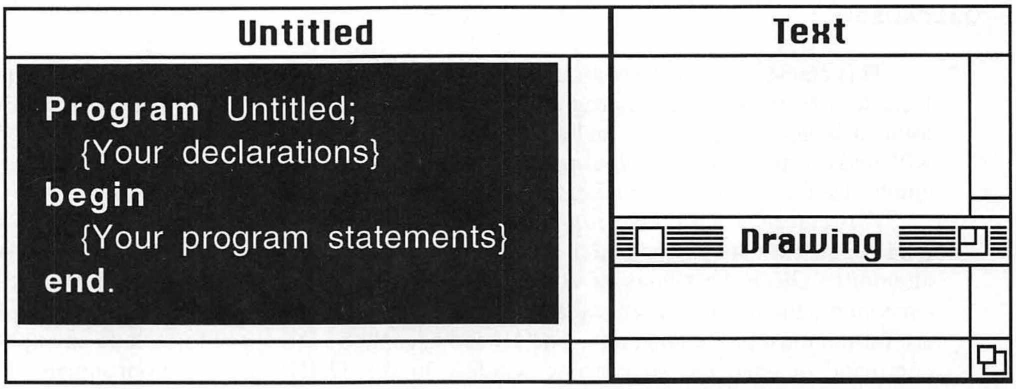
This clause is not allowed in a THINK Pascal program, however. Inclusion of the `QuickDraw1` statement under the **uses** command will cause an error when you attempt to check the syntax or compile the source program. With this and two other exceptions, the Macintosh Pascal and THINK Pascal programs discussed in this chapter are for the most part compatible. The second exception is the need to include an explicit command to open the appropriate window in the THINK Pascal environment. For example, most of the programs in this chapter require the insertion of the `ShowDrawing` command after the first **begin** if a program is to execute under THINK Pascal. The third exception is the format of the `GetMouse` command, which is handled differently by Macintosh and THINK Pascal. The difference is explained in Section 6.4.

In this chapter we will be concerned with the procedures `HideAll`, `ShowText`, `ShowDrawing`, `ShowTextRect`, `SetDrawingRect`, `GetTextRect`, and `GetDrawingRect`. These procedures can change the content of the Drawing and Text windows. Although these procedures are part of the the `QuickDraw1` library, they are not part of the embedded firmware procedures of the Macintosh computer. The **uses** clause allows us to borrow these routines from the `QuickDraw1` library for drawing and displaying text.

The complete Macintosh screen consists of an array of individual points called *pixels*. For a Macintosh with a nine-inch screen, this array consists of 175104 pixels.[1] The word pixel is derived from *picture element*. Each pixel is displayed as either a white or black dot. The nine-inch screen is 342 pixels high and 512 pixels wide, with a density of 72 pixels per inch in both the horizontal and vertical directions. The ability to display graphic images enables the Macintosh computer to define and draw varying environments called *grafPorts*. For the Macintosh computer, a grafPort represents a complete drawing environment for defining wh ~e graphic operations are performed and their effect on the drawing environment. As shown in Figure 6.1, when Macintosh Pascal is first loaded, three grafPorts, referred to as the Program window, the Text window, and the Drawing window, are displayed. When executing a program, the Pascal system displays text only to the Text window and graphic information only to the Drawing window. Entering and editing a Pascal program occurs in the Program window only. By dragging the mouse, you can cause any one of these three windows to capture the complete screen. Initially the

[1] The number of pixels will vary depending on the type of monitor or machine you are using. The early Macintoshes, the Macintosh SE and SE/30, and the Macintosh Classics all use a nine-inch screen. The Macintosh II, Macintosh LC, and other newer machines can use a variety of monitors of different sizes. For the RGB 13-inch monitor, the array consists of 307,200 pixels; 480 vertical pixels and 640 horizontal pixels. If you are using a different screen, experiment with the program `Figure6_2` to determine the height and width or your screen.

Drawing window is a 201-by-201 pixel square, shown in Figure 6.2. By dragging the mouse, the Drawing window can be made to fill the screen, becoming 500 pixels wide by 300 pixels high.

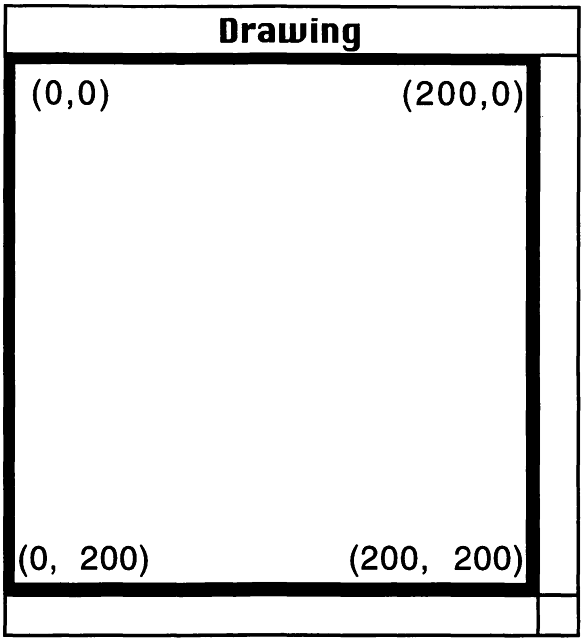


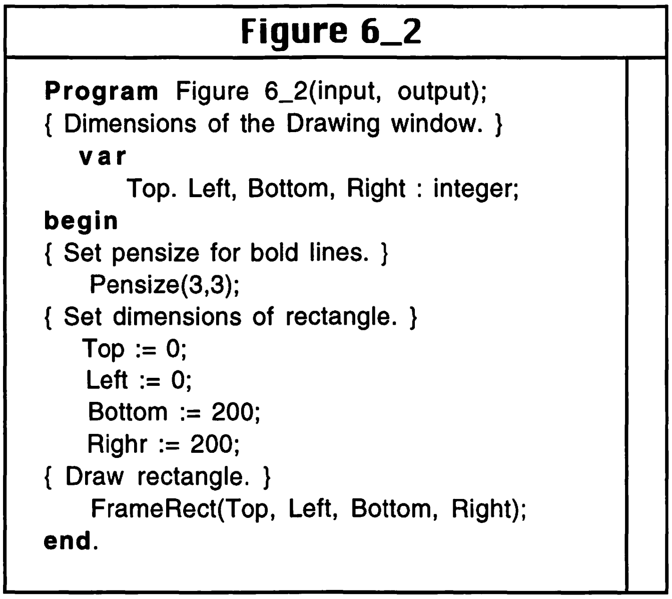
```
Program Untitled;
  {Your declarations}
begin
  {Your program statements}
end.
```

Figure 6.1 The Macintosh Pascal grafPorts.

Figure 6.2 presents the listing of a Macintosh Pascal program in the Program Window. This program displays a bold-outlined rectangle to the Drawing window (also shown in Figure 6.2). The coordinates at each corner are inserted for reference purposes and are not a part of the program. To demonstrate this program in THINK Pascal, you should insert a `ShowDrawing` command immediately after the **begin**. The command `ShowDrawing` opens the Drawing window, allowing you to view the lines and/or figures displayed to this window. This is important in a THINK Pascal program because, unlike Macintosh Pascal, THINK Pascal does not automatically open the Drawing window when it is needed. The program then appears as shown below:

```
program Figure6_2(input, output);
{ Purpose:  Set dimensions of the Drawing window }

   var
      Top, Left, Bottom, Right : integer;

begin
   ShowDrawing;
{ Set pen for bold lines. }
   Pensize(3, 3);
{ Set dimensions of rectangle. }
   Top := 0;
   Left := 0;
   Bottom := 200;
   Right := 200;
{ Draw rectangle. }
   FrameRect(Top, Left, Bottom, Right);
end.
```

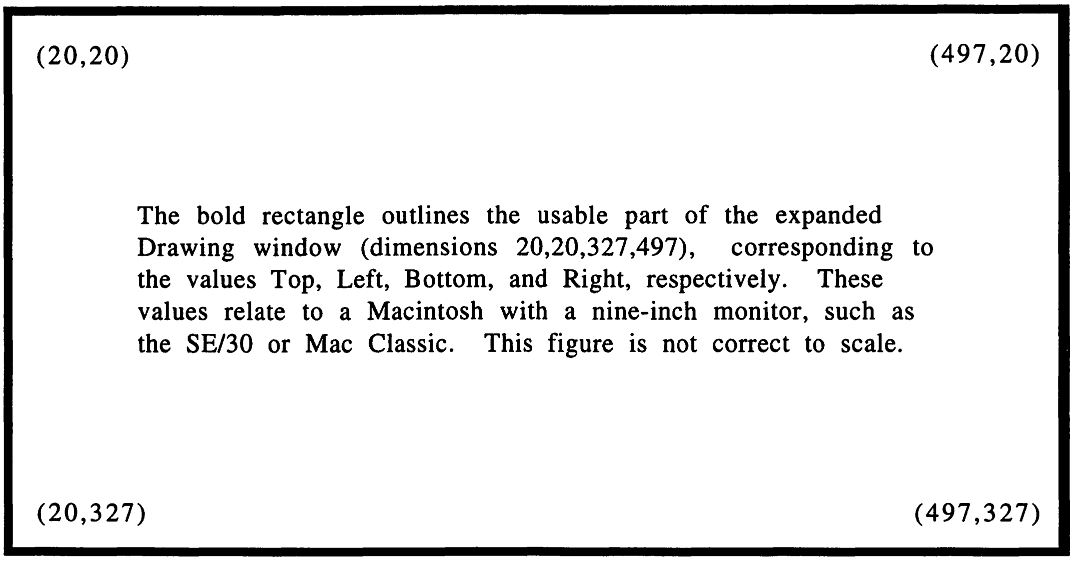




```
Program Figure 6_2(input, output);
{ Dimensions of the Drawing window. }
  var
      Top. Left, Bottom, Right : integer;
begin
{ Set pensize for bold lines. }
    Pensize(3,3);
{ Set dimensions of rectangle. }
   Top := 0;
   Left := 0;
   Bottom := 200;
   Righr := 200;
{ Draw rectangle. }
    FrameRect(Top, Left, Bottom, Right);
end.
```

Figure 6.2 Dimensions of the Drawing window in Macintosh Pascal (initial size).

Figure 6.3 shows the coordinates for the Drawing window, using the complete screen except for the space given to the menu bar and the elevator bars. Notice that the origin is still located at the upper left corner of the window. For convenience, we consider the horizontal axis as the *x*-axis and the vertical axis as the *y*-axis of a two-dimensional coordinate system. Any point within the Drawing window may be represented by an (*x*,*y*) pair. For example, the point (40, 50) represents *x* as 40 and *y* as 50. That is, the first number, 40, is the horizontal location, and the second number, 50, is the vertical location for a single screen pixel. It is important to note that the geometric representation used here is different from the standard convention of an *x*,*y* plane where the origin is at the lower left corner.



The bold rectangle outlines the usable part of the expanded Drawing window (dimensions 20,20,327,497), corresponding to the values Top, Left, Bottom, and Right, respectively. These values relate to a Macintosh with a nine-inch monitor, such as the SE/30 or Mac Classic. This figure is not correct to scale.

Figure 6.3 The expanded Drawing window.

6.2 DRAWING SIMPLE LINES

In this section we examine three procedures for drawing lines: `MoveTo`, `LineTo`, and `DrawLine`. Two other procedures, `PenSize` and `WriteDraw`, are also considered. To begin, assume that an imaginary pen exists in a grafPort such as the Drawing window. Before the pen is used for drawing, we must position it on the screen. The procedure used to accomplish this is called `MoveTo`, which uses the following syntax:

```
MoveTo( x, y );
```

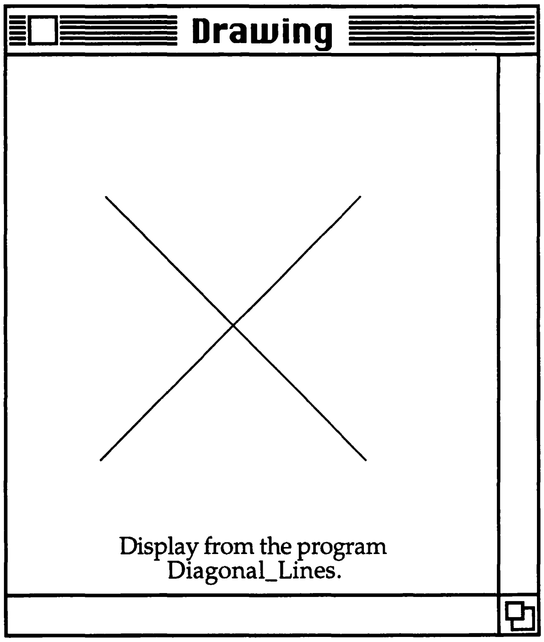
When executed, this statement moves the point of the pen to a location in the Drawing window given by the (*x* , *y*) pair. It does not perform any drawing.

A second procedure, called `LineTo`, allows you to draw a line from one point to another. `LineTo` uses the following syntax:

```
LineTo( x, y );
```

When executed, this command draws a line from the current pen location to the position specified by the values of *x* and *y*. After the line has been drawn, the pen has a new current pen position given by arguments (*x*,*y*). For example, the following Macintosh Pascal program, `Diagonal_Lines`, first moves the pen to the center of the

Drawing window, given as (100, 100). It then draws four diagonal lines from the center to the following four points: (50, 50), (50, 150), (150, 50), (150, 150). After each line that is drawn using the procedure `LineTo`, procedure `MoveTo` is used to reposition the pen to the center point (100, 100). Figure 6.4 shows the Drawing window after execution.



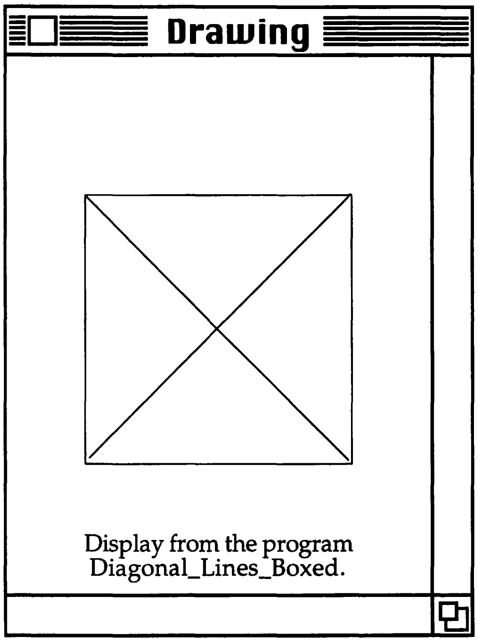
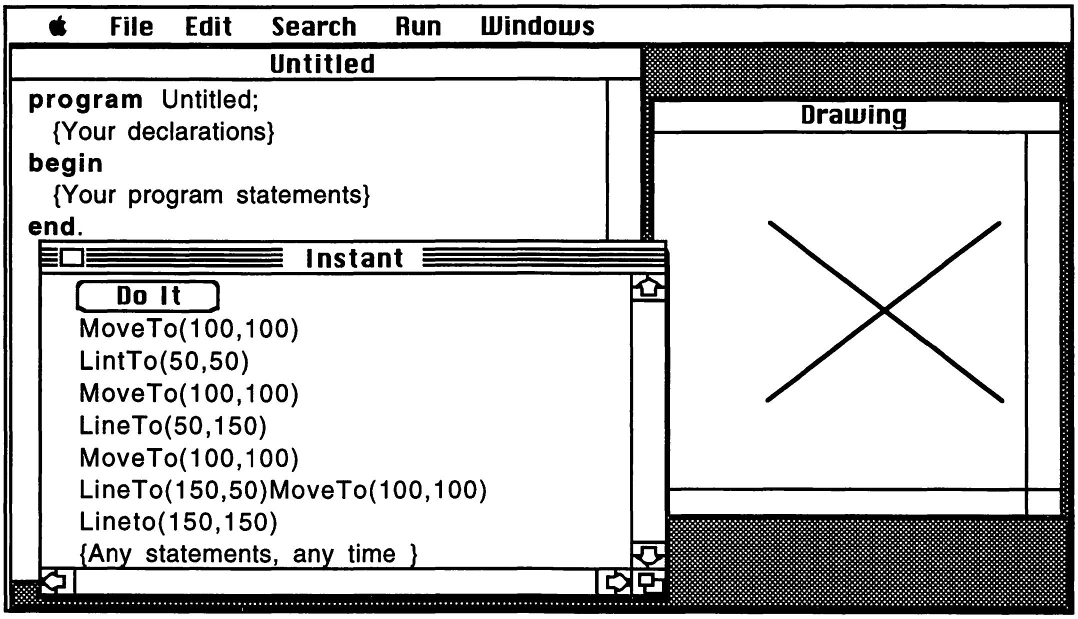


Figure 6.4 Displays from the programs `Diagonal_Lines` and `Diagonal_Lines_Boxed`.

```
program Diagonal_Lines(input, output);
{ Purpose:  This Macintosh pascal program executes the library }
{           procedures MoveTo and LineTo. }
   uses
      QuickDraw1;
begin
{ Move the pen to the center of the Drawing window and draw a }
{ diagonal line. }
   MoveTo(100, 100);
   LineTo(50, 50);
{ Move the pen to the center of the Drawing window and draw }
{ another diagonal line. }
   MoveTo(100, 100);
   LineTo(50, 150);
{ Move the pen to the center of the Drawing window and draw }
{ another diagonal line. }
   MoveTo(100, 100);
   LineTo(150, 50);
{ Move the pen to the center of the Drawing window and draw }
{ another diagonal line. }
```

```
   MoveTo(100, 100);
   LineTo(150, 150);
end.
```

The same steps can be tested without execution of a full Pascal program by using the Instant window.[2] Figure 6.5 illustrates this for Macintosh Pascal with both the Instant and Drawing windows opened. Eight program lines are inserted and then executed by clicking on the **Do It** button. Once successfully tested, these program lines can be either cut or copied, by using the **Cut** or **Copy** options in the **Edit** menu, and pasted into the Program window. Be sure to keep the Instant window in mind when you need to experiment with procedures or functions from the `QuickDraw1` library.



```
program Untitled;
  {Your declarations}
begin
  {Your program statements}
end.
```

```
MoveTo(100,100)
LintTo(50,50)
MoveTo(100,100)
LineTo(50,150)
MoveTo(100,100)
LineTo(150,50)MoveTo(100,100)
Lineto(150,150)
{Any statements, any time }
```

Figure 6.5 Using the Instant window to test procedures from the QuickDraw1 library.

Another procedure, called `DrawLine`, also allows a line to be drawn. `DrawLine` uses the following syntax:

`DrawLine ( x₁, y₁, x₂, y₂ );`

When executed, this command draws a line from a beginning point (x_1, y_1) to an end point (x_2, y_2). After execution, the pen is positioned at the point (x_2, y_2).

Suppose we want to modify the program `Diagonal_Lines` by drawing lines from point (50, 50) to point (150, 50), then to point (150, 150), then to point (50, 150), and finally to point (50, 50). The drawing produced by this program is shown in Figure 6.4.

[2] In THINK Pascal, the Instant window is operational only when a program has been halted in execution. This can be done either by clicking on the spray can at the right side of the menu bar or by inserting stops and activating the **Stops In** command. These restrictions make the use of the Instant window more difficult in the THINK Pascal environment.

```
program Diagonal_Lines_Boxed(input, output);
{ Purpose:  This Macintosh Pascal program executes the library }
{           procedures MoveTo, LineTo, and DrawLine. }
   uses
      QuickDraw1;
begin
{ Move the pen to the center of the Drawing window and draw a }
{ diagonal line. }
   MoveTo(100, 100);
   LineTo(50, 50);
{ Move the pen to the center of the Drawing window and draw a }
{ diagonal line. }
   MoveTo(100, 100);
   LineTo(50, 150);
{ Move the pen to the center of the Drawing window and draw a }
{ diagonal line. }
   MoveTo(100, 100);
   LineTo(150, 50);
{ Move the pen to the center of the Drawing window and draw a }
{ diagonal line. }
   MoveTo(100, 100);
   LineTo(150, 150);
{ Draw a square around the two diagonal lines. }
   DrawLine(50, 50, 150, 50);
   DrawLine(150, 50, 150, 150);
   DrawLine(150, 150, 50, 150);
   DrawLine(50, 150, 50, 50);
end.
```

All *x* and *y* values in `MoveTo`, `LineTo`, and `DrawLine` must be integers or `integer` data types. If either or both of these values are declared or assigned as real values, the program will halt when the procedure is executed and report an error.

The procedure `PenSize` provides the capability of changing the width, the height, or both the width and the height of the line when it is drawn on the screen. The syntax for calling the procedure `PenSize` is as follows:

```
PenSize( width , height );
```

where `width` and `height` are integers or `integer` data types with values greater than zero. Using `real` data types will halt your program. The procedure alters only the appearance of a line and has no effect on the display of any text within the Drawing window. It also has no effect on the position of the drawing pen.

For example, insert the line `PenSize(5,15)` just before the first `DrawLine` command in `Diagonal_Lines_Boxed`, and execute the program. You will observe a heavy line displayed around the two diagonal lines, a line that is 5 units wide and 15 units high. It is important to note that if any value for width or height is equal to or less than zero, the pen assumes zero, and no lines are displayed as they are drawn. For example, if the command

```
PenSize( -5, 15);
```

is used, only the horizontal lines will be drawn, with a height of 15 units. No vertical lines will appear because the argument for width is less than zero. Change the first parameter value to –5 in `PenSize` and observe the effect. The drawing pen can be hidden by executing the command `PenSize(0, 0)`.

The last procedure we will discuss in this section is `WriteDraw`. The `WriteDraw` command displays text to the Drawing window, beginning at the current location of the drawing pen. For this reason it may be necessary to precede the `WriteDraw` command with the `MoveTo` command. For example, the commands for displaying the message `Diagonal_Lines` at the top of the Drawing window in the program `Diagonal_Lines` follow:

```
MoveTo(55, 20);
WriteDraw('Diagonal_Lines');
```

Insert these commands and observe their effect by stepping through the entire program. The `MoveTo` command centers the message over the figure, and the `WriteDraw` command draws the message `Diagonal_Lines` using the standard font.

Like the procedures `LineTo` and `DrawLine`, the `WriteDraw` procedure leaves the drawing pen in a new location after execution. The standard system font characters drawn by `WriteDraw` are 6 or 7 pixels wide and 9 pixels high. By using the option **Font Control** from the menu **Windows**, you can change the type and size of the font for displaying characters in the Drawing window. Select the type of font, click the Program window button, and then click the **OK** button. You can also change the type and size of font for characters displayed in the Text window by using the same steps after clicking the Text window button.

6.3 DRAWING SIMPLE GEOMETRIC PATTERNS

In this section we examine some of the `QuickDraw1` commands for drawing circles, rectangles, and ovals within the boundaries of rectangles. We will also consider commands for painting, inverting, and erasing these geometric patterns. Two procedures for drawing a circle are `PaintCircle` and `InvertCircle`. The procedure `PaintCircle` draws a black circle on the screen for a given radius about a given point. The proper syntax for this command is

```
PaintCircle ( x, y , r );
```

where the point (x , y) represents the center of the circle, and r is the radius from this center. All three values, x , y , and r , must be integers or `integer` data types. The procedure `InvertCircle` draws a black circle if the background within the Drawing window is white and a white circle if the background is black. This procedure is different from procedure `PaintCircle`, which always draws a black circle regardless of the background color. The syntax for `InvertCircle` is as follows:

```
InvertCircle( x, y, r );
```

where x , y , and r are integers or `integer` data types, and the point (x , y) is the center of a circle having a radius r.

You might assume that there is also a procedure named `EraseCircle` used to erase from the screen any circle drawn by `PaintCircle` or `InvertCircle`. Actually there is no such procedure, because the procedure `InvertCircle` can be used to erase a circle drawn by `PaintCircle`. For example, consider the program `Target`, which uses procedures `PaintCircle` and `InvertCircle`. The drawing produced by this program is shown in Figure 6.6.

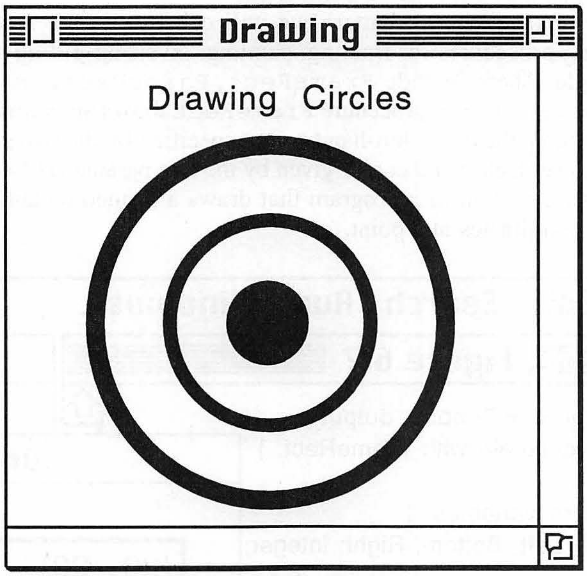


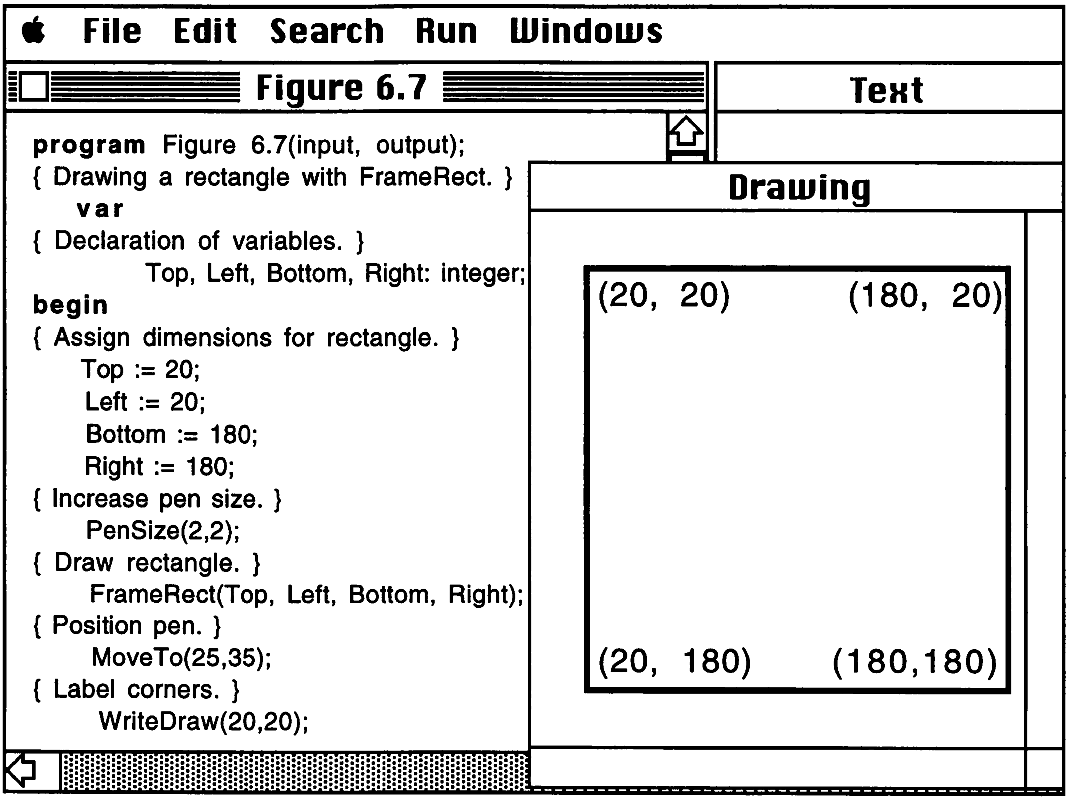
Figure 6.6 The Drawing window, showing the result of executing the program, `Target`.

```
program Target(input, output);
{ Purpose:  This Macintosh Pascal program executes the }
{           procedures PaintCircle and InvertCircle. }

   uses
      QuickDraw1; { Remove uses clause for THINK Pascal. }

begin
{ Place a title in the Drawing window. }
   MoveTo(50, 18);
   WriteDraw(' Drawing Circles');
{ Paint a circle of radius 70 with the center located at point
(100,100). }
   PaintCircle(100, 100, 70);
{ Erase part of this circle using the procedure InvertCircle. }
   InvertCircle(100, 100, 65);
{ Paint a new circle of radius 40. }
   PaintCircle(100, 100, 40);
{ Create a ring by using InvertCircle. }
   InvertCircle(100, 100, 35);
```

```
{ Place a small circle with a radius of 15 located in the }
{ center of the ring. }
   InvertCircle(100, 100, 15);
end.
```

Neither `PaintCircle` nor `InvertCircle` has any effect on the location of the drawing pen after execution is completed.

The next geometric pattern we will consider is the rectangle. There are several interesting procedures for framing, painting, inverting, filling with a pattern, and erasing a rectangle. These include `FrameRect`, `PaintRect`, `InvertRect`, `FillRect`, and `EraseRect`. The procedure `FrameRect` draws an outline of a rectangle, given two special points: the upper left-hand corner specified by the two parameters `Top` and `Left`, and the lower right-hand corner given by the two parameters `Bottom` and `Right`. Figure 6.7 shows a portion of a program that draws a framed rectangle and labels each corner with the coordinates of a point.



```
program Figure 6.7(input, output);
{ Drawing a rectangle with FrameRect. }
  var
{ Declaration of variables. }
        Top, Left, Bottom, Right: integer;
begin
{ Assign dimensions for rectangle. }
    Top := 20;
    Left := 20;
    Bottom := 180;
    Right := 180;
{ Increase pen size. }
    PenSize(2,2);
{ Draw rectangle. }
    FrameRect(Top, Left, Bottom, Right);
{ Position pen. }
    MoveTo(25,35);
{ Label corners. }
     WriteDraw(20,20);
```

Figure 6.7 Drawing a rectangle with `FrameRect`.

The syntax required for the procedure `FrameRect` follows:

```
FrameRect( Top, Left, Bottom, Right );
```

where Top, Left, Bottom, and Right are integers or integer data types. Later in this chapter we will show how to insert these four values into a special data type defined in the QuickDraw1 library as type Rect (short for *rectangle*).

The procedure PaintRect allows a rectangle specified by Top, Left, Bottom, and Right to be painted using an existing pattern. The syntax required for the procedure PaintRect follows:

```
PaintRect( Top, Left, Bottom, Right );
```

The procedure InvertRect allows the pixels enclosed by a rectangle specified by Top, Left, Bottom, and Right to become black if drawn on a white background or white if drawn on a black background. The syntax required to call the procedure InvertRect follows:

```
InvertRect( Top, Left, Bottom, Right );
```

The procedure FillRect can be used to fill a rectangle specified by Top, Left, Bottom, and Right with a predefined pattern: *white*, *black*, *gray*, *ltgray*, or *dkgray.* [3] The syntax required for calling on FillRect follows:

```
FillRect( Top, Left, Bottom, Right, Pat );
```

where Pat is a data type defined by the QuickDraw1 library to be of type Pattern; pat must have one of five possible values: *white*, *black*, *gray*, *ltgray*, or *dkgray.*

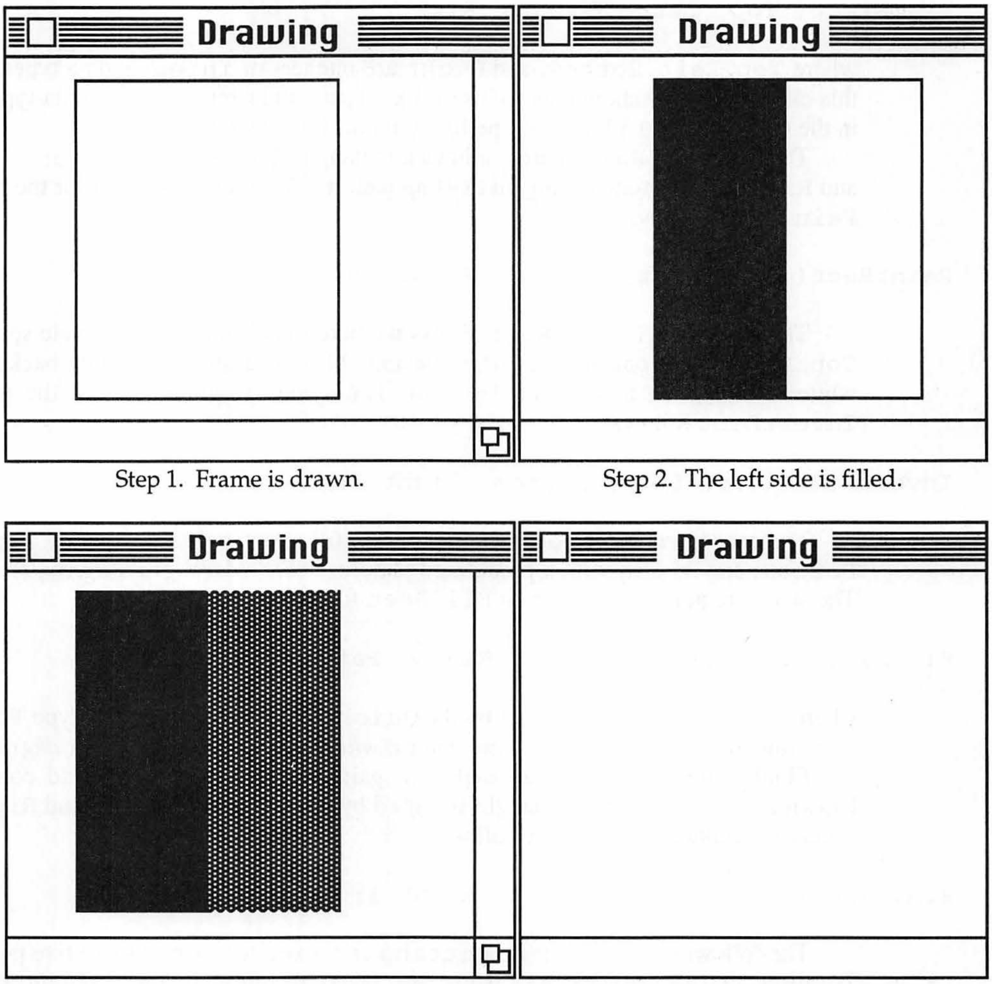
Finally, the procedure EraseRect paints the current background color of the Drawing window within a rectangle specified by Top, Left, Bottom, and Right. The syntax for calling EraseRect follows:

```
EraseRect( Top, Left, Bottom, Right );
```

The following program, titled Rectangles, executes each of these five procedures. To allow you to observe the results, we inserted delays in the program after each procedure. The major stages of the drawing produced by Rectangles are shown in Figure 6.8.

```
program Rectangles(input, output);
{ Purpose:  Test procedures FrameRect, PaintRect, InvertRect, }
{           FillRect, and EraseRect. }
   uses
      QuickDraw1;
   const
      Delay = 50000;
{ Program listing continues after Figure 6.8 }
```

[3] When using QuickDraw2, eight standard colors called*blackcolor*,*whitecolor*, *redcolor*, *greencolor*,*bluecolor*, *cyancolor*, *magentacolor*, and*yellowcolor* are available for color monitors, using the procedures ForeColor and BackColor.



Step 1. Frame is drawn. Step 2. The left side is filled.

Steps 3 and 4. The right side is filled. Step 5. The rectangle is erased.

Figure 6.8 The actions of the program `Rectangles`.

```
{ Continuation of Rectangles listing.}
   var
      Top, Left, Bottom, Right : integer;
      Pat : Pattern;
      I : longint;
begin
{ Assign values to Top, Left, Bottom, and Right and draw an }
{ outline of  a rectangle. }
   Top := 40;
   Left := 50;
   Bottom := 160;
   Right := 150;
   FrameRect(Top, Left, Bottom, Right);
   for I := 1 to Delay do { nothing }
      ;
{ Paint a rectangle in  half of the Drawing window. }
   Right := 100;
   PaintRect(Top, Left, Bottom, Right);
   for I := 1 to Delay do { nothing }
      ;
{ Fill the right part of the rectangle with light gray. }
   Left := 100;
   Right := 150;
   Pat := ltgray;
   FillRect(Top, Left, Bottom, Right, Pat);
   for I := 1 to Delay do { nothing }
      ;
{ Invert the right part of the rectangle. }
   InvertRect(Top, Left, Bottom, Right);
   for I := 1 to Delay do { nothing }
      ;
   Left := 50;
   Right := 150;
   EraseRect(Top, Left, Bottom, Right);
end.
```

As an additional example using `FrameRect`, `PaintCircle`, and `InvertCircle`, consider the Drawing window shown in Figure 6.9. The Macintosh Pascal program `Motion` draws a black circle starting at the left-hand corner of the frame. It then begins to roll this circle along the left edge of the frame until it reaches the bottom left corner, then rolls the circle along the bottom edge to the bottom right corner, then up the right edge to the top right corner, and then along the top edge, leaving the circle in the upper left-hand corner.

```
program Motion(input, output);
{ Purpose:  This Macintosh Pascal program moves a circle around }
{           the interior of a rectangle. }
   uses
      QuickDraw1;
   const
      Radius = 19;
   var
      X, Y : integer;
begin
{ Establish the frame in which the circle will move. }
   FrameRect(50, 50, 220, 220);
{ Establish the circle at the upper left corner of the screen }
{ and let it drop vertically along the inside edge of the frame. }
   X := 70;
{ The program listing continues after Figure 6.9. }
```

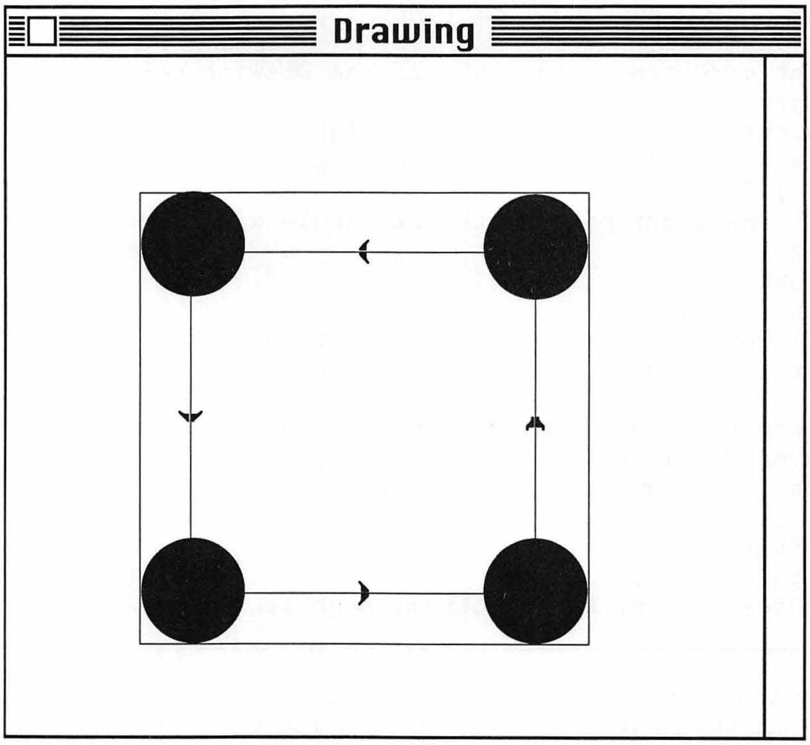


Figure 6.9 Direction in which the program `Motion` moves a circle within a frame.

```
{ Continuation of the listing for the program Motion. }
   for Y := 70 to 200 do
      begin
         PaintCircle(X, Y, Radius);
         InvertCircle(X, Y, Radius);
      end;
{ Move the circle from left to right along the bottom of the }
{ frame. }
   Y := 200;
   X := 70;
   for X := 70 to 200 do
      begin
         PaintCircle(X, Y, Radius);
         InvertCircle(X, Y, Radius);
      end;
{ Raise the circle vertically along the right edge of the frame. }
   Y := 200;
   X := 200;
   for Y := 200 downto 70 do
      begin
         PaintCircle(X, Y, Radius);
         InvertCircle(X, Y, Radius);
   end;
{ Move the circle from right to left along the top of the frame. }
   Y := 70;
   X := 200;
   for X := 200 downto 70 do
      begin
         PaintCircle(X, Y, Radius);
         InvertCircle(X, Y, Radius);
      end;
{ Leave the circle in the upper left-hand corner of the frame. }
   PaintCircle(70, 70, Radius);
end.
```

Notice that the procedure `PaintCircle` is executed before `InvertCircle`. If these two are reversed within each of the **for** loops, a black border is drawn within the boundary of the frame.

In computer graphics, motion can be displayed by rapidly drawing one or more points, erasing these points, and drawing new points. In our example, `PaintCircle` draws all of the points for the black circle while `InvertCircle` erases these points. When the program `Motion` is executed, the circle seems to move slowly, following the boundary of the frame. You may wonder if it can be made to execute faster. For Macintosh Pascal, the answer is no; not all of the Macintosh Pascal program lines may be translated into machine code when the program is executed. Part of the Macintosh Pascal program may be examined during execution after the program has been checked for syntax errors. For example, the procedures `PaintCircle` and `InvertCircle` have their arguments checked at execution time (to see if they are properly declared) before performing their stated action. This interpretation increases the execution time, because both procedures are placed in four separate **for** loops, with each loop iterated 130 times. An alternative to using these procedures is to compute all of the points necessary to plot each individual black circle and store all of them in main memory. Some simple calculations, however, reveal the need to compute and store approximately 589,740 individual points before any of the 520 circles can be drawn. Each point requires 4 bytes of storage (2 bytes per integer), so this would require 1,179,479 bytes of memory. This of course exceeds the RAM memory capacity of some Macintosh computers, particularly when we consider the additional memory requirements of the system and application programs.

On the other hand, the THINK Pascal environment offers a faster execution time, because the source code of the program is compiled into machine language prior to execution.

The remaining procedures to be discussed in this section include `FrameOval`, `PaintOval`, `InvertOval`, `FillOval`, and `EraseOval`. The procedure `FrameOval` draws the boundary of an oval within a rectangle specified by `Top`, `Left`, `Bottom`, and `Right`, and `PaintOval` paints an oval within a specified rectangle. `InvertOval` inverts the pixels within an oval to a color opposite that of the background. `EraseOval` paints an oval enclosed within a specified rectangle with the current background color.

Ovals appear elliptic unless the specified rectangles are square, in which case the ovals appear as circles. The syntax required for calling each of these five procedures follows:

```
FrameOval ( Top, Left, Bottom, Right);
PaintOval ( Top, Left, Bottom, Right );
InvertOval ( Top, Left, Bottom, Right);
FillOval (Top, Left, Bottom, Right, pat );
EraseOval ( Top, Left, Bottom, Right);
```

The program `Random_Dots` uses the procedure `FillOval` to place large random dots having one of five different patterns. The program begins by computing random values for the coordinates of the top left corner of a rectangle, then adds 30 to both `Top` and `Bottom` for assigning the coordinates of the bottom right corner. Even though the function `random` can return a random integer value between –32768 and 32767, only values between 0 and 32767 are required. Using the function `abs` (absolute) and **`mod`** division with a divisor 201 limits the resulting values to values between 0 and 200. The second **`mod`** division by 5 results in one of five patterns being selected for the next large dot, while the **`case`** statement allows one of the five patterns to be chosen, depending on the value assigned to `Background`. After the coordinates of both the top left and bottom right corners and the patterns, `Pat`, are assigned values, the procedure `FillOval` is executed to draw an oval within the specified rectangle given by `Top, Left, Bottom, and Right`. In this example, the **`while-do`** command forces the program to execute continuously because its condition always remains *true*. Figure 6.10 shows sample output from the program.

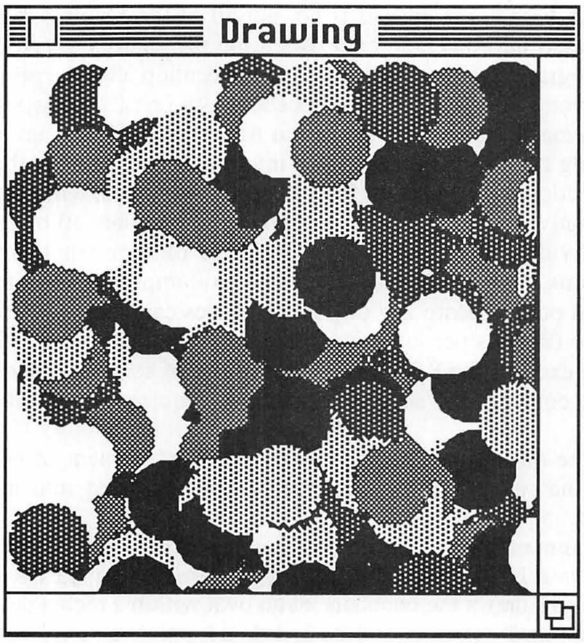


Figure 6.10 Sample results from the program `Random_Dots`.

This program can be halted by choosing the **Halt** option in the **Pause** menu. (Click the spray can to halt the THINK Pascal version.)

```
program Random_Dots(input, output);
{ Purpose:  This Macintosh Pascal program paints random dots }
{           having different patterns in the Drawing window. }

   uses
      QuickDraw1;
   var
      Top, Left, Bottom, Right : integer;
      Pat : Pattern;
      Background : integer;

begin
{ Use the function random to choose the corners of a square }
{ and a pattern. }
   while true do

      begin
      { Randomly select the rectangles for drawing an oval. }
         Top := abs(random) mod 201;
         Left := abs(random) mod 201;
         Bottom := Top + 30;
         Right := Left + 30;
      { Randomly select the background color. }
         Background := abs(random) mod 5;
         case Background of
            0 :  Pat := white;
            1 :  Pat := black;
            2 :  Pat := gray;
            3 :  Pat := ltgray;
            4 :  Pat := dkgray;
         end; { End case }

{ Display the oval in a rectangle with a chosen background }
{ pattern. }
         FillOval(Top, Left, Bottom, Right, Pat);
   end; { End while }
end.
```

None of the procedures discussed in this section has any effect on the location of the drawing pen after being executed. In executing any of the procedures `FrameRect`, `PaintRect`, `FillRect`, `InvertRect`, `EraseRect`, `FrameOval`, `PaintOval`, `InvertOval`, and `FillOval`, the top left point must be above and to the left of the bottom right point for drawing. Drawing occurs in the Drawing window if the following condition holds true:

```
(Top < Bottom  and  Left < Right).
```

If the value of this condition is *false*, nothing is drawn; however, no execution error is reported.

6.4 MOUSE CURSOR COMMANDS

In this section, we present three procedures for allowing input from the mouse. Keep in mind that the Macintosh computer is an event-driven system. That is, it keeps track of all present events initiated by the user and processes each as needed. These events include pressing a key on the keyboard, clicking the mouse button, and dragging the mouse. In Macintosh Pascal programs, commands such as `Button`, `GetMouse`, and `StillDown` provide control through clicking the mouse button or the moving action of the mouse.

Two functions deal specifically with the button on the mouse. The first is called `Button`, which returns a `Boolean` value *true* if the mouse button is being held down at the time that this function is executed, and *false* if the button has been released. Consider the following Macintosh Pascal program, titled `Mouse_Button`.

```
program Mouse_Button(input, output);
{ Purpose:  This program performs a test of the function Button. }
   uses
      QuickDraw1;
   var
       Down : Boolean;
begin
{ Prompt the user with a message. }
   MoveTo(5, 50);
   WriteDraw(' Press mouse button to continue: ');
{ Wait until the mouse button has been pressed. }
   Down := false;
   while not Down do
      Down := Button;
{ Prompt the user with a second message. }
   MoveTo(5, 100);
   WriteDraw(' Continue to keep mouse button pressed: ');
{ Wait until the user has released the mouse button. }
   while Down do
      Down := Button;
{ Prompt the user with a third message. }
   MoveTo(5, 150);
   WriteDraw(' End of execution: ');
end.
```

This example shows how you can display a prompt in the Drawing window. Execution of the program continues after the mouse button has been pressed or clicked. The second loop (the second **while-do** statement) continues to execute until the mouse button is released. You will find it convenient to use both the **Step-Step** and the **Go** options of the **Run** menu to test this program.

The function `StillDown` is different from the function `Button`. After a mouse event has occurred (such as pushing the mouse button down or moving the mouse), the function `StillDown` will check to see if the mouse button remains down; it will return the value *true* if the button is currently down and no other mouse events have occurred. Otherwise, it will return the value *false*. Moving the mouse while the button remains down is interpreted to mean that another event has occurred. For example, if the mouse button is pressed, released, and then pressed again, the function `Button` will return the value *true*, but the function `StillDown` returns the value *false*. Why? Pressing the mouse button a second time means that a second mouse event has occurred. In order for `StillDown` to return the value *true*, the button must remain down.

The procedure `GetMouse` locates the current position of the mouse in the Drawing window and returns this point through two parameters, the first for the horizontal coordinate and the second for the vertical coordinate.[4] The syntax for calling on this procedure is as follows:

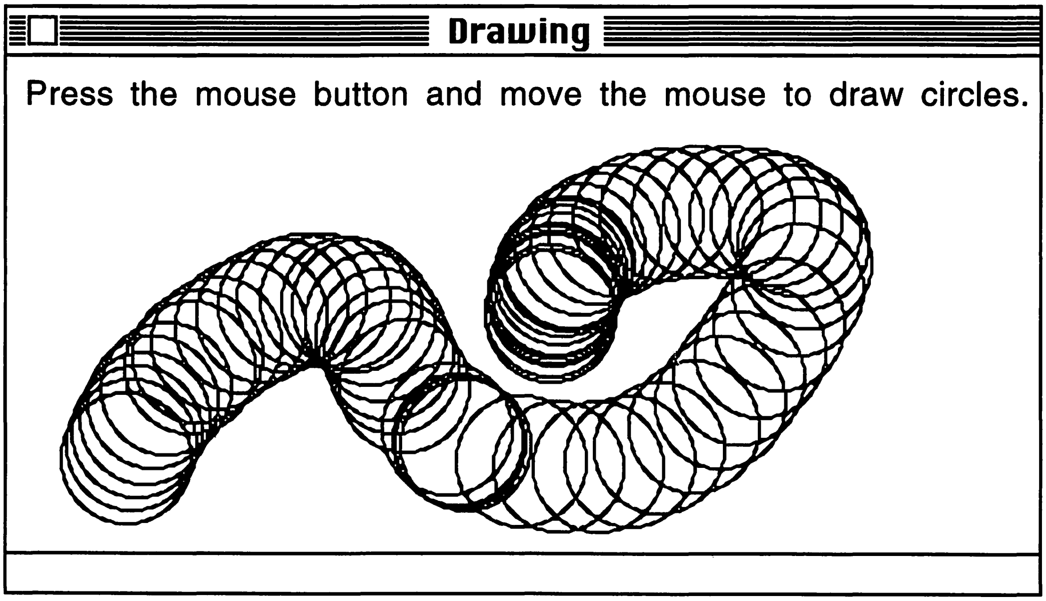
```
GetMouse(p);
```

where p is of data type `point`. The following THINK Pascal program, titled `Rings`, shows how to display rings in the Drawing window using the command `GetMouse`.

```
program Rings(input, output);
{ Purpose:  This program tests the GetMouse procedure by drawing }
{           inverted circles of a fixed radius. }
   const
      Number = 1000;
 var
      J: integer;
      Screen_Point: Point;
      Message: string;
begin
   ShowDrawing;
{ Prompt the user with instructions. }
   MoveTo(5, 15);
   Message := 'Press the mouse button and move the mouse to draw
                  circles:';
   WriteDraw(Message);
   while true do
      begin
      { Return the mouse point. }
         GetMouse(Screen_Point);
      { If mouse button is pressed, display an inverted circle.}
         if Button then
            begin
               FrameOval(Screen_Point.v - 25, Screen_Point.h - 25,
               Screen_Point.v + 25, Screen_Point.h + 25);
            { Provide a short delay while releasing mouse }
            { button. }
               for J := 1 to Number do {nothing}
                  ;
            end;
      end;
end.
```

[4] The `GetMouse` command is handled differently in Macintosh Pascal. A single argument of type `point` is used to communicate the location of the mouse. This will be explained shortly.

Notice that the coordinate for the center of each circle is known only when the mouse button is pressed. If the button is not pressed, the program remains in a simple loop waiting for the function `Button` to return a value of *true*. You will observe something different if you remove the short delay from the **`if-then`** statement. This delay loop is necessary so that you will have enough time to release the mouse button.[5] A drawing produced by this program is shown in Figure 6.11



Press the mouse button and move the mouse to draw circles.

Figure 6.11 A drawing created with the program `Rings`.

The `GetMouse` command is different for Macintosh Pascal. To execute the above program under Macintosh Pascal you would have to change the `GetMouse` statement as follows:

```
GetMouse(Screen_Point.h, Screen_Point.v);
```

where `Screen_Point.h` and `Screen_Point.v` are `integer` values representing the X,Y coordinates on the screen. Therefore you would also need to change the variable declaration part of the program, as shown below.

```
var
   J: integer;
   Screen_Point.h, Screen_Point.v: integer;
   Message: string;
```

[5] In THINK Pascal, a procedure called `Delay` is available for the purpose of delaying the execution of a program. This routine will be described in more detail in Section 7.7.

6.5 SETTING THE SIZE AND DISPLAY OF TEXT AND DRAWING WINDOWS

As you may have seen with the desktop folder and other Macintosh software, you can remove a window on the screen by clicking the close box. You can also expand the size of a window by moving the cursor to the size box of the window, pressing the mouse button, and dragging the mouse until you have adjusted the window to the desired size. By using special Macintosh Pascal procedures from the `QuickDraw1` library, you can automate your Pascal programs to hide and set windows without the need for dragging and clicking the mouse. The first of these procedures is `HideAll`. When executed, this procedure causes windows on the screen to be hidden. Remember that all hidden Pascal windows can be revealed by using the Windows menu. Once the windows are hidden, you can display the Text and Drawing windows by using the following pairs of commands: `SetTextRect` and `ShowText` or `SetDrawingRect` and `ShowDrawing`. The procedures `SetTextRect` and `SetDrawingRect` allow you to set the size and location of either the Text or the Drawing window, but they do not open the window for display. Each procedure requires only one actual parameter, a variable of data type `Rect`.

Chapter 9 discusses record structures in greater detail, but let us at this point show how the `QuickDraw1` library declares the special record structure called `Rect`. In Pascal a record structure allows us to establish a type having several different properties. `Rect` is a special type declared as follows:

```
Rect   =   record case integer of
              0 :  (   top :       integer;
                       left:       integer;
                       bottom:     integer;
                       right:      integer );
              1 :  (   topLeft :   point;
                       botRight :  point );
           end;
```

For this record structure, a variable declared of type `Rect` can either have the properties associated with label 0 or label 1. The choice of labels is determined by use of an assignment statement. For example, the program Partial declares `Box` to be of type `Rect`. The statements

```
Box.top := 0;
Box.left := 0;
Box.bottom := 342;
Box.right := 512;
```

assign four separate values to `Box`, each represented by a variable name followed by a period followed by a unique field name such as `top`, `left`, `bottom`, or `right`. Thus in this program the variable `Box` is using the record structure represented by label 0. It is through the execution of the procedure `SetDrawingRect` that the boundaries of the Drawing window are established using the four values of variable `Box`. In this example, all windows are first hidden by executing the procedure `HideAll`. Next, the complete screen, other than the menu bar at the top, is captured for the Drawing window. Remember that for maximum size, the Drawing window requires an upper left corner with the coordinate (0, 0) and a lower right corner with the coordinate (512, 342).

```
program Partial(input, output);
{ Purpose:  Demosntration of the procedure SetDrawingRect. }
   uses
      QuickDraw1;
   var
      Box : Rect;
begin
{ Hide all windows. }
   HideAll;
{ Set the boundary of the Drawing window as the complete screen. }
   Box.top := 0;
   Box.left := 0;
   Box.bottom := 342;
   Box.right := 512;
   SetDrawingRect( Box );
{ Show the new Drawing window. }
   ShowDrawing;
{ Continue with the remainder of the program. }
end.
```

The procedure `ShowDrawing` requires no parameters and, when executed, opens the Drawing window.

There is an easier way to establish the boundaries of a rectangle for viewing a window. The procedure called `SetRect` allows a rectangle of type `Rect` to be assigned the top left and bottom right points of a rectangle. Syntax for using this procedure follows:

```
SetRect( Rect_Window, Left, Top, Right, Bottom );
```

where `Left`, `Top`, `Right`, `Bottom` are integers or `integer` data types. Be sure you understand that the procedure `SetRect` requires the top left and bottom right points to be specified by the list `Left`, `Top`, `Right`, `Bottom`. This is different from our earlier drawing procedures, which required top left and bottom right to be specified by the list `Top`, `Left`, `Bottom`, `Right`. We can modify our example with the following lines:

```
program Partial_Revised(input, output);
{ Purpose:  Demosntration of the procedure SetDrawingRect. }
   uses
      QuickDraw1;
   var
      Box : Rect;
begin
{ Hide all windows. }
   HideAll;
{ Set the boundary of the Drawing window as the complete screen. }
   SetRect( Box, 0 , 0, 512, 342 );
   SetDrawingRect( Box );
{ Show the new Drawing window. }
   ShowDrawing;
{ Continue with the remainder of the program. }
```

Suppose you want to split the screen into two parts. The left half of the screen will be the Drawing window, and the right half will be the Text window. Adding the following statements will result in the Text window being shown:

```
{ Modify only the top and left values of Box. }
   SetRect( Box, 256, 18, 512, 342 );
   SetTextRect( Box );
{ Show the new Text window. }
     ShowText;
```

The value Top is set at 18 rather than zero in order to prevent the menu bar from hiding any text. Left is set to 256 because this value represents the center of the screen as you move from left to right along a horizontal axis.

The procedures GetTextRect and GetDrawingRect return the coordinates of the Text and Drawing windows, respectively. The procedure GetTextRect (Window_Rectangle) returns the current size and present position of the Text window, and GetDrawingRect(Window_Rectangle) returns the current size and present position of the Drawing window. In either case, the variable Window_Rectangle must be of type Rect. For example, during execution, if you move the Text window by clicking on the the size box or by dragging on the menu bar with the mouse, execution of

```
GetRectText( Text_Window )
```

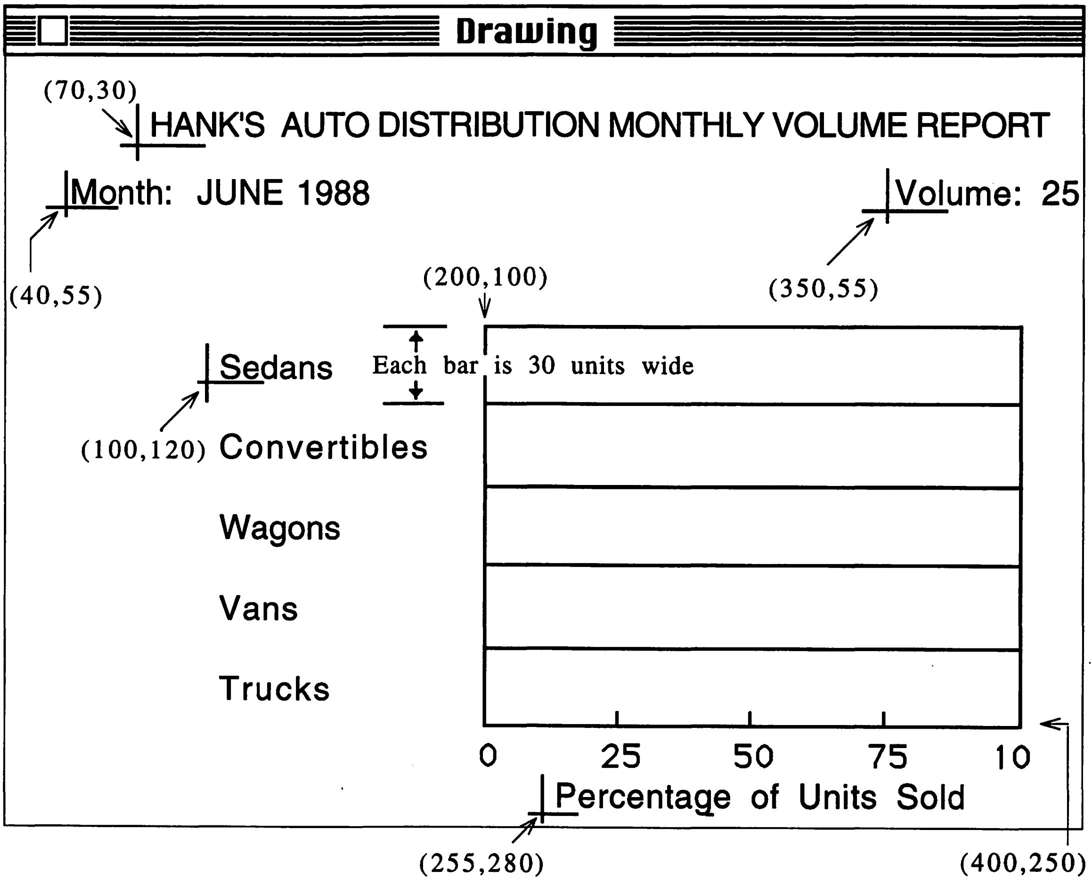
would result in the coordinates of Text_Window being changed. This applies to the procedure GetDrawingRect as well. Remember that during the execution of a Macintosh program, these two procedures can be useful for recording any changes to the coordinates of Text and Drawing rectangles.

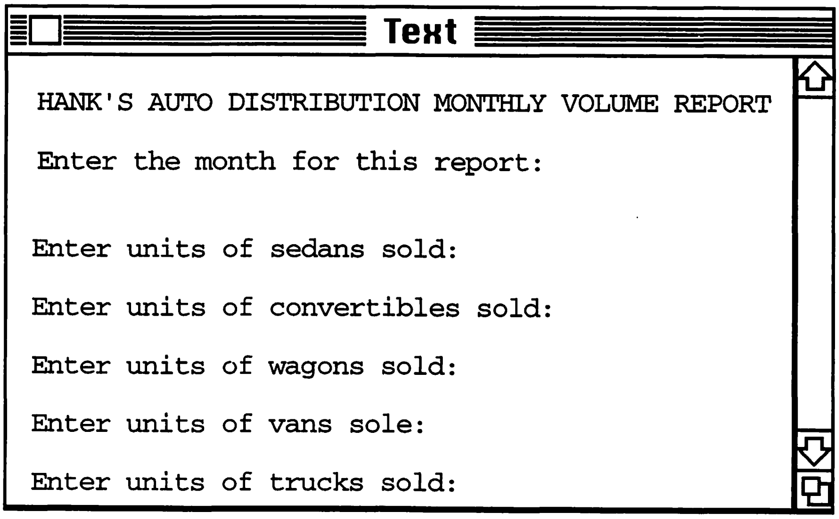
6.6 SOME APPLICATIONS OF THE QUICKDRAW1 LIBRARY

Let us illustrate the application of the QuickDraw1 library by writing a program that draws a bar chart, using data entered from the keyboard. Consider the case of Hank's Auto Distribution Center. Hank wants to employ a Macintosh computer to display a bar chart reporting the percentage of vehicles sold and the total number sold. Figure 6.12 shows the formats for both the Text and Drawing windows. The coordinates for displaying messages and selecting the frame of the bar charts may need to be determined by trial and error. This requires writing parts of the Macintosh Pascal program and then testing them by executing only those procedures used for opening and displaying windows and messages. You may need to repeat this until you are satisfied with the appearance of all windows.

Our algorithm requires six inputs from the person using the program:

```
Month (year can also be included)
Number of sedans
Number of convertibles
Number of wagons
Number of vans
Number of trucks
```





```
HANK'S AUTO DISTRIBUTION MONTHLY VOLUME REPORT

Enter the month for this report:


Enter units of sedans sold:

Enter units of convertibles sold:

Enter units of wagons sold:

Enter units of vans sole:

Enter units of trucks sold:
```

Figure 6.12 Layout for the Drawing and Text windows in the program `Bar_Chart`.

Output will be a bar chart in the grid shown in Figure 6.12. When executed, the algorithm will compute six values:

```
Total volume (units) sold
Percentage of sedans sold
Percentage of convertibles sold
Percentage of wagons sold
Percentage of vans sold
Percentage of trucks sold
```

Here are the initial steps in the algorithm for drawing the bar chart:

1. Hide all windows.
2. Establish boundaries for both the Text and Drawing windows.
3. Open the Text window.
4. Prompt the user for and accept input data.
5. Compute the total volume and all required percentages.
6. Hide the Text window, and open the Drawing window.
7. Draw the title, and then label and draw the bar chart.

The expanded algorithm follows. Some detail, such as the use of `writeln`, `readln`, and `MoveTo` commands, has been left for the program listing.

```
Algorithm Bar_Chart;
{ This algorithm draws a horizontal bar chart for five data
  items. }
begin
{ Hide all windows. }
   HideAll;
{ Establish boundaries for the Text and Drawing windows. }
   SetRect( Text_Window, 100, 100, 400, 300);
   SetRect(Drawing_Window, 0, 40, 512, 342);
   SetTextRect(Text_Window);
   SetDrawingRect(Drawing_Window);
{ Open the Text window. }
   ShowText;
{ Prompt the user for initial data. }
   write(' HANK`S AUTO DISTRIBUTION MONTHLY VOLUME REPORT');
   write(' Enter the month for the report: ');  Read ( Month );
   write(' Enter units of sedans sold: ');  Read (Sedans);
   write(' Enter units of convertibles sold: ');  Read
(Convertibles);
   write(' Enter units of wagons sold: ');  Read (Wagons);
   write(' Enter units of vans sold: ');  Read (Vans);
   write(' Enter units of trucks sold: ');  Read (Trucks);
{ Compute the total volume and all percentages. }
   if ( Total_Volume <> 0 ) then
      begin
         Total_Volume <-- Sedans + Convertibles + Wagons +
                          Vans + Trucks;
```

```
            Percentage_Sedans <-- round( Sedans/Total_Volume * 100 );
            Percentage_Convertibles <--
            round( Convertibles/Total_Volume * 100 );
            Percentage_Wagons <-- round( Wagons/Total_Volume * 100 );
            Percentage_Vans <-- round( Vans/Total_Volume * 100 );
            Percentage_Trucks <-- round( Trucks/Total_Volume * 100 );
         end;
{ Close the Text window and open the Drawing window. }
   HideAll;  ShowDrawing;
{ Draw the bar chart. }
{ Display the title, month, and total volume. }
   WriteDraw(' HANK`S AUTO DISTRIBUTION MONTHLY VOLUME REPORT;);
   WriteDraw(' Month:  ', Month );
   WriteDraw(' Volume: ', Total_Volume );
{ Display the frame for the bar chart. }
   FrameRect(100, 200, 250,4 00);
{ Label the vertical axis of the bar chart. }
   WriteDraw( 'Sedans' );
   WriteDraw( 'Convertibles' );
   WriteDraw( 'Wagons' );
   WriteDraw( 'Vans' );
   WriteDraw( 'Trucks' );
{ Scale the horizontal axis. }
   Scale <-- 0;
   X <-- 170;
   Y <-- 265;
   for Mark <-- 1 to 5 do
      begin
      { Move to X,Y point }
         WriteDraw( Scale );
         DrawLine( X + 30, Y - 16, X + 30, Y - 21 );
         Scale <-- Scale + 25;
         X <-- X + 50;
      end; { for-loop }
{ Label the horizontal axis. }
   WriteDraw(' Percentage of Units Sold');
{ Display each of the five percentage bars. }
   FillRect(100, 200,1 30, 200 + Percentage_Sedans * 2, black );
   FillRect(130, 200, 160, 200 + Percentage_Convertibles * 2,
               gray);
   FillRect(160, 200, 190, 200 + Percentage_Wagons * 2, black );
   FillRect(190, 200, 220, 200 + Percentage_Vans * 2, ltgray );
   FillRect(220, 200, 250, 200 + Percentage_Trucks * 2, dkgray );
end.
```

First, all variables other than `Month` are `integer`. `Month` will be declared as a `string` type with a maximum length of 20 characters. Second, the percentages are computed first as real values, scaled by a factor of 100, and then rounded up to an integer number. To avoid division by zero, the variable `Total_Volume` is tested before any percentages are computed. We assume that all variables are initially zero before execution begins. If not, the following conditional statement would be required:

```
If ( Total_Volume <> 0 ) then
   begin
      Total_Volume <-- Sedans + Convertibles + Wagons +
                                            Vans + Trucks;
      Percentage_Sedans <-- round( Sedans/Total_Volume * 100 );
      Percentage_Convertibles <--
                     round( Convertibles/Total_Volume * 100 );
      Percentage_Wagons <-- round( Wagons/Total_Volume * 100 );
      Percentage_Vans <-- round( Vans/Total_Volume * 100 );
      Percentage_Trucks <-- round( Trucks/Total_Volume * 100 );
   end
else
   begin
      Total_Volume <-- 0;
      Percentage_Sedans <-- 0;
      Percentage_Convertibles <-- 0;
      Percentage_Wagons <-- 0;
      Percentage_Vans <-- 0;
      Percentage_Trucks <-- 0;
   end;
```

Third, each pixel in the bar chart represents one-half of a percentage point. Because of this, we must multiply each of the corresponding percentages by a factor of 2 before adding it to 200 when computing the bottom right point of the rectangle being filled by a pattern using the procedure `FillRect`. Each bar is to be shown using a different pattern.

The Macintosh Pascal program `Bar_Chart` is based on this algorithm. This program includes added `writeln` and `MoveTo` statements to show the positions of the text cursor and drawing pen.

```
program Bar_Chart(input, output);
{ Purpose:  This program draws a horizontal bar chart for five }
{           data items. }

    uses
      QuickDraw1;
    var
      Text_Window, Drawing_Window : Rect;
      Sedans, Convertibles, Wagons, Vans, Trucks : integer;
      Percentage_Sedans, Percentage_Convertibles : integer;
      Percentage_Wagons, Percentage_Vans :integer;
      Total_Volume, Percentage_Trucks: integer;
      Month : string[20];
      Scale, Mark, X, Y : integer;

begin
{ Hide all windows. }
   HideAll;
{ Establish boundaries for both the Text and Drawing windows. }
   SetRect(Text_Window, 100, 100, 400, 300);
   SetRect(Drawing_Window, 0, 40, 512, 342);
   SetTextRect(Text_Window);
   SetDrawingRect(Drawing_Window);
{ Open the Text window and prompt the user for initial data. }
   ShowText;
   writeln;
   writeln(' HANK`S AUTO DISTRIBUTION MONTHLY VOLUME REPORT ');
   writeln;
   write(' Enter the month for this report: ');
   readln(Month);
   writeln;
   writeln;
   write(' Enter units of sedans sold: ');
   readln(Sedans);
   writeln;
   write(' Enter units of convertibles sold: ');
   readln(Convertibles);
   writeln;
   write(' Enter units of wagons sold: ');
   readln(Wagons);
   writeln;
   write(' Enter units of vans sold: ');
   readln(Vans);
   writeln;
   write(' Enter units of trucks sold: ');
   readln(Trucks);
{ Compute the total volume and all relevant percentages. }
   Total_Volume := Sedans + Convertibles + Wagons + Vans + Trucks;
   if Total_Volume <> 0 then

      begin
         Percentage_Sedans := round(Sedans / Total_Volume * 100);
         Percentage_Convertibles := round(Convertibles /
                                          Total_Volume * 100);
         Percentage_Wagons := round(Wagons / Total_Volume * 100);
         Percentage_Vans := round(Vans / Total_Volume * 100);
         Percentage_Trucks := round(Trucks / Total_Volume * 100);
      end;

{ Close the Text window and open the Drawing window. }
   HideAll;
   ShowDrawing;
{ Display the title, month, and total volume of units sold. }
   MoveTo(70, 30);
   WriteDraw(' HANK`S AUTO DISTRIBUTION MONTHLY VOLUME REPORT');
   MoveTo(40, 55);
   WriteDraw(' Month:  ', Month);
   MoveTo(350, 55);
   WriteDraw(' Volume: ', Total_Volume);
{ Display the frame for the bar chart. }
   FrameRect(100, 200, 250, 402);
{ Labels the vertical axis of the bar chart. }
```

```
   MoveTo(100, 120);
   WriteDraw('Sedans');
   MoveTo(100, 150);
   WriteDraw('Convertibles');
   MoveTo(100, 180);
   WriteDraw('Wagons');
   MoveTo(100, 210);
   WriteDraw('Vans');
   MoveTo(100, 240);
   WriteDraw('Trucks');
{ Scale the horizontal axis of the bar chart. }
   Scale := 0;
   X := 170;
   Y := 265;
   for Mark := 1 to 5 do

      begin
         MoveTo(X, Y);
         WriteDraw(Scale);
         DrawLine(X + 30, Y - 16, X + 30, Y - 21);
         Scale := Scale + 25;
         X := X + 50;
      end;

{ Label the horizontal axis of the bar chart. }
   MoveTo(225, 280);
   WriteDraw(' Percentage of Units Sold');
{ Display each of the five percentage bars. }
   FillRect(100, 200, 130, 200 + Percentage_Sedans * 2, black);
   FillRect(130, 200, 160, 200 + Percentage_Convertibles * 2,
            gray);
   FillRect(160, 200, 190, 200 + Percentage_Wagons * 2, black);
   FillRect(190, 200, 220, 200 + Percentage_Vans * 2, ltgray);
   FillRect(220, 200, 250, 200 + Percentage_Trucks * 2, dkgray);
end.
```

Figure 6.13 shows an examp. of the output displayed by the program `Bar_Chart`. The style and size of the font displayed in the Drawing window can be changed by selecting a different font, using the option **Font Control** from the **Windows** menu.

As a second example, consider a Macintosh Program for drawing one or more directed lines. The person using this program will be prompted to select an initial point within the Drawing window by first moving the cursor with the mouse. Once an initial point is selected, the mouse button is pressed and held down until a second point is selected. Once the second point is selected and the mouse button is released, a line is drawn from the initial point to the second point. The program continues to execute, allowing other direct lines to be drawn, until the option **Halt** is chosen from the **Pause** menu.

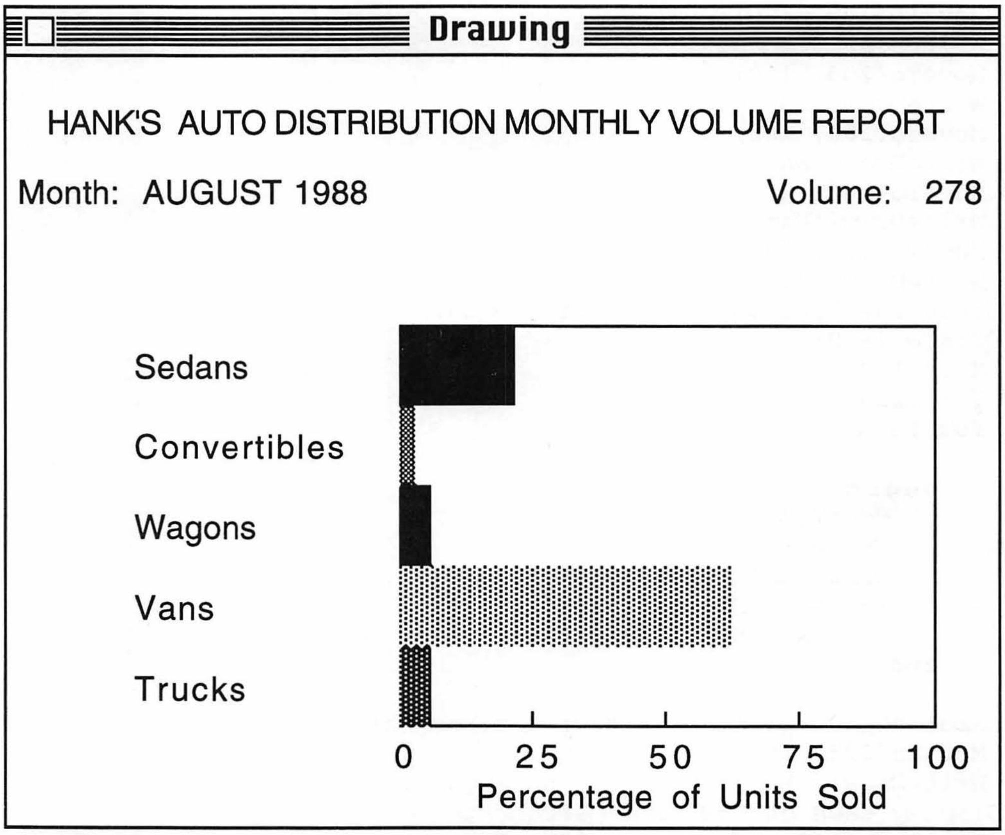
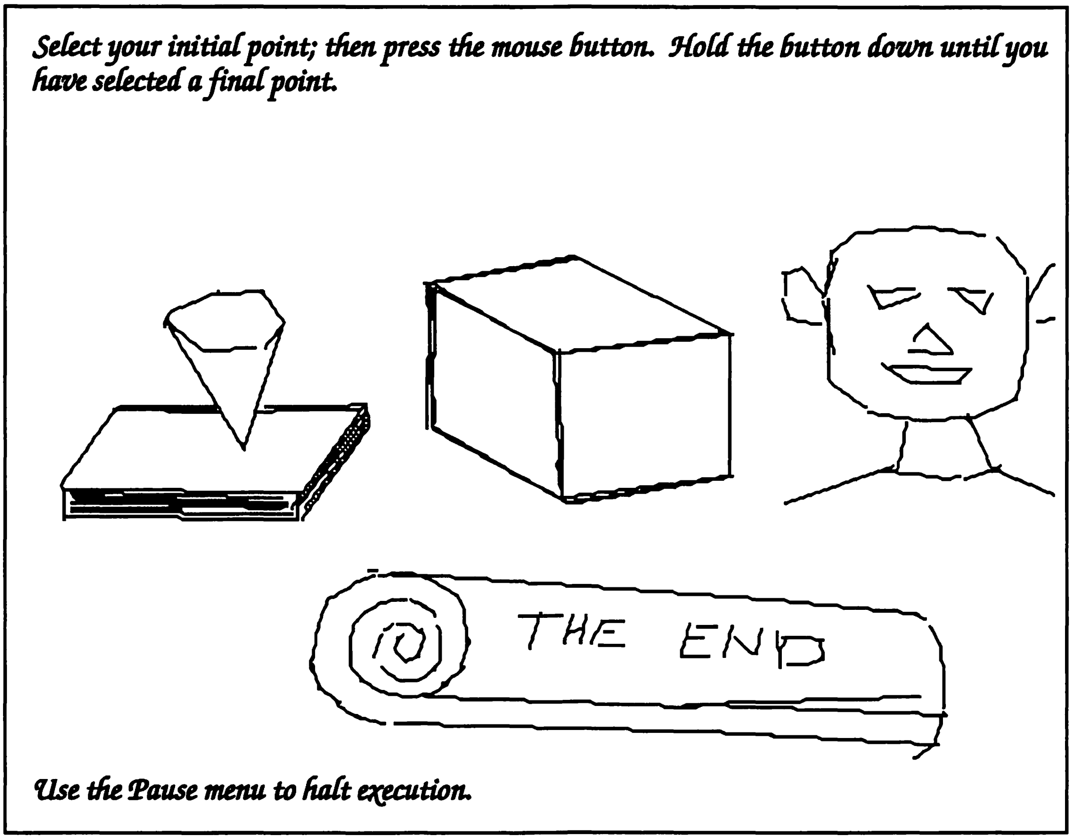


Figure 6.13 Sample output from the program `Bar_Chart`.

Figure 6.14 shows an example of the output created by this program. When drawing the text characters with `WriteDraw`, the font type selected was Venice, with a font size of 14. Below is an algorithm for this program.

```
Algorithm Directed_Lines;
{ This algorithm allows one or more lines to be drawn. }
begin
{ Hide all windows. }
   HideAll;
{ Establish boundary for the Drawing window using SetRect and }
{ SetDrawingRect. }
   SetRect(Drawing_Window, 0, 18, 512, 342 );
   SetDrawingRect( Drawing_Window );
{ Open the Drawing window. }
   ShowDrawing;
{ Prompt the user for selecting an initial point. }
{ Algorithm continues after Figure 6.14. }
```


Select your initial point; then press the mouse button. Hold the button down until you have selected a final point.

Use the Pause menu to halt execution.

Figure 6.14 Using the procedures `GetMouse` and `DrawLine` and the function `Button`, we can write a simple sketching program.

```
{ Continuation of algorithm for Directed_Lines }
   WriteDraw('Select your initial point; then press the mouse
               button.');
   WriteDraw('Hold the mouse button down until you have selected a
               final point.');
   WriteDraw('Use the Pause menu to halt execution.');
{ Repeat drawing lines between two points until option Halt is
     chosen. }
   repeat
      Down <-- false;
   { Wait for the mouse button to be pressed. }
      while { button is not down } do
         Down <-- Button;
{ Get the initial point X1, Y1.}
      GetMouse( X1, Y1 );
   { Wait for the mouse button to be released. }
      while { mouse button is down } do
         Down <-- Button;
   { Get the final point X2, Y2.}
      GetMouse( X2, Y2 );
   { Draw a line from the initial point to the final point. }
      DrawLine( X1, Y1, X2, Y2 );
   until { selected option Halt from Pause menu }
end.
```

Following is the Macintosh Pascal program `Directed_Lines`:

```
program Directed_Lines(input, output);
{ Purpose: This program allows one or more lines to be drawn. }
   uses
      QuickDraw1;
   const
      Halt = false;
   var
      Drawing_Window : Rect;
      X1, Y1, X2, Y2 : integer;
      Down : Boolean;
begin
{ Hide all windows. }
   HideAll;
{ Establish boundary for the Drawing window using SetRect and }
{ SetDrawingRect. }
   SetRect(Drawing_Window, 0, 18, 512, 342 );
   SetDrawingRect( Drawing_Window );
{ Open the Drawing window. }
   ShowDrawing;
{ Prompt the user for selecting an initial point. }
   MoveTo(20, 20);
   WriteDraw('Select your initial point; then press the mouse
               button.');
   MoveTo(20, 35);
   WriteDraw('Hold the mouse button down until' );
   WriteDraw(' you have selected a final point.');
   MoveTo(20, 300);
   WriteDraw('Use the Pause menu to halt execution.');
{ Repeat drawing lines between two points until option Halt is }
{ chosen. }
   repeat
      Down := false;
   { Wait for the mouse button to be pressed. }
      while not Down do
         Down := Button;
   { Get the initial point X1, Y1.}
         GetMouse( X1, Y1 );
   { Wait for the mouse button to be released. }
         while Down do
            Down := Button;
   { Get the final point X2, Y2.}
         GetMouse( X2, Y2 );
   { Draw a line from the initial point to the final point. }
         DrawLine( X1, Y1, X2, Y2 );
   until Halt;
end.
```

A Boolean constant called Halt is used as the condition of the **repeat-until** loop. This same loop can be replaced with a **while-do** loop having the following form:

```
while not Halt do
   begin
      Down := false;
{ Wait for the mouse button to be pressed. }
   while not Down do
      Down := Button;
{ Get the initial point X1, Y1.}
   GetMouse( X1, Y1 );
{ Wait for the mouse button to be released. }
   while Down do
      Down := Button;
{ Get the final point X2, Y2.}
   GetMouse( X2, Y2 );
{ Draw a line from the initial point to the final point. }
   DrawLine( X1, Y1, X2, Y2 );
end;
```

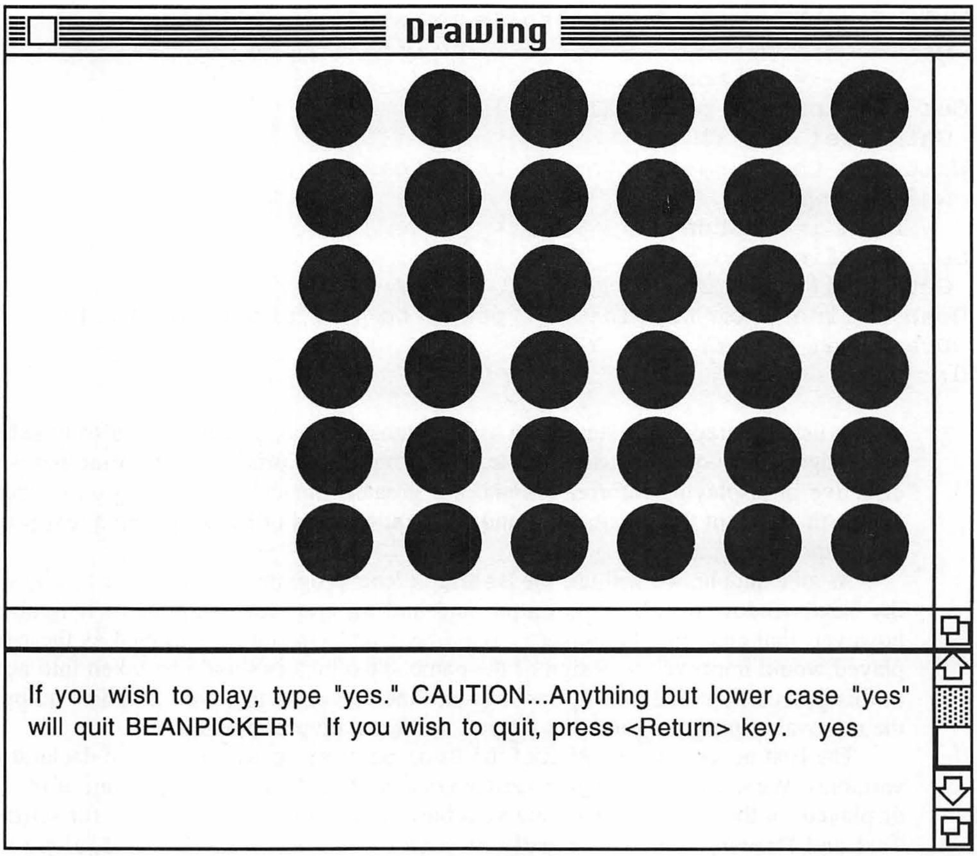
In using a graphics system such as Macintosh Pascal, you may want to transform a text program into one that uses both text and graphics to produce a program that is more effective in displaying information and has greater impact. This discussion encourages you to think about the possibilities and the requirements of transforming a text program to a graphics program.

As an example, we will use the Beanpicker program from Chapter 5, which used the Text window for all of its output and had no graphics component. It is obvious, however, that enabling the player to see a bean or beans being removed as the game is played would improve the design of the game. Two factors should be taken into account in this process: the addition of the program lines required to create graphics output and the removal of program lines that support the text only presentation.

The first necessary alteration of the Beanpicker program is in the declaration of variables. We need an integer variable called Max_Bean to keep count of the beans displayed on the screen. Three new variables of type Rect are required for setting the Text and Drawing windows. Finally, two integer variables, X and Y, along with a constant, Radius (which has a value of 14), will be needed to locate and draw the beans. At the beginning of the execution of the revised program, both the Text window and the Drawing window will be activated. These changes are achieved by the following program lines:

```
const
   Radius = 14;
var
   Number_of_Beans, Player_Pick, MAC_Pick : integer;
   X, Y, Index, Max_Bean : integer;
   Continuation : string[80];
   Box1, Box2, Box3 : Rect;
begin
   SetRect(Box1, 1, 40, 512, 345);
   SetRect(Box2, 1, 40, 512, 260);
   SetRect(Box3, 1, 260, 512, 335);
   SetTextRect(Box1);
   ShowText;
```

As shown in Figure 6.15, the introduction to the program and the rules of the game are displayed in the Text window.



If you wish to play, type "yes." CAUTION...Anything but lower case "yes" will quit BEANPICKER! If you wish to quit, press <Return> key. yes

Figure 6.15 Opening screen for the `Modified_Beanpicker` Game.

If the player chooses to continue, 36 black circles representing 36 beans will be drawn in the Drawing window through the execution of the following program lines:

```
while Continuation = 'yes' do
   begin
   { Set Text and Drawing windows. }
      SetTextRect(Box3);
      SetDrawingRect(Box2);
   { Initialize variables. }
      X := 138;  { X is the coordinate of first bean.}
      Y := 16;    { Y is the coordinate of first bean.}
      Max_Bean := 1;
   { Open Drawing window for viewing beans. }
      ShowDrawing;
      while Max_Bean <= 36 do
         begin
         { Display 36 black beans. }
            PaintCircle(X, Y, Radius);
            if ((Max_Bean mod 6) <> 0) then
            { If not the sixth bean in the row, increment X }
            { coordinate. }
               X := X + 40
            else
            { If sixth bean in row, set X,Y coordinates for new }
            { row. }
               begin
                  if (Y < 176) then
                  { This will not allow the setting of X,Y }
                  { coordinates to begin in the seventh row. }
                  { This does allow the position of bean #36 }
                  { to be the initial removal position. }
                     begin
                        X := 138;
                        Y := Y + 32;
                     end;
               end;
                     Max_Bean := Max_Bean + 1;
         end;
   for Index := 1 to 4800 do; { Provide for a short delay. }
      Max_Bean := 36
end;
```

The value of the variable `Max_Bean` reaches 37 during the execution of the drawing loop, so the value must be reset to 36 if the remainder of the program is to function properly.

Both Text and Drawing windows are shown simultaneously on the screen. Even though the Text window will become the active window, the Drawing window will still show the changes as beans are removed. Notice the use of delays, which were not necessary in the text-only version. Delays give the illusion of pacing and also allow the player to read the instructions in the Text window before action takes place in the Drawing window. You might need to adjust these delays to get a pleasing response from your Macintosh, because different Macs execute at different speeds.

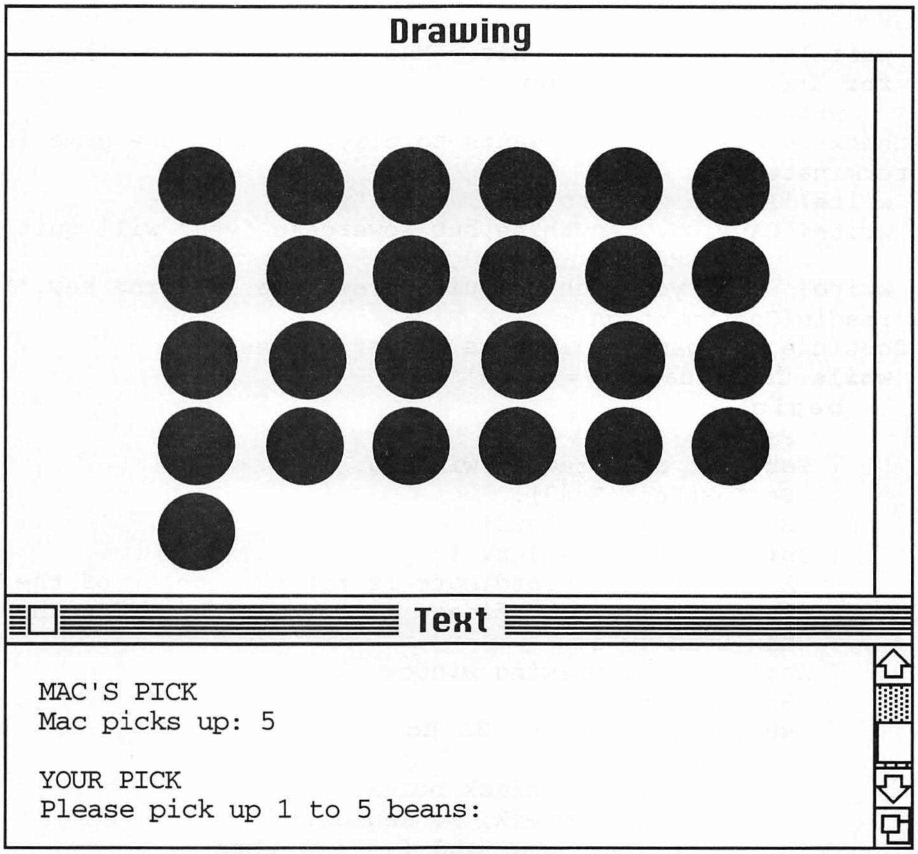
The 36 beans are drawn beginning in the top left portion of the Drawing window and proceeding from left to right. Six rows of 6 beans are drawn, for a total of 36. Because the position of the 36th bean is also the beginning position to be used when the `InvertCircle` command is executed to erase or "pick up" the beans, removal of the beans is in reverse order and is achieved through the following program lines:

```
{ Remove MAC_Pick of beans from display. }
repeat
   begin
   { Uses position of bean #36 as the beginning removal point. }
      InvertCircle(X, Y, Radius);
      for Index := 1 to 600 do; { Provide for a short delay. }
         X := X - 40;
      Max_Bean := Max_Bean - 1;

      if Max_Bean mod 6 = 0 then
         begin
            X := 338;
            Y := Y - 32;
         end;
      MAC_Pick := MAC_Pick - 1;
   end;
until MAC_Pick = 0;
```

As Figure 6.16 shows, execution of these program lines results in the removal of the number of beans that Mac picks. Similar steps are then executed to represent the player's pick. The ShowText command should be used to reactivate the Text window. The player will need to input his or her choice of the number of beans to pick, and this will be done in the Text window. The program lines used by Mac to pick up or remove beans can then be duplicated to have the program remove the number of beans selected by the player. However, the line `MAC_Pick := MAC_Pick - 1` must be replaced by `Player_Pick := Player_Pick - 1`, and the line **until** MAC_Pick = 0 must be replaced by **until** Player_Pick = 0.

Here is the complete listing of the modified Beanpicker program:

```
program Modified_Beanpicker(input, output);
{ Purpose:  This program is a game of skill in which the player }
{           competes against the computer. The computer will win }
{           most of the time. It also demonstrates several }
{           different types of loops and loop-control }
{           techniques. }
   const
      Radius = 14;
   var
      Number_of_Beans, Player_Pick, MAC_Pick : integer;
      X, Y, Index, Max_Bean : integer;
      Continuation : string[3];
      Box1, Box2, Box3 : rect;
begin
{ Hide all windows. }
   HideAll;
{ Establish boundaries for several different sizes of the }
{ Drawing window. }
   SetRect(Box1, 1, 40, 512, 345);
{ Listing continues after Figure 6.16. }
```



```
MAC'S PICK
Mac picks up: 5

YOUR PICK
Please pick up 1 to 5 beans:
```

Figure 6.16 Modified_Beanpicker game screens showing removal of beans picked by Mac and request for player to pick. The Text window is active at this moment.

```
{ Continuation of Modified_Beanpicker. }
   SetRect(Box2, 1, 40, 512, 260);
   SetRect(Box3, 1, 260, 512, 335);
{ Show the Text window to the user. }
   SetTextRect(Box1);
   ShowText;
{ Introduction and game header }
   for Index := 1 to 5 do
      writeln;
   writeln('                              ***** BEANPICKER *****');
   for Index := 1 to 5 do
      writeln;
   write('There are 36 black beans. You may pick up one (1) to ');
   write(' five (5) beans; the object of the game is to be the
         last');
   write(' one to pick up 1 to 5 beans, leaving no beans for
            your ');
   writeln('opponent, the Mac.  Mac picks up first. ');
   for Index := 1 to 4 do
      writeln;
{ Check to see if player wants to play; if not, the game is }
{ terminated. }
   write('If you wish to play, type "yes".   ');
   write('CAUTION...Anything but lowercase "yes" will quit
            BEANPICKER! ');
   write(' **If you wish to quit, press the <Return> key.** ');
   readln(Continuation);
{ Continue the game as long as player wishes. }
   while Continuation = 'yes' do
      begin
         Page;
      { Set Text and Drawing windows. }
         SetTextRect(Box3);
         SetDrawingRect(Box2);
      { Initialize variables. }
         X := 138; { X coordinate is for the center of the bean. }
         Y := 16;  { Y coordinate is for the center of the bean. }
         Max_Bean := 1;
      { Activate the Drawing window. }
         ShowDrawing;
         while Max_Bean <= 36 do
            begin
            { Display 36 black beans. }
               PaintCircle(X, Y, Radius);
               if (Max_Bean mod 6) <> 0 then
               { If not the sixth bean in the row, increment X }
               { coordinate. }
                  X := X + 40
               else
               { If sixth bean in row, set X,Y coordinates for }
               { new row. }
                  begin
                     if Y < 176  then
                  { This will not allow setting of X,Y }
                  { coordinates to begin in the seventh row.  }
                  { It does allow bean #36 to be positioned for }
                  { removal. }
                        begin
                           X := 138;
                           Y := Y + 32;
                        end;
                  end;
               Max_Bean := Max_Bean + 1;
            end;
         for Index := 1 to 4800 do ;
         { Provide for a short delay. }
            Max_Bean := Max_Bean - 1;
      { Set Max_Bean value to 36, Max_Bean value was 37. }
      for Index := 1 to 4 do
         writeln;
      Number_of_Beans := 36;
   { Begin the game and repeat until completed. }
      repeat
      { Computer selection of number to pick.  The result is }
      { displayed. }
         writeln('MAC`S PICK');
         for Index := 1 to 2400 do ;
         { Provide for a short delay. }
            if Number_of_Beans = 36 then
               MAC_Pick := abs(random) mod 6
            else
               MAC_Pick := Number_of_Beans mod 6;
            { Minimum pick of one. }
               if MAC_Pick = 0 then
                  MAC_Pick := 1;
      { Display number picked and reduce number }
      { of beans by number picked. }
         writeln('Mac picks up: ', MAC_Pick : 1);
         Number_of_Beans := Number_of_Beans - MAC_Pick;
      { Remove MAC_Pick of beans from display. }
         repeat
         { Position of bean #36 is used as the beginning of a }
         { removal point.}
            InvertCircle(X, Y, Radius);
            for Index := 1 to 600 do ;
            { Provide for a short delay. }
               X := X - 40;
               Max_Bean := Max_Bean - 1;
            if Max_Bean mod 6 = 0 then
               begin
                  X := 338;
                  Y := Y - 32;
               end;
            MAC_Pick := MAC_Pick - 1;
         until MAC_Pick = 0;
         if Number_of_Beans = 0 then
            begin
               writeln('>>>>> MAC WINS! <<<<< ');
               writeln;
            end
         else
            begin
               for Index := 1 to 1200 do ;
               { Provide for a short delay. }
                  ShowText;
               { Player inputs number picked. The results }
               { are displayed. Input request with error trap.}
                  if Number_of_Beans > 0 then
```

```
                          repeat
                             writeln('YOUR PICK');
                             write('Please pick up 1 to 5 beans. ');
                             readln(Player_Pick);
                          until (Player_Pick > 0) and (Player_Pick
                                 < 6);
     { Reduce number of beans by number picked and display }
     { new number. }
       Number_of_Beans := Number_of_Beans - Player_Pick;
     { Remove Player_Pick of beans from display. }
       repeat
          InvertCircle(X, Y, Radius);
          for Index := 1 to 600 do ;
          { Provide for a short delay. }
             X := X - 40;
          Max_Bean := Max_Bean - 1;
          if Max_Bean mod 6 = 0 then
             begin
                X := 338;
                Y := Y - 32;
             end;
          Player_Pick := Player_Pick - 1;
       until Player_Pick = 0;
       if Number_of_Beans = 0 then
          writeln('>>>>> YOU WIN! <<<<< ');
       for Index := 1 to 2400 do ;
       { Provide for a short delay. }
          writeln;
       end;
     { Game ends when the number of beans is zero. }
     until Number_of_Beans = 0;
     for Index := 1 to 2 do
        writeln;
     { Check to see if player wants to play again or quit; }
     { the game returns to the beginning if player wants to }
     { continue.  If not, it passes to the end. }
     write('Do you wish to play again?  Type "yes".  ');
     write('CAUTION...Anything but lower case "yes" will quit' );
     write(' BEANPICKER! ');
     write('     If you wish to quit, press the <Return> key. ');
     readln(Continuation);
  end;
{ Sign-off message. }
  writeln;
  writeln('Thank you for playing BEANPICKER.  Have a nice
          day!');
end.
```

Another consideration in modifying this program is the removal of lines that have become unnecessary: lines in the text-only version of `Beanpicker` that informed the player of the number of beans remaining. The countdown to zero beans is clearly visible as the beans disappear from the Drawing window.

One more point before we leave `Modified_Beanpicker` is the modification required for the THINK Pascal version. This is the only program in this chapter that requires modification beyond removal of the **uses** clause identifying the `QuickDraw1` library and the insertion of a `ShowDrawing` statement at the beginning of the program. Even so, the program requires only a very minor additional modification: the insertion of the `ShowDrawing` command at the beginning of the third **repeat-until** loop. This is shown in the few program lines added below.

```
{ Reduce number of beans by number picked and display }
{ new number. }
Number_of_Beans := Number_of_Beans - Player_Pick;
{ Remove Player_Pick of beans from display. }
repeat
   ShowDrawing;
   InvertCircle(X, Y, Radius);
   for Index := 1 to 600 do
      ;
   { Provide for a short delay. }
   X := X - 40;
   Max_Bean := Max_Bean - 1;
```

Without this addition, the program will execute, but the changes in the Drawing window will go unobserved, because only the original 36 beans will be drawn.

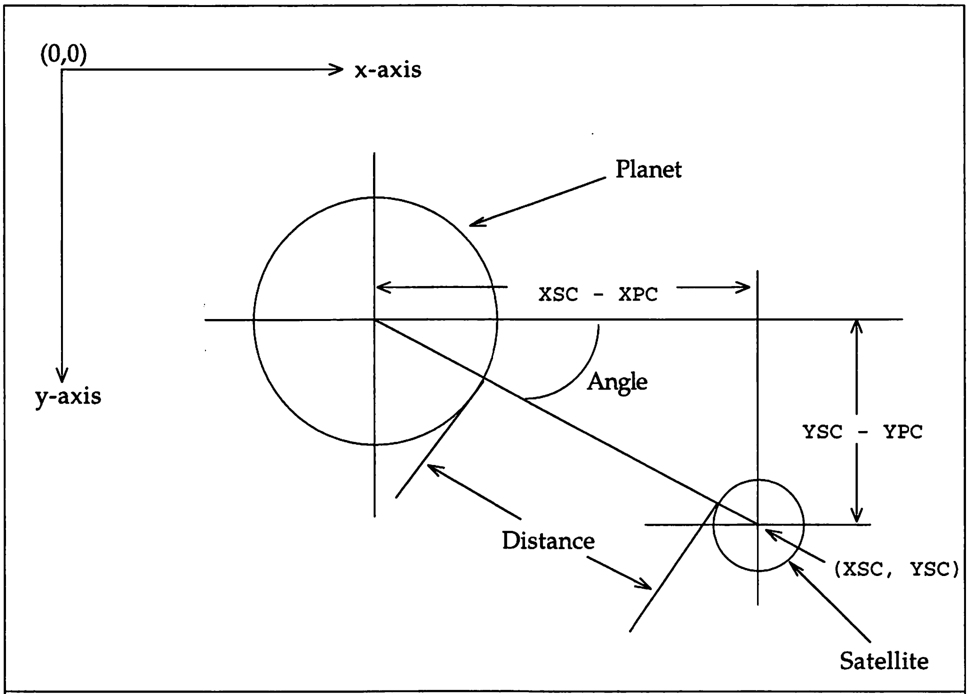
As a last example of graphics in a Macintosh Pascal program, consider writing a program to display the orbiting of a small satellite about a planet. The user of the program is required to enter the diameter of the planet, the diameter of the satellite, and the distance of the satellite from the edge of the planet. Figure 6.17 shows a view of the satellite and planet with respect to the radius of each object. It is obvious that the distance from the center of the planet to the center of the satellite can be specified by `Distance + (Diameter_Planet + Diameter_Satellite)/2`. The program must require that the distance between satellite and planet and the diameters of both satellite and planet be greater than zero. In addition, the diameter of the satellite cannot exceed the diameter of the planet. If it does, the program must clear the screen and request that new data be entered.

In computing the area for drawing the satellite, it is necessary to compute the center of the satellite with respect to the center of the planet. It would seem convenient to consider the center of the planet as the point (0, 0), but the Drawing window of the Macintosh forces us to place the center of the planet at the point (256, 156). In Figure 6.17 we have labeled this point `(XPC, YPC)`. In terms of the point `(XPC, YPC)`, the center of the satellite is given by the relationship

```
Total_Distance = Distance + (Diameter_Planet +
                              Diameter_Satellite) / 2.0

YSC - YPC = Total_Distance * sin( Angle )
XSC - XPC = Total_Distance * cos( Angle )
```

or

```
YSC = YPC + Total_Distance * sin( Angle )
XSC = XPC + Total_Distance * cos( Angle )
```



Distance between (XPC, YPC) and (XSC, YSC):

$$\text{Total\_Distance} = \text{Distance} + \frac{(\text{Diameter\_Planet} + \text{Diameter\_Satellite})}{2.0}$$

Relationship between the angle of rotation and the distance between the center of the planet and the satellite:

```
XSC - XPC = Total_Distance * cos(Angle)

YSC - YPC = Total_Distance * sin(Angle)
```

Figure 6.17 Coordinates for calculating the center of a satellite orbiting around a planet.

The action of this program is to begin with an initial angle of zero radians, display the satellite for a short period of time, then erase the satellite and redraw it after incrementing the value of `Angle`. Here are the steps of an algorithm describing the actions for entering data and drawing the rotating satellite:

```
Algorithm Orbiting_Satellite;
{ The purpose of this algorithm is to show the steps for
   displaying the rotation of a satellite about a planet. Two
   constants exist: XPC and YPC, given by 256 and 156,
   respectively. }
begin
{ Hide all windows from view. }
{ Establish a rectangle as the viewing area for both Text and }
{ Drawing windows. }
{ Set the size of the Text window. }
{ Show the Text window for input of data. }
   repeat
      Page;
      write( 'Enter the diameter of the planet: ');
      readln( Diameter_Planet );
      write( 'Enter the diameter of the satellite: ');
      readln( Diameter_Satellite );
      write( 'Enter the distance between the satellite and the
               planet: ');
      readln( Distance );
      Properties <-- ( Diameter_Planet > 0 ) and
               ( Diameter_Satellite > 0 ) and ( Distance > 0 ) and
                ( Diameter_Satellite < Diameter_Planet );
   until Properties;
{ Compute the distance between the center of the planet and that
   of the satellite. }
   Total_Distance <-- Distance + ( Diameter_Planet
                                    +  Diameter_Satellite ) / 2.0;
{ Set the size of the Drawing window. }
{ Show the Drawing window for viewing the rotation of the
   satellite. }
{ Write a title in the Drawing window. }
   MoveTo( 50, 50 );
   WriteDraw(' ORBITING SATELLITE ');
{ Draw the planet in the center of the screen. }
   Radius <-- Diameter_Planet / 2.0;
   SetRect( Area,  trunc( 256 - Radius ), trunc( 156 - Radius ),
              trunc( 256 + Radius ), trunc( 156 + Radius ) );
   FillOval( Area, dkgray );
{ Display the satellite rotating about the planet. }
   Radius <-- Diameter_Satellite / 2.0;
   while true do
      begin
         Angle <-- 0.0;
         repeat
         { Compute the center point of the satellite. }
            YSC <-- YPC + Total_Distance * sin( Angle );
            XSC <-- XPC + Total_Distance * cos( Angle );
         { Compute the boundary points for a rectangle where the }
         { satellite is to be drawn. }
```

```
            SetRect( Area, trunc( XSC - Radius ), trunc( YSC -
                  Radius ), trunc( XSC + Radius ), trunc( YSC +
                  Radius ) );
            FillOval( Area, black );
         { Delay execution for a short period of time. }
         { Erase the view of the satellite. }
           EraseOval( Area );
           Angle <-- Angle + 0.025;
         until Angle > 2 * pi;
      end;
end.
```

Here is our algorithm as a Macintosh Pascal program, `Orbiting_Satellite`:

```
program Orbiting_Satellite(input, output);
{ Purpose:  To show the steps required to display }
{           the rotation of a satellite about a }
{           planet. }
   uses
      QuickDraw1;
   const
      XPC = 256;
      YPC = 156;
   var
      Area : Rect;
      Time : integer;
      XSC, YSC, Radius, Angle : real;
      Diameter_Satellite, Diameter_Planet, Distance : real;
      Total_Distance : real;
      Properties : Boolean;
begin
{ Hide all windows from view. }
   HideAll;
{ Set the size of the Text and Drawing windows. }
   SetRect(Area, 0, 20, 512, 312);
{ Set the area for viewing the Text window. }
   SetTextRect(Area);
{ Show the Text window for viewing. }
   ShowText;
   repeat
      Page;
      write('Enter the diameter of the planet: ');
      readln(Diameter_Planet);
      write('Enter the diameter of the satellite: ');
      readln(Diameter_Satellite);
      write('Enter the distance between the satellite and the
              planet: ');
      readln(Distance);
      Properties := (Diameter_Planet > 0) and (Diameter_Satellite
                    > 0) and (Distance > 0) and
                    (Diameter_Satellite < Diameter_Planet);
   until Properties;
{ Compute the distance between the center of the planet and }
{ that of the satellite. }

   Total_Distance := Distance +
                ( Diameter_Planet + Diameter_Satellite ) / 2.0;
{ Show the Drawing window for viewing the satellite. }
   HideAll;
{ Set the area for viewing the Drawing window. }
   SetDrawingRect(Area);
{ Show the Text window for viewing. }
   ShowDrawing;
{ Write a title in the Drawing window. }
   MoveTo(50, 50);
   WriteDraw(' Orbiting  Satellite ');
{ Draw the planet in the center of the screen. }
   Radius := Diameter_Planet / 2.0;
   SetRect(Area, trunc(256 - Radius), trunc(156 - Radius),
                 trunc(256 + Radius), trunc(156 + Radius));
   FillOval(Area, dkgray);
{ Display the satellite rotating about the planet. }
   Radius := Diameter_Satellite / 2.0;
   while true do
      begin
         Angle := 0.0;
         repeat
         { Compute the center point of the satellite. }
            YSC := YPC + Total_Distance * sin(Angle);
            XSC := XPC + Total_Distance * cos(Angle);
         { Compute the boundary points for a rectangle where }
         { the satellite is to be drawn. }
            SetRect(Area, trunc(XSC - Radius), trunc(YSC -
               Radius), trunc(XSC + Radius), trunc(YSC + Radius));
            FillOval(Area, black);
         { Delay execution for a short period of time. }
            for Time := 1 to 100 do; { nothing }
         { Erase the view of the satellite. }
            EraseOval(Area);
            Angle := Angle + 0.025;
         until Angle > 2 * pi;
      end;
end.
```

If someone tells you that the satellite is orbiting in the wrong direction, you can replace the statement

```
Angle := Angle + 0.025;
```

with

```
Angle := Angle - 0.025;
```

to change the direction of the orbit. Only the standard minor modifications are needed to change the program to THINK Pascal.

6.7 USING COLOR GRAPHICS

Macintosh and THINK Pascal programs can draw on color output devices by using QuickDraw procedures that are capable of setting both the foreground and background colors. Eight standard colors (sometimes referred to as the *old-style* grafPort colors) as constants are predefined in Macintosh Pascal and include *blackColor*, *whiteColor*, *redColor*, *greenColor*, *blueColor*, *cyanColor*, *magentaColor*, and *yellowColor*. Included in library QuickDraw2 are two routines `ForeColor` and `BackColor` that enable applications to draw on color output devices such as the screen of a color monitor. A brief discussion of these two procedures follows.

```
procedure ForeColor ( Color : longint );
```

This routine sets the foreground color for all drawings in the Drawing window, using the current value of `Color`. The argument `Color` can be one of eight standard predefined colors.

```
procedure BackColor ( Color : longint );
```

This routine sets the background color for all drawings in the Drawing window, using the current value for `Color`. The argument `Color` can be one of eight standard predefined colors.

The routine `ForeColor` allows you to draw with a color that shades the area being drawn. In short, it fills the foreground area with the value of a color given by the argument `Color`. The routine `BackColor` allows you to erase an area with a background color specified by the value of the argument `Color`. By having both routines, the background color can be set and remain unchanged, while the foreground color can be changed by executing the procedure `Foreground` with `Color` being assigned a new value each time.

The following Macintosh Pascal program demonstrates both of these two procedures. Their effect can only be seen with a color monitor, because black-and-white monitors always produce a black pattern when any color other than white is drawn. This program draws six differently colored rectangles by resetting the foreground color with the procedure `ForeColor`. This routine is also used to display two different prompts to the Drawing window, using the procedure `WriteDraw`. The procedure `BackColor` is used to set a white background color for erasing the colored rectangles as well as an area where text is displayed:

```
program Foreground_And_Background_Colors (input, output);
{ Purpose:  This program demonstrates the procedures ForeColor }
{           and BackColor by drawing several colored rectangles }
{           in the foreground and erasing the rectangles with }
{           a white background color. }
  uses
    QuickDraw1, QuickDraw2;
  var
```

```
   Box : Rect;
   Top, Left, Bottom, Right, X_Delta, Y_Delta, Mark_Time: integer;
   Angle : real;
begin
{ Establish a boundary for the Drawing window. }
   Box.top := 40;
   Box.left := 40;
   Box.bottom := 300;
   Box.right := 500;
   SetDrawingRect(Box);
{ Show the Drawing window and a rectangular frame within }
{ the Drawing window. }
   ShowDrawing;
   FrameRect(Box.top, Box.left, Box.bottom - 91, Box.right - 91);
{ Paint nested colored rectangles in the Drawing window. The }
{ for-loops are provided for their effect when viewing the }
{ screen. }
   ForeColor(cyanColor);
   PaintRect(41, 41, 208, 408);
   for Mark_Time := 1 to 4000 do
   ;
   ForeColor(magentaColor);
   PaintRect(56, 76, 194, 374);
   for Mark_Time := 1 to 4000 do
   ;
   ForeColor(greenColor);
   PaintRect(71, 111, 179, 339);
   for Mark_Time := 1 to 4000 do
   ;
   ForeColor(blueColor);
   PaintRect(86, 146, 164, 304);
   for Mark_Time := 1 to 4000 do
   ;
   ForeColor(redColor);
   PaintRect(101, 181, 149, 269);
   for Mark_Time := 1 to 4000 do
   ;
   ForeColor(yellowColor);
   PaintRect(111, 206, 138, 243);
   for Mark_Time := 1 to 4000 do
   ;
{ Draw in red letters a prompt to continue execution. }
   ForeColor(redColor);
   MoveTo(100, 230);
   PenSize(5, 5);
   WriteDraw(' Press the mouse button to continue : ');
   while (not Button) do
   ;
{ Now erase the area containing the prompt with a rectangle }
{ having white as its background color. }
   BackColor(whiteColor);
   EraseRect(210, 100, 230, 330);
{ Now erase the areas containing colored rectangles by drawing }
{ 184 rectangles with a white background color. The for loop is }
{ used for its effect on drawing rectangles that erase the }
{ colored areas of the screen.}
   Angle := (208 - 41) / (408 - 41);
   for X_Delta := 1 to 183 do
      begin
         Y_Delta := trunc(X_Delta * Angle);
         Left := 224 - X_Delta;
         Right := 224 + X_Delta;
         Top := 124 - Y_Delta;
         Bottom := 124 + Y_Delta;
         EraseRect(Top, Left, Bottom, Right);
      end;
   EraseRect(41, 41, 208, 408);
{ Draw in blue letters a prompt to quit execution. }
   ForeColor(blueColor);
   MoveTo(120, 230);
   PenSize(5, 5);
   WriteDraw(' Press the mouse button to quit : ');
   while (not Button) do
   ;
   HideAll;
end.
```

Notice that the routines `LineTo`, `DrawLine`, `WriteDraw`, `PaintCircle`, `FrameRect`, `PaintRect`, `FrameOval`, and `PaintOval` draw shaded areas using the present foreground color, while the routines `FillRect`, `EraseRect`, `FillOval`, and `EraseOval` draw shaded areas using the present background color. While the procedures `FillRect` and `FillOval` require a pattern of either *white*, *black*, *gray*, *ltgray*, or *dkgray*, the areas being drawn will use color for the pattern that is presently assigned to the background color. Both the routines `InvertCircle` and `InvertRect` will draw a black shaded area if the foreground is white and a white shaded area if the foreground is black. If the foreground is a color other than black or white, it will invert the foreground color with some other color. Based on a test program executed by the authors, the inverted colors are as follows:

Standard Color	Inverted Color
black	white
red	light green
blue	pink
yellow	dark gray
green	magenta
white	black
magenta	black

SUMMARY

In this chapter we discussed some of the easier procedures and functions that are available through the `QuickDraw1` library. Keep in mind that when using the Drawing window, several procedures are available for drawing and painting to this grafPort. For quick reference, these are summarized alphabetically.

Procedure	Parameter(s)	Brief description
`Button`	None	Returns `Boolean` value *true* if mouse button is pressed, *false* if button is released.
`DrawLine(`x_1, y_1, x_2, y_2`)`	x_1, y_1, x_2, y_2 (all `integer`)	Draws line from point (x_1, y_1) to point (x_2, y_2).
`EraseOval(Top,Left, Bottom,Right)`[a]		Erases an oval that has been painted within a specified rectangle.
`EraseRect(Top,Left, Bottom,Right)`[a]		Erases a rectangle that has been painted in the Drawing window.
`FillOval(Top,Left, Bottom,Right,Pat)`[a,b]		Fills an oval within a specified rectangle with a pattern.
`FillRect(Top,Left, Bottom,Right,Pat)`[a,b]		Fills a rectangle with a pattern.
`FrameOval(Top,Left, Bottom,Right)`[a]		Draws a specified rectangular frame about an oval.
`FrameRect(Top,Left, Bottom,Right)`[a]		Draws an outline of a rectangle, given the coordinates for the upper left and lower right corners.
`GetDrawingRect(Box)`	Box is of type `Rect`.	Assigns the current Drawing window coordinates to Box.
`GetMouse(`x, y`)`	x, y (both `integer`)	Assigns the current mouse position to the point (x, y) (Macintosh Pascal).
`GetMouse(`P`)`	P (point)	Assigns the current mouse position to the point (x, y) (THINK Pascal).

Procedure	Parameter(s)	Brief description
`GetTextRect(Box)`	Box is of type `Rect`.	Assigns the current Text window coordinates to Box.
`HideAll`	None	Hides all windows.
`InvertCircle(`*x*`,`*y*`,`*r*`)`	*x*, *y*, *r* (all `integer`)	Paints a circle of radius *r* opposite in color to the background color within a circle; can be used to erase a circle.
`InvertOval(Top,Left, Bottom,Right)`[a]		Paints the interior of an oval within a specified rectangle a color opposite to the background color of the oval.
`InvertRect(Top,Left, Bottom,Right)`[a]		Paints a rectangle specified by the left and right corners a color opposite to the rectangle's present background color.
`LineTo(`*x*`,`*y*`)`	*x*, *y* both `integer`	Draws a direct line from the current pen position to the point (*x*, *y*).
`MoveTo(x, y)`	x, y both `integer`	Moves the pen to a point in the Drawing window but does not draw.
`Page`	None	Clears the Text window of any characters.
`PaintCircle(`*x*`,`*y*`,`*r*`)`	*x*, *y*, *r* (all `integer`)	Paints a circle of radius *r* with a center located at point (*x*, *y*).
`PaintOval(Top,Left, Bottom,Right)`[a]		Paints an oval within a rectangle specified by the left and right corners.
`PaintRect(Top, Left, Bottom, Right)`[a]		Paints the interior of a specified rectangle a color opposite to the background color of the rectangle.
`PenSize(width, height)`	Width, height both integer	Allows the line width in the Drawing window to be changed.
`SetDrawingRect( Box)`	Box is of type `Rect`	Sets the size and location of the Drawing window but does not show the Drawing window.

Procedure	Parameter(s)	Brief description
`SetRect(Box, Left, Top, Right, Bottom)`[a]	Box is of type `Rect`.	Assigns the boundary points to the rectangle `Box`.
`SetTextRect( Box )`	Box is of type `Rect`.	Sets the size and location of the Text window but does not show the Text window.
`ShowDrawing`	None	Reveals the Drawing window after it has been set.
`ShowText`	None	Reveals the Text window after it has been set.
`WriteDraw( exp1,...,expn)`		Displays text in the Drawing window.
`ForeColor(Color)`	`Color` is of type `longint`.	Sets foreground color to the current value of `Color`.
`BackColor(Color)`	`Color` is of type `longint`.	Sets background color to the current value of `Color`.

[a] `Top, Left, Bottom, Right` are all of type `integer`.
[b] `Pat` is of type `Pattern`.

REVIEW QUESTIONS

1. Explain the meaning of the term *procedure*.
2. What two special libraries in Macintosh Pascal are devoted to graphics?
3. What purpose does the **uses** clause serve in Macintosh Pascal?
4. When writing a Macintosh Pascal program requiring procedures from the `QuickDraw1` library, is it necessary to use the following declaration? How does this differ in THINK Pascal?

```
uses
   QuickDraw1;
```

5. What is meant by the term *pixel*?
6. How many pixels exist in the horizontal and vertical directions on the screen of the Macintosh computer?
7. What is the density of pixels per inch on the Macintosh screen?
8. What is meant by the term *grafPort*?
9. What names are given to the three grafPorts when the Macintosh Pascal desktop is initiated?
10. What is the initial size of the Drawing window in terms of pixels?
11. If a pixel in the Drawing window is represented by an (x, y) pair (x representing the horizontal axis, and y the vertical axis), what are the x and y values of the following points?

(a) (45, 89) (b) (212, 300)
(c) (50, 78) (d) (500, 312)

12. What command is used to reposition the drawing pen?
13. What command allows you to draw a straight line?
14. Write the commands to draw a direct line from point (50, 50) to point (300, 412).
15. How does the command `DrawLine` differ from the command `LineTo`?
16. Can either command, `LineTo` or `DrawLine`, be used to draw a dot? Test your answer, using the Instant window.
17. What is the purpose of the procedure `PenSize`?
18. If *N* represents the line width for the drawing pen, show how the procedure `PenSize` would be called if the vertical size were twice the horizontal width.
19. If *N* represents the line width for the drawing pen, show how the procedure `PenSize` would be called when the horizontal size is three times the vertical width.
20. How can the normal line width for the drawing pen be reestablished?
21. How can the drawing pen be hidden?
22. What is the purpose of the command `WriteDraw`?
23. Can executing the command `PenSize` affect the drawing of characters using the command `WriteDraw`?
24. How can the **Font Control** option of the **Windows** menu be used to change the type and size of font used in drawing characters in the Drawing window?
25. What is the difference between the procedures `PaintCircle` and `InvertCircle`?
26. What is the purpose of the procedure `FrameRect`?
27. What does the special data type `Rect` represent?
28. How do the commands `PaintRect` and `InvertRect` differ from `PaintCircle` and `InvertCircle`?
29. How can a rectangle be painted with a background pattern other than black?
30. What does the data type `Pattern` represent? What possible values can a data object of type `Pattern` be assigned?
31. Using Macintosh or THINK Pascal, how can motion be simulated?
32. How do the procedures for painting an oval offer alternatives for drawing circles in the Drawing window?
33. How can an oval-drawing procedure be used to erase a circle from the Drawing window?
34. Define and briefly explain three mouse-control commands.
35. What code is needed to see if a mouse button is still down?
36. Write the code that randomly selects a point inside the Drawing window and then checks to see if the user is able to locate this point while moving the mouse with the mouse button pressed down.
37. How can the size of the Text window be set when executing a Macintosh Pascal program?
38. How can the size of the Drawing window be set when executing a Macintosh Pascal program?

39. What commands are necessary for displaying both the Text window and the Drawing window?
40. How is the data type `Rect` represented at the level of Macintosh Pascal?
41. Write the necessary code for dividing the screen in half so that the right half will show the Text window and the left half the Drawing window.
42. What is the purpose of `GetTextRect` and `GetDrawingRect`?
43. What eight standard colors are supported by Macintosh and THINK Pascal?
44. If the eight standard colors are constants, how could you determine their numeric values from each of their given names?
45. What is the purpose of procedure `ForeColor`? What is the purpose of procedure `BackColor`?

PROGRAMMING EXERCISES

1. The following program draws a pyramid (see Figure 6.18), including the hidden lines *DE* , *EB* , and *EA* , and the corners designated as *A, B, C, D,* and *E*, respectively:

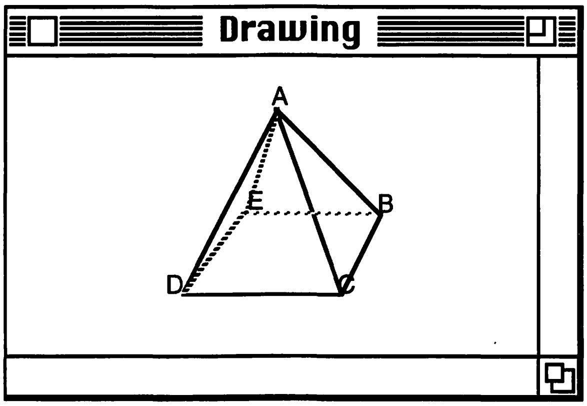


Figure 6.18

```
program Exercise_One (input, output);
{ Purpose:  This program displays a pyramid with hidden lines }
{           being shown. }
begin
   ShowDrawing;
{ Label five points on the screen. }
   MoveTo(100, 70);
   WriteDraw('A');
   MoveTo(140, 110);
   WriteDraw('B');
   MoveTo(125, 140);
   WriteDraw('C');
   MoveTo(60, 140);
   WriteDraw('D');
   MoveTo(90, 110);
   WriteDraw('E');
{Draw a line from point A to point B.}
   DrawLine(100, 70, 140, 110);
{Draw a line from point B to point C.}
   LineTo(125, 140);
{Draw a line from point C to point D.}
   LineTo(65, 140);
{Draw a line from point D to point E.}
   LineTo(90, 110);
{Draw a line from point E to point B.}
   LineTo(140, 110);
{Finish drawing the remaining lines.}
   DrawLine(100, 70, 125, 140);
   DrawLine(100, 70, 65, 140);
   DrawLine(100, 70, 90, 110);
end.
```

Modify this program so that the hidden lines *DE*, *EA*, and *EB* are represented by short dashes, as shown in Figure 6.18.

2. Modify the program in Exercise 1 so that the labels for corners `A`, `B`, `C`, `D`, and `E` are not touched by any lines connecting these points.

3. Using the procedures `PenSize`, `DrawLine`, and `LineTo`, write and execute a program to form the squares shown in Figure 6.19.

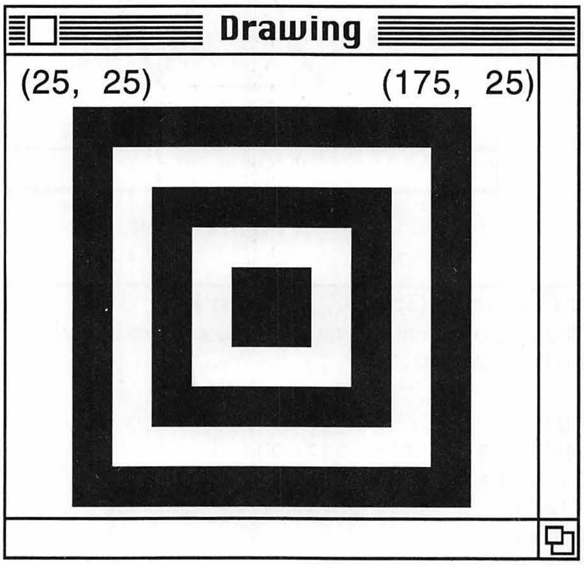


Figure 6.19

Set the pen width and height to 15 units. Note the coordinates for the upper left and upper right corners. *Hint*: To understand the effect of height and width when using the procedure `PenSize`, try drawing two vertical lines, one from (175, 0) to (175, 200) and another from (25, 0) to (25, 200), and two horizontal lines, one from (0, 25) to (200, 25) and the other from (0, 175) to (200, 175).

4. Rewrite the program in Exercise 3 so that only `PaintRect` and `InvertRect` are used to draw the same set of squares.

5. Write a program for drawing the target shown in Figure 6.20 using the procedures `PaintCircle`, `InvertCircle`, `MoveTo`, and `WriteDraw`. Radii of the circles are 100, 85, 70, 55, 40, and 25 (each dark line being 15 units in width).

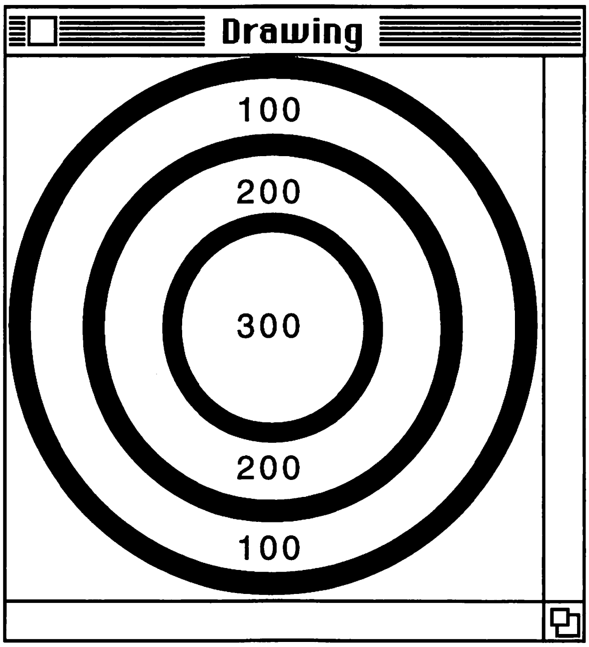


Figure 6.20

6. Modify the program titled `Random_Dots` to make it randomly draw square dots of length 10 units instead of circles. In addition, have the program set the Drawing window for the complete screen. *Hint* : Add the following variable to those presently declared:

```
Box : Rect;
```

and the following statements at the beginning of the body of the program:

```
HideAll;
SetRect( Box, 0, 18, 512, 342 );
SetDrawingRect( Box );
```

```
ShowDrawing;
```

7. The purpose of the next program is to apply the procedures `GetMouse` and `Button` to draw a direct line between two points, one to be displayed as point *A* and a second as point *B*. Complete this program so that when executed, moving the mouse to a point on the screen and clicking the button, then moving the mouse to a second point on the screen and again clicking the button causes two labels, `A` and `B`, to be displayed and a direct line to be drawn between the two points.

```
program Exercise_Seven(input, output);
{ Purpose:  A program for drawing direct lines using GetMouse }
{           for input. }

   const
      Number =  ? ;  { Try your own value to see if your delay }
                     { is working. }
   var
{ Declare variables for the x, y values of the first and second }
{ points. }
   J : integer;
   Down : Boolean;
   Screen : Rect;

begin
{ Hide all windows; then set and show the Drawing window. }
{ Continuous loop for establishing the first and second points. }
   while true do
      begin
         Down := false;
         while not Down do
            begin
            { Get the first point using GetMouse and assign Down }
            { a value using the function Button. }
            end;
         { Execute delay for the time needed to release button. }
            for J := 1 to number do {nothing};
         { Move to the first point and display the letter A on }
         { the screen. }
            Down := false;
         while not Down do
            begin
            { Get second point using GetMouse and assign Down }
            { a value using the function Button.}
            end;
         { Execute a delay for the time needed to release the }
         { button. }
         for J := 1 to number do {nothing};
         { Move to the second point and display the letter }
         { B on the screen.  Use DrawLine to draw a direct }
         { line from the first point to the second point. }
```

```
    end;
  end.
```

8. Write a program that executes the following steps:

   (a) Hide all windows.
   (b) Set and show the Text window in the center of the screen.
   (c) Prompt the user for the following information: last name, first name, middle name, street address, city, state, zip code.
   (d) Hide the Text window.
   (e) Set and show the Drawing window in the center of the screen.
   (f) Using `WriteDraw`, display the following four lines of data to the Drawing window:
      (i) The person's complete name in the order of first, middle, last
      (ii) Street address
      (iii) City followed by state
      (iv) Zip code

   This data should appear within a rectangle having a border that is 5 units wide.

9. Write a program that first hides all windows, then splits the screen with the left half of the screen for the Text window and the right half for the Drawing window. In the Text window, use the `write` and `writeln` commands to explain how to compute the areas of a circle and a rectangle. In the Drawing window, show shaded figures to demonstrate your brief explanation. Use labels with your figures to demonstrate the radius of a circle and the sides of a rectangle.

10. A supermarket wants to use the Macintosh computer to display unit price labels in the format shown in the Figure 6.21. This problem requires the following input from the keyboard:

Price: 0.98	Actual Size: 12.6 OZ
Item: Baked beans with bacon	
Unit Price: 1.24 per 16 OZ	

Figure 6.21

   (a) Price of the item represented in dollars and cents
   (b) Standard unit size (16 for ounces, 01 for pounds)
   (c) Actual size of the item
   (d) Description of the unit size (OZ for ounces, LB for pounds)
   (e) Description of the product, limited to 25 characters

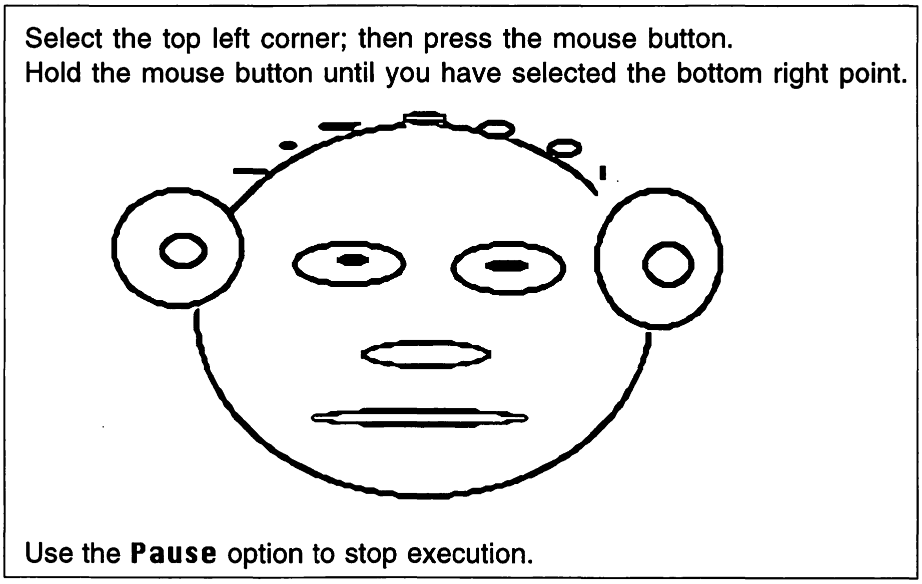
After entering this information in response to prompts appearing in the Text window, and before displaying the label to the Drawing window, compute the unit price, using the following rule:

```
Unit_price <-- Price_of_item/Actual_Size * Standard_Unit_Size
```

For example, the sample label appearing in the figure is from the following set of data:

```
Price:    0.98             Standard unit size:     16
Size:     12.6             Unit size description:  OZ
Description of item:       Baked beans with bacon
```

11. Using the general concept of the algorithm and the program titled `Directed_Lines`, rewrite these to draw only ovals to the Drawing window by picking an initial point with the mouse and moving the mouse toward the lower right while the button is down. Once the mouse button has been released, an oval is drawn from the initial point where the mouse button was pushed and the point where it was released (top left corner versus bottom right corner of a rectangle). You may experience a problem where many ovals are drawn as you move the cursor across the screen by changing the position of the mouse. *Hint:* Think about erasing an oval immediately after it has been drawn. Figure 6.22 shows an example of output for this program.

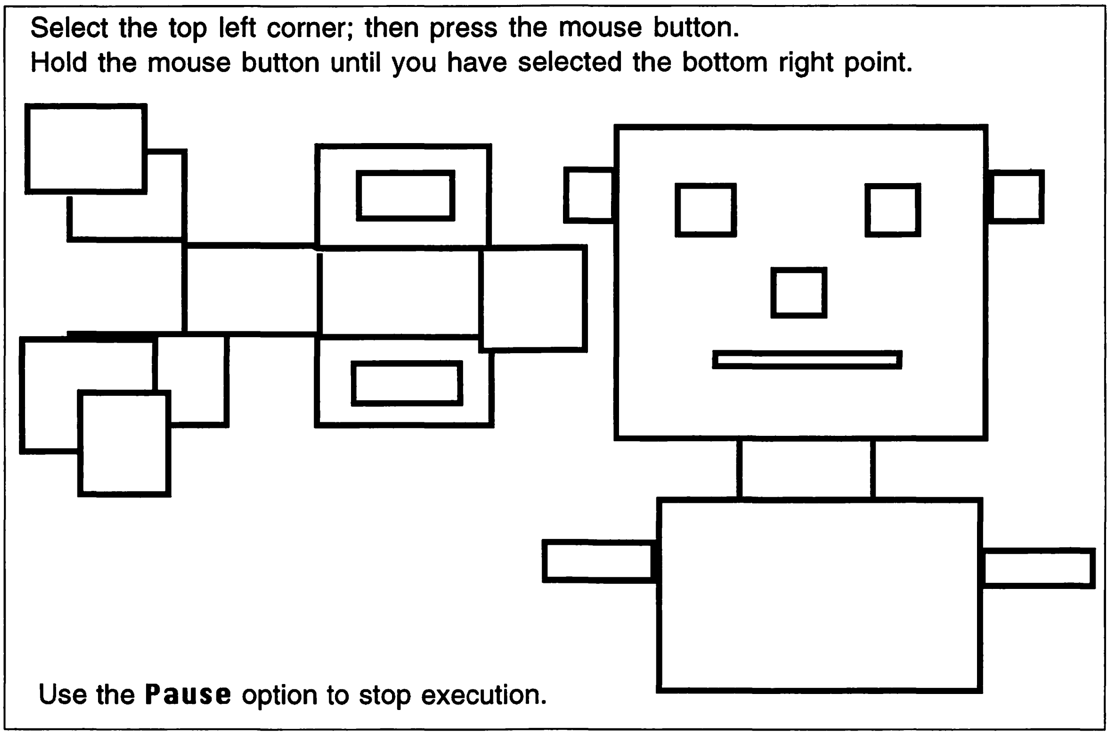

Select the top left corner; then press the mouse button.
Hold the mouse button until you have selected the bottom right point.

Use the **Pause** option to stop execution.

Figure 6.22

12. Using the concept from Exercise 11, develop an algorithm and write a program for drawing only rectangles to the Drawing window by picking an initial point with the mouse and moving the mouse toward the lower

right while the button is down. Be careful to avoid the problem described with ovals. Figure 6.23 shows an example of output for this program.



Select the top left corner; then press the mouse button.
Hold the mouse button until you have selected the bottom right point.

Use the **Pause** option to stop execution.

Figure 6.23

13. Develop an algorithm using the concepts learned from the algorithm `Directed_Lines` and the programs written for Exercises 11 and 12 to allow the user the option of drawing directed lines, ovals, or rectangles. Convert this algorithm into a Macintosh or THINK Pascal program, and show ᵗhat the program is functional.

14. Here is a program for displaying the character set of Macintosh Pascal to the Text window:

```
program Displaying_Characters(input, output);
   var
      Counter : integer;
begin
{ Purpose:  Display the character set of the Macintosh. }
   HideAll;
   ShowText;
{ Display characters 0 through 127. }
   for Counter := 0 to 127 do
      write( chr( Counter ));
```

```
    writeln;
end.
```

Modify the program so that the Macintosh screen is split into two separate windows: Text and Drawing. While executing, have each character displayed in the Text window using the write command and drawn in the Drawing window using the `WriteDraw` command. By using **Font Control** from the **Windows** option, change the font and font size, and again execute your program to see if the character set has changed.

15. Rewrite the algorithm `Bar_Chart` for drawing the bars of a bar chart vertically instead of horizontally. Transfer your algorithm from its present form into a Macintosh Pascal program to show that it is functional.

16. Develop an algorithm for simulating the functioning of traffic lights at an intersection. As Figure 6.24 shows, choose a black pattern to represent red, dark gray for yellow, and light gray for green. The lights should change periodically, with the sequence from green to yellow to red requiring several seconds of real time. Keep the traffic lights set for one to two minutes before changing.

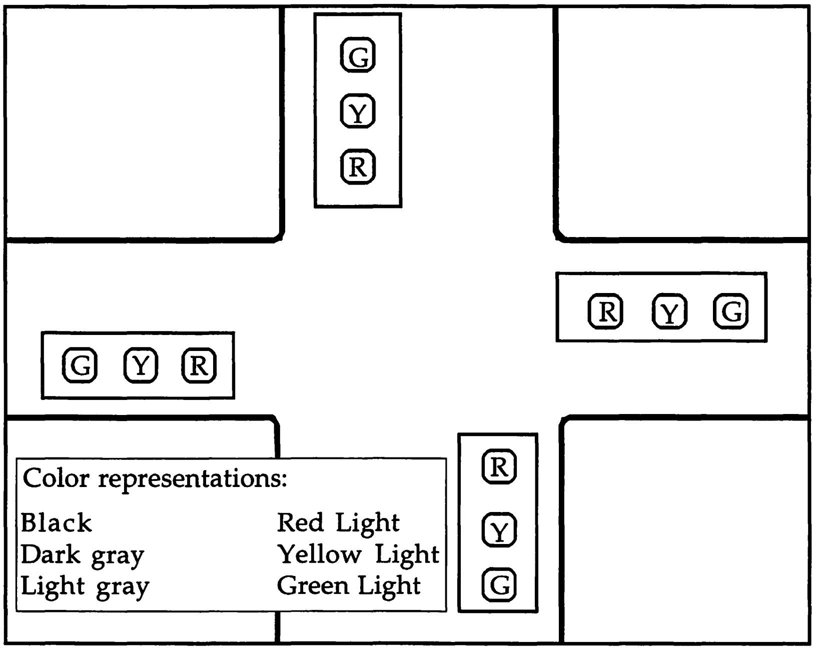


Figure 6.24

17. Modify the algorithm and program from Exercise 16 so that the lights are flashing red in one direction and yellow in the other. Place two

buttons on the screen so that if the mouse is moved to one of the buttons while the mouse button is down, the flashing red will change to green after the flashing yellow has changed to red. After one minute, have the green change to yellow and then to flashing red, while the opposing traffic light goes to flashing yellow (see Figure 6.25).

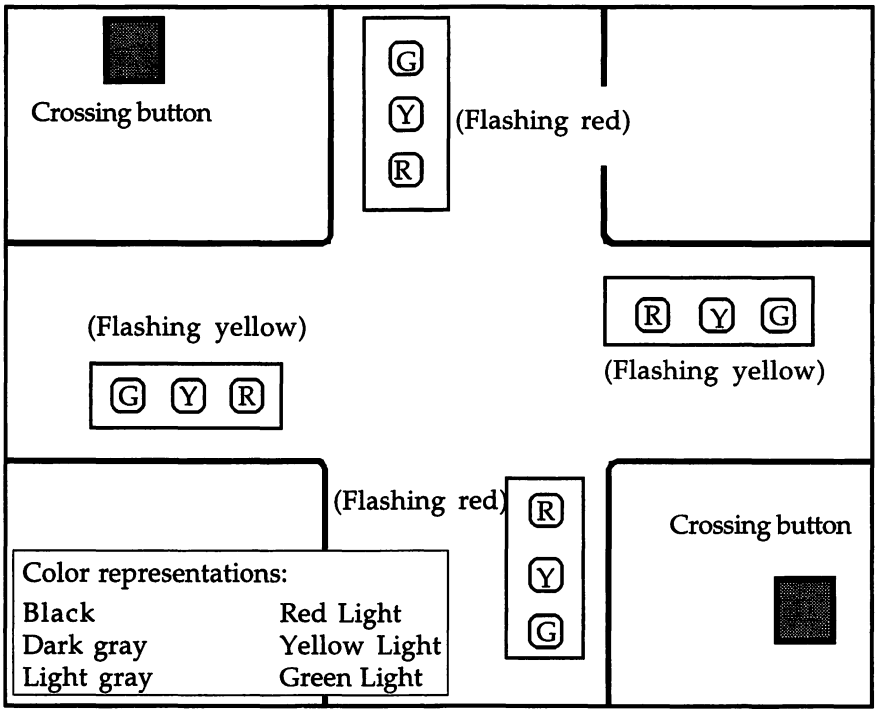


Figure 6.25

18. As a more difficult exercise, write an algorithm for drawing the characters of the Macintosh Pascal character set to the Drawing window as shown in Figure B.1 of Appendix B. Convert your algorithm into a program, and show ʰat it is functional by choosing several font types and sizes from the option **Font Control**.

19. If you have a color monitor, rewrite either Exercise 16 or 17 so that the traffic lights use the colors red, yellow, or green as foreground colors and white as a background color.

20. Write a program that will conveniently show the inverse colors for each of the eight standard colors.

Chapter 7

Procedures and Functions

OBJECTIVES

After completing Chapter 7, you will know the following:

1. What is meant by the terms *procedure* and *function.*
2. How to pass information between actual and formal parameters.
3. What is meant by *value* and *variable* parameters.
4. What is meant by *global* and *local* variables.
5. How to use forward declarations to ease the writing of procedures and functions.
6. How to define and write recursive procedures and functions in Pascal.
7. How to use a structure chart as a tool in the development of an application.
8. How to use procedure and function names as formal parameters.

7.1 THE CONCEPT OF A PASCAL PROCEDURE

In earlier chapters we have used predefined procedures and functions such as `write`, `writeln`, `read`, and `readln` as well as procedures and functions from the QuickDraw library to perform special tasks. Although we have not examined any of the Pascal code for these library routines, we know that a procedure or function is a collection of executable statements capable of performing a specific operation. A function differs from a procedure in that a function has a value returned through its name, whereas a procedure results in an effect. In this chapter we consider the steps necessary to write our own procedures and functions. Procedures and functions coded by programmers are frequently referred to as *user-defined* routines.

There are several reasons why it is important for a higher level language like Pascal to support the use of procedures and functions. First, time is saved by placing repetitive statements within the body of a procedure, or function. Thus, we need only call upon (invoke) the procedure, or function, each time the repetitive code is needed. Second, a program using procedures and functions is easier to document and read as an algorithm. Specifically, inclusion of a procedure or function in an algorithm defers the detailed reading of code, so that we can focus our attention on understanding the major steps in solving a problem. We can save time in examining a program, because we are not required to read repetitive executable code. Rather, we can direct our attention to reading those procedures, or functions, that require our immediate attention. Third, by using procedures and functions, we extend the definition of the language. For example, though the QuickDraw library supports only basic drawing routines, we can create our own procedures for drawing specific figures such as bar or pie charts. In addition, if the computer language has the ability to group these procedures and functions into separate file units, we can create our own libraries of programs for other applications to borrow. Fourth, by writing procedures and functions, we divide problems into smaller units, referred to as *subprograms* or *program units*, with each subprogram consisting of a manageable number of executable statements. Calling a subprogram can be equated to executing a major step of an algorithm, so each of the substeps for executing a major step in an algorithm can in turn be represented by one or more subprograms. This allows us to view a program as an algorithm for an intricate problem, with each step of the algorithm partitioned into smaller portions. In short, we can practice the principles of step-wise refinement and top-down design in the development of a software application.

7.1.1 Definition of a Pascal Procedure

As Figure 7.1 shows, all Pascal programs in Macintosh and THINK Pascal share the same format.

```
   program  Program_Name(input, output);

   { uses clause }

   { list of label declarations }
   { list of constant declarations }
   { list of user-defined types }
   { list of variable declarations }
   { list of procedure and function declarations }

begin
   { executable body of the main Pascal program }
end.
```

Figure 7.1 Pascal program format.

The **program** statement serves as the entry point into a Pascal program. The program parameter `input` directs the Pascal run-time system to accept information from the standard input file device, that is, the keyboard, and the program parameter `output` directs the Pascal run-time system to transfer information to the standard output file device, the screen. For Macintosh and THINK Pascal, the use of either identifier, `input` or `output`, is optional. These two parameters are used to conform to standard Pascal code. The **program** statement is followed by the **uses** clause: a block composed of a declaration part and a statement part. The declaration part can declare labels, constants, user-defined types, and variables, as well as procedures or functions local to the program. These procedures and functions are often referred to as *internal subprogram units*. These objects are then known throughout the body of the main program. The statement part, given by comments and executable statements between the **begin** and **end** brackets, represents the main executable body of the Pascal program. In this chapter we will refer to it as the *body of the main program*. The program begins execution by executing the statement **begin** and ends with execution of the last end statement **end.** (**end** period).

As Figure 7.2 shows, a Pascal procedure is similar to a Pascal program in both its form and conception. In Figure 7.2 the reserved word **procedure**, along with the name of the procedure and what is called the *formal parameter list*, represent the entry point into the procedure, where you would begin to read the procedure and where the procedure would begin execution. In some instances it is referred to as the *procedure-header*. Rules for naming procedures in Pascal are the same as the rules for naming any other identifier, except that names for all procedures used in any Pascal program must be unique and different from the program name. We do recommend that procedures and functions be named with verb phrases. A verb phrase implies action, and that is the purpose of a procedure or function: to make something happen in a program. The name of a procedure or function should resemble the action performed for a step in an algorithm.

```
   procedure Procedure_Name( { formal parameter list } );

   { list of label declarations known only to this procedure }
   { list of constant declarations known only to this procedure }
   { list of user-defined types known only to this procedure }
   { list of variable declarations known only to this procedure }
   { list of procedure and/or function declarations known only }
   { to this procedure }
   { directive(s) }
   { in-line body }

begin
   { executable body of the procedure }
end;
```

Figure 7.2 Pascal procedure format.

The formal parameter list is similar in concept to the program parameter list, except that it allows us to pass information to the procedure from the point at which the procedure is called, and can allow the procedure to pass information back to its calling point. It implies a path of communication between where the procedure is invoked and where the procedure is being executed. The information transferred along this path can be values for parameters (formal and actual).

Following the procedure-header is the *procedure-body*, composed of a block. A block is represented by a declaration-part followed by a statement-part. In turn, the declaration-part can be composed of a label-declaration part, constant-declaration-part, type-declaration-part, variable-declaration-part, and procedure-and-function-declaration-part.

THINK Pascal offers alternative declarations to those of a block. In THINK Pascal you have the option of choosing two other declarations: directive or inline-body. A *directive* is a special designation for a procedure or function. THINK Pascal supports three directives: external, forward, and override. For THINK Pascal, none of these three words are reserved. *External* indicates that a procedure is declared external to the program unit in which the header appears, whereas *forward* indicates that the body of the procedure is defined by a later declaration. *Override* is special to object-types and is discussed in Chapter 13. An *in-line* declaration has the reserved word **inline** followed by one or more `integer` constants representing machine code. When compiled, the machine code representing the body of the procedure is substituted wherever the procedure is invoked. A non-inline procedure or function is different. Here a single machine instruction is inserted, representing a jump to where the body of the procedure is to be executed. It is important to remember that insertion of the actual machine code is a function of the translator and not of the programmer.

Notice that the **uses** clause does not appear in the context of a Pascal procedure in THINK or Macintosh Pascal. The **uses** clause can only be declared in the main Pascal program and can then be used to borrow data types, procedures, and functions from external Pascal libraries. Unfortunately, Macintosh Pascal has more limitations on what can be borrowed than does THINK Pascal.

As with a the main Pascal program, **begin** and **end** bracket the executable body of a procedure, but a semicolon terminates the definition (declaration) instead of a period. In the main program, a period after the last **end** statement signifies the termination of the complete program unit. Execution of the body of the procedure begins with the first **begin** bracket and ends with the last **end** statement. The parameter list is represented by a set of formal parameters acting as variables local to the procedure. This means that the names associated with formal parameters have no relationship to the same names of other identifiers in either the main body of the program or in other procedures or functions. It is common to refer to them as *dummy variables*.

As an example of applying procedures, consider the following program, titled `Prompt_User`. This program requires several steps that include setting boundaries for the Text window, drawing the Text window, and other numerous repetitive lines of code for displaying the characters ' * ' and ' – ' to the Text window.

```
 program Prompt_User( input, output);
{ Purpose:  Prompts for and allows the entry of a person's full }
{           name, street address, city, and state. }
  var
     Full_Name, Address, City_State: string[30];
     Count: integer;
     Border: Rect;
{ ================== Body for the main program.================ }
begin
{ Hide all of the windows. }
  HideAll;
{ Establish the boundaries for displaying the Text window. }
  SetRect( Border, 10, 40, 400, 250 );
  SetTextRect( Border );
{ Show the Text window for viewing output.}
  ShowText;
{ Display a line representing a header.}
  for Count := 1 to 50 do
     write('*');
```

```
   writeln;
{ Prompt for and enter the full name of a person.}
   write(' Enter full name: ');
   readln( Full_Name );
{ Display a line of characters separating two lines of input. }
   for Count := 1 to 50 do
      write('-');
   writeln;
{ Prompt for a street address. }
   write(' Enter street address: ');
   readln( Address );
{ Display a line of characters separating two lines of input. }
   for Count := 1 to 50 do
      write('-');
   writeln;
{ Prompt for and enter city and state. }
   write(' Enter city and state: ');
   readln( City_State );
{ Display a line representing a trailer. }
   for Count := 1 to 50 do
      write('*');
   writeln;
end.
```

The body of the main program becomes easier to read if we replace the two repetitive **for** statements with a call to a procedure titled `Display_Character_Line` that contains the **for** loop necessary for displaying the string of characters across the Text window.

```
procedure Display_Character_Line( C: char; N : integer );
{ Purpose:  This procedure displays the character C  N times to }
{           the Text window. }
   var
      Count : integer;
begin
{ Repeat displaying a character represented by C to the Text }
{ window. }
   for Count := 1 to N do
      write( C );
   { Terminate the print line. }
   writeln;
end; { Display_Character_Line }
```

The formal parameter list for this procedure is composed of two identifiers: `C`, a `char` type representing the character that is to be displayed in the Text window, and `N`, an `integer` type. Note that the formal parameter name is always followed by a data type indicating the properties that we are associating with the formal parameters. Each formal parameter declaration is separated by a semicolon. This format is similar to the declaration of variables in the main program. Following the procedure-header is a declaration of an identifier called Count, which is local to this procedure and has no direct connection to any other identifier named Count.

The executable body of our procedure Display_Character_Line is represented by a **for** loop and writeln command enclosed within the **begin-end** bracket. This procedure begins execution when the following call is executed from the body of the main program:

```
Display_Character_Line ( '-' , 50 );
```

The two expressions, the character constant '–' and the integer constant 50, in this statement are referred to as *actual parameters*. When this call statement is executed, the main program is interrupted, and control is passed to the procedure named Display_Character_Line. Before the body of the procedure is executed, parameter C is given the character '–' and parameter N is given the integer value 50. The body of the procedure is then executed, and the **for** loop displays N (50) dashes across the Text window. On completion of the **for** loop, the line of text is terminated by a writeln command. Because this is the last executable statement in the body of the procedure, it now terminates execution, and control returns to the body of the main program and to the statement following the call to Display_Character_Line.

An additional procedure called Set_And_Show_Text_Window is also defined. The purpose of this procedure is to hide all windows, establish the rectangular boundary of the Text window, and then display the Text window for viewing lines of text. Notice that the procedure Set_And_Show_Text_Window has no formal parameters. There are no actual parameters, so a call to this procedure is simply given by the format:

```
Set_And_Show_Text_Window ;
```

The following code shows the Pascal statements for this second procedure:

```
procedure Set_And_Show_Text_Window;
{ Purpose:  Sets the boundary of and opens the Text window for }
{           viewing. }
   var
   Border: Rect;
begin
{ Hide all of the windows. }
   HideAll;
{ Establish the boundaries for displaying the Text window. }
   SetRect( Border, 0, 20, 500, 300 );
   SetTextRect( Border );
{ Show the Text window for viewing output.}
   ShowText;
end; { Set_and_Show_Text_Window }
```

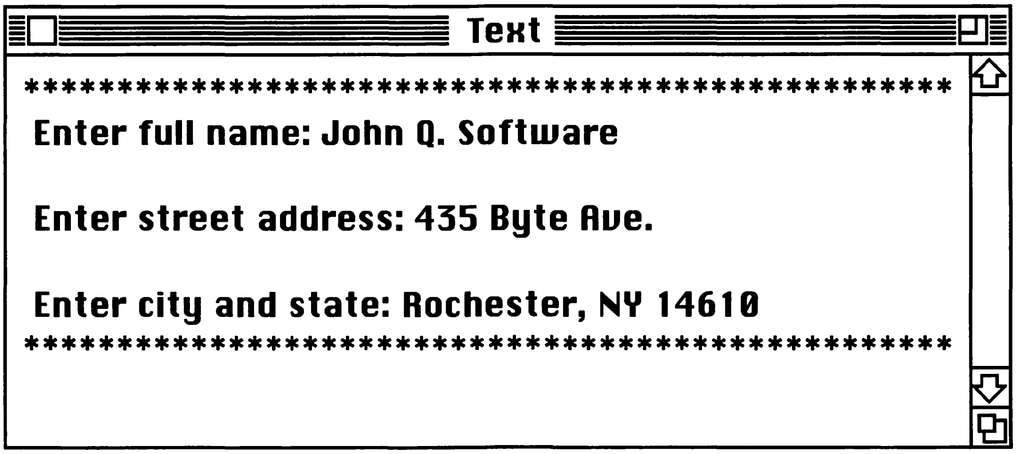
The following example shows the physical locations for both procedures in the Pascal program, as well as the statements for calling each procedure.

```
program Revised_Prompt_User( input, output);
{ Purpose:  Prompts for and allows the entry of a person's full }
{           name,  street address, city, and state. }
```

```
   var
      Full_Name, Address, City_State: string[30];
{ ------------------------------------------------------------------- }
   procedure Display_Character_Line ( C: char; N : integer );
   { Purpose: This procedure displays a character represented }
   { by C  N times to the Text window. }
      var
         Count : integer;
   begin
   { Repeat displaying a character represented by C to the Text }
   { window. }
      for Count := 1 to N do
         write( C );
   { Terminate the print line. }
      writeln;
   end; { Display_Character_Line }
{ ------------------------------------------------------------------- }
   procedure Set_And_Show_Text_Window;
   { Purpose:  This procedure sets the boundary of and opens }
   {           the Text window for viewing. }
      var
         Border: Rect;
   begin
   { Hide all of the windows. }
      HideAll;
   { Establish the boundaries for displaying the Text window. }
      SetRect( Border, 0, 20, 500, 300 );
      SetTextRect( Border );
   { Show the Text window for viewing output.}
      ShowText;
   end; { Set_and_Show_Text_Window }
{ ================ Body for the main program.=================== }

begin
{ Establish boundary of and show the Text window for viewing }
{ output.}
   Set_And_Show_Text_Window;
{ Display a line representing a header.}
   Display_Character_Line( '*' , 50 );
{ Prompt for and enter the full name of a person.}
   write(' Enter full name: ');
   readln( Full_Name );
{ Display a line of characters separating two lines of input. }
   Display_Character_Line( '-' , 50 );
{ Prompt for a street address. }
   write(' Enter street address: ');
   readln( Address );
{ Display a line of characters separating two lines of input. }
   Display_Character_Line( '-' , 50 );
{ Prompt for and enter city and state. }
   write(' Enter city and state: ');
   readln( City_State );
{ Display a line representing a trailer. }
   Display_Character_Line( '*' , 50 );
end.
```

If we count executable lines of code such as **begin**, **end**, write, writeln, readln, **for**, calls to our new procedures Display_Character_Line and Set_And_Show_Text_Window, and calls to library routines such as HideAll, SetRect, SetTextRect and ShowText, we see that the number of executable lines in the body of the main program has been reduced from 24 to 11, a savings of 54%. Comments are not counted, because they are not executable lines; they are only used to provide information when reading the program. The compiler omits these lines when it translates the program from source code into machine code. Although the total number of executable lines remains approximately the same when we include the bodies of our two new procedures, the body of our main program follows more closely the major steps of an algorithm, and fewer lines of code need to be read.

The advantage of writing procedures even though there is only a minimal savings in the number lines of executable code becomes clear when we need to change the code in the body of a procedure: we only need to do it once. Without the use of a procedure, we would have to make the same change several times, at each place where the repetitive code appeared. By using procedures we improve our ability to maintain the software that we have created. Figure 7.3 shows the output from the program Revised_Prompt_User.



```
**************************************************
Enter full name: John Q. Software

Enter street address: 435 Byte Ave.

Enter city and state: Rochester, NY 14610
**************************************************
```

Figure 7.3 Output from the program Revised_Prompt_User.

As a second example of using procedures, consider the program Diagonal_Lines from Chapter 6:

```
program Diagonal_Lines ( input, output);
{Purpose:   This program executes the QuickDraw library }
{           procedures MoveTo and LineTo for drawing four }
{           diagonal lines from a center point.}
begin
{ Open the drawing window for viewing the actions of MoveTo }
{ and LineTo. }
   ShowDrawing;
{ Move the pen to the center of the Drawing window and draw }
{ the first diagonal line. }
   MoveTo(100, 100);
   LineTo(50, 50);
{ Move the pen to the center of the Drawing window and draw }
{ another diagonal line. }
   MoveTo(100, 100);
   LineTo(50, 150);
{ Move the pen to the center of the Drawing window and draw }
{ another diagonal line. }
   MoveTo(100, 100);
   LineTo(150, 50);
{ Move the pen to the center of the Drawing window and draw }
{ the last diagonal line. }
   MoveTo(100, 100);
   LineTo(150, 150);
end.
```

Several lines are composed of the command MoveTo followed by LineTo, so we can reduce the number of executable lines of code in the main body of this program by replacing these commands with calls to a procedure named Draw_Diagonal_Line. The body of this new procedure will contain our two commands MoveTo and LineTo, each command having the formal parameters (arguments) X_Point and Y_Point. The following defines our new procedure:

```
procedure Draw_Diagonal_Line( X_Point, Y_Point : integer );
{ Purpose:  This procedure draws a diagonal line from }
{           the point (100,100) to a point represented }
{           by (X_Point, Y_Point). }
begin
   MoveTo( 100, 100 );
   LineTo( X_Point, Y_Point );
end; { Draw_Diagonal_Line }
```

Notice that the formal parameters X_Point and Y_Point receive values from actual parameters when the procedure is invoked. A revision of Diagonal_Lines follows.

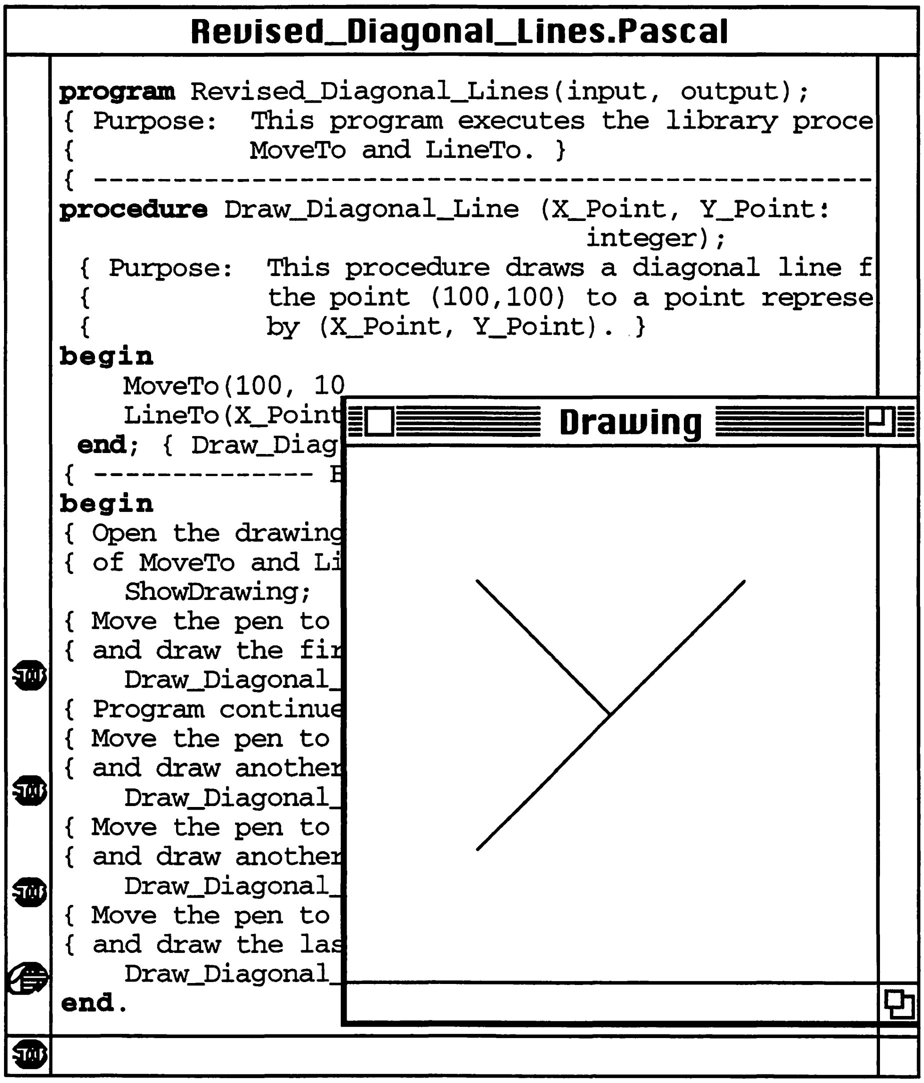
```
program Revised_Diagonal_Lines(input, output);
{ Purpose:  This program executes the library procedures }
{           MoveTo and LineTo. }
{ ---------------------------------------------------------- }
   procedure Draw_Diagonal_Line ( X_Point, Y_Point : integer );
   { Purpose:  This procedure draws a diagonal line from }
   {           the point (100,100) to a point represented }
   {           by (X_Point, Y_Point). }
   begin
      MoveTo( 100, 100 );
      LineTo( X_Point, Y_Point );
   end; { Draw_Diagonal_Line }
{ ----------------- Body of the main program. ---------------- }
begin
{ Open the drawing window for viewing the actions of MoveTo }
{ and LineTo. }
   ShowDrawing;
{ Move the pen to the center of the Drawing window and draw }
{ the first diagonal line. }
   Draw_Diagonal_Line (50, 50);
{ Move the pen to the center of the Drawing window and draw }
{ another diagonal line. }
   Draw_Diagonal_Line (50, 150);
{ Move the pen to the center of the Drawing window and draw }
{ another diagonal line. }
   Draw_Diagonal_Line (150, 50);
{ Move the pen to the center of the Drawing window and draw }
{ the last diagonal line. }
   Draw_Diagonal_Line (150, 150);
end.
```

By using the procedure Draw_Diagonal_Line, we have reduced the number of executable statements in the body of the main program by 36%, even though the total number of lines of code, including the new procedure, remains unchanged. Figure 7.4 shows a screen dump of the revised program, where the **StopsIn** option from the **Debug** menu of THINK Pascal is used to trace the actions of the procedure Draw_Diagonal_Line.

7.1.2 Passing Information to Formal Parameters: Value Parameters

Notice that the formal parameters shown in a procedure-header are given in a list of the formal names followed by a colon, followed by a data type. For example, Draw_Diagonal_Line has two formal parameters, X_Point and Y_Point, both of type integer. The procedure Display_Character_Line has two formal parameters, C and N, C being of type char, and N of type integer. In this context each formal parameter is called a *value* type. That is, each formal parameter receives a value from its corresponding actual parameter upon execution of the procedure. Where are the actual parameters? When a procedure is called, each formal parameter is allocated storage in memory. The value of the corresponding actual parameter is copied and assigned to the storage location of its corresponding formal parameter.

The actual parameter must be represented as an expression, that is, either a constant, a variable, or a combination of one or more constants and variables forming a valid expression. Throughout the execution of a procedure, the value associated with the actual parameter remains unchanged. For example, when executing the calling statement Draw_Diagonal_Line(50, 150), the formal parameters X_Point and Y_Point are allocated storage, with X_Point being given the integer value 50, and Y_Point the integer value 150. The values of these two formal parameters remain unchanged while the procedure Draw_Diagonal_Line is executed. Once the procedure has ended execution, storage for all formal parameters is deallocated. Any values assigned to the formal parameters are lost when the body of the procedure terminates.



Revised_Diagonal_Lines.Pascal

```
program Revised_Diagonal_Lines(input, output);
{ Purpose:  This program executes the library proce
{           MoveTo and LineTo. }
{ --------------------------------------------------
procedure Draw_Diagonal_Line (X_Point, Y_Point:
                              integer);
 { Purpose:  This procedure draws a diagonal line f
 {           the point (100,100) to a point represe
 {           by (X_Point, Y_Point). }
begin
    MoveTo(100, 10
    LineTo(X_Point
 end; { Draw_Diag
{ ------------- E
begin
{ Open the drawing
{ of MoveTo and Li
    ShowDrawing;
{ Move the pen to
{ and draw the fir
    Draw_Diagonal_
{ Program continue
{ Move the pen to
{ and draw another
    Draw_Diagonal_
{ Move the pen to
{ and draw another
    Draw_Diagonal_
{ Move the pen to
{ and draw the las
    Draw_Diagonal_
end.
```

Figure 7.4 Using stops to observe the output from Revised_Diagonal_Lines.

As an additional example of value-type parameters, let us write a procedure for drawing a series of rings, as shown in Figure 7.5. Each ring has equal thickness, and is represented by the value R. When called, the formal parameters of a procedure titled Draw_Concentric_Rings will receive three values. The first represents the x-value of the origin, the second the y-value of the origin, and the third the radius of the first concentric circle. This radius also represents the thickness of each ring. These three values are represented by the formal parameter names X_Point, Y_Point, and R, respectively. The procedure Draw_Concentric_Rings automatically draws a set of rings with radii R, 2 * R, 3 * R, and 4 * R. Let us consider an initial solution for performing these steps:

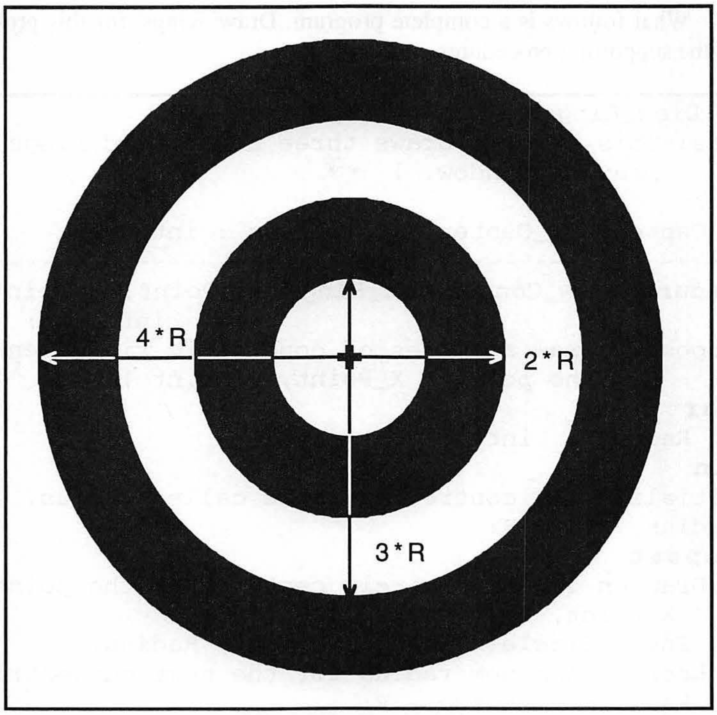


Figure 7.5 The set of circles displayed by the procedure `Draw_Concentric_Circles.`

1. Paint a circle having a radius 4 * R with a color inverse to the present background color.
2. Paint a second circle having a radius 3 * R with a color inverse to the present background color.
3. Paint a third circle having a radius 2 * R with a color inverse to the present background color.
4. Paint the last circle having a radius R with a color inverse to the present background color.

In Pascal, we can express these steps with a simple **repeat-until** loop, using the identifier Radius as a control variable:

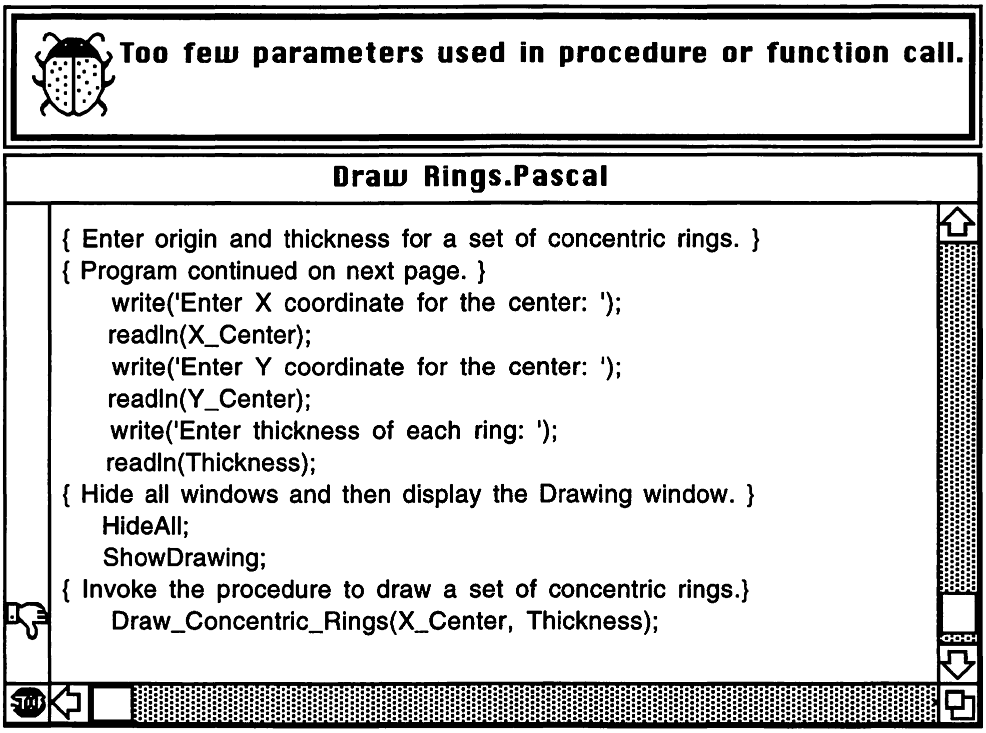
```
{ Initialize the control variable called Radius.}
Radius := 4 * R;
repeat
{ Draw an inverted circle centered at (X_Point , Y_Point) with }
{ value of Radius. }
  InvertCircle ( X_Point , Y_Point, Radius );
{ Establish a new radius for the next concentric circle. }
```

```
   Radius := Radius - R;
until  (Radius <  R);
```

What follows is a complete program, Draw_Rings, for this problem, along with all of the supporting procedures.

```
program Draw_Rings( input, output );
{ Purpose: This program draws three concentric rings to the }
{          Drawing window. }
   var
      X_Center,  Y_Center,  Thickness : integer;
{ ------------------------------------------------------------- }
   procedure Draw_Concentric_Rings (X_Point, Y_Point, R :
                                          integer);
   { Purpose:  Draw a series of concentric rings centered at }
   {           the point ( X_Point, Y_Point ).}
     var
        Radius :  integer;
   begin
   { Initialize the control variable called Radius. }
      Radius := 4 * R;
      repeat
      { Draw an inverted circle centered at the point }
      { ( X_Point, Y_Point  ). }
         InvertCircle(X_Point, Y_Point, Radius);
      { Establish a new radius for the next concentric circle. }
         Radius := Radius - R;
      until (Radius < R);
   end; { Draw_Concentric_Rings }
{ ------------------------------------------------------------- }
   procedure Set_And_Show_Text_Window;
   { Purpose:  This procedure sets the boundary of and opens }
   {           the Text window for viewing. }
      var
         Border: Rect;
   begin
   { Hide all windows from being viewed. }
      HideAll ;
   { Establish the boundaries for displaying the Text window. }
      SetRect( Border, 0, 20, 500, 300 );
      SetTextRect( Border );
   { Show the Text window for viewing output.}
      ShowText;
   end; { Set_and_Show_Text_Window }
{ ================ Body of the main program.================== }
begin
{ Hide all windows and then display the Text window. }
   Set_And_Show_Text_Window;
{ Enter the origin and thickness for a set of concentric rings. }
   write('Enter X coordinate for the center: ');
   readln(X_Center);
   write('Enter Y coordinate for the center: ');
   readln(Y_Center);
   write('Enter thickness of each ring: ');
   readln(Thickness);
{ Hide all windows and then display the Drawing window. }
   HideAll ;
   ShowDrawing;
{ Invoke the procedure to draw a set of concentric rings.}
   Draw_Concentric_Rings(X_Center, Y_Center, Thickness);
 end.
```

When writing any procedure, it is important to remember that the data types of the actual parameters must agree with each of their corresponding formal parameters. For example, X_Center is of type integer, so the formal parameter X_Point must also be of type integer. In addition, the procedure-header must also have three formal parameters to match the number of actual parameters. If the types fail to match or the number of formal parameters fails to match the number of actual parameters, the program will fail to compile. Figure 7.6 shows a portion of the program Draw_Rings with the statement calling the procedure Draw_Concentric_Rings changed. The actual parameter Y_Center is missing, which results in an error message.



Too few parameters used in procedure or function call.

Draw Rings.Pascal

```
{ Enter origin and thickness for a set of concentric rings. }
{ Program continued on next page. }
    write('Enter X coordinate for the center: ');
    readln(X_Center);
    write('Enter Y coordinate for the center: ');
    readln(Y_Center);
    write('Enter thickness of each ring: ');
    readln(Thickness);
{ Hide all windows and then display the Drawing window. }
   HideAll;
   ShowDrawing;
{ Invoke the procedure to draw a set of concentric rings.}
    Draw_Concentric_Rings(X_Center, Thickness);
```

Figure 7.6 An example of an error message for too few parameters.

When the procedure Draw_Concentric_Rings is invoked, the value of the actual parameter X_Center is transferred to its corresponding formal parameter, X_Point; the value of the actual parameter Y_Center is transferred to its corresponding formal parameter, Y_Point; and the value of the actual parameter Thickness is transferred to its corresponding formal parameter, R. What follows is the execution the body of the procedure. When the last **end** statement in the procedure definition is executed, the Pascal run-time system returns control to the statement following the statement that called the procedure. From this point, the Pascal program continues to execute. In our example this happens to be the end of the program. While we can see that formal parameters of value types can be used as constants, value type parameters can also be used as variables local to a procedure.

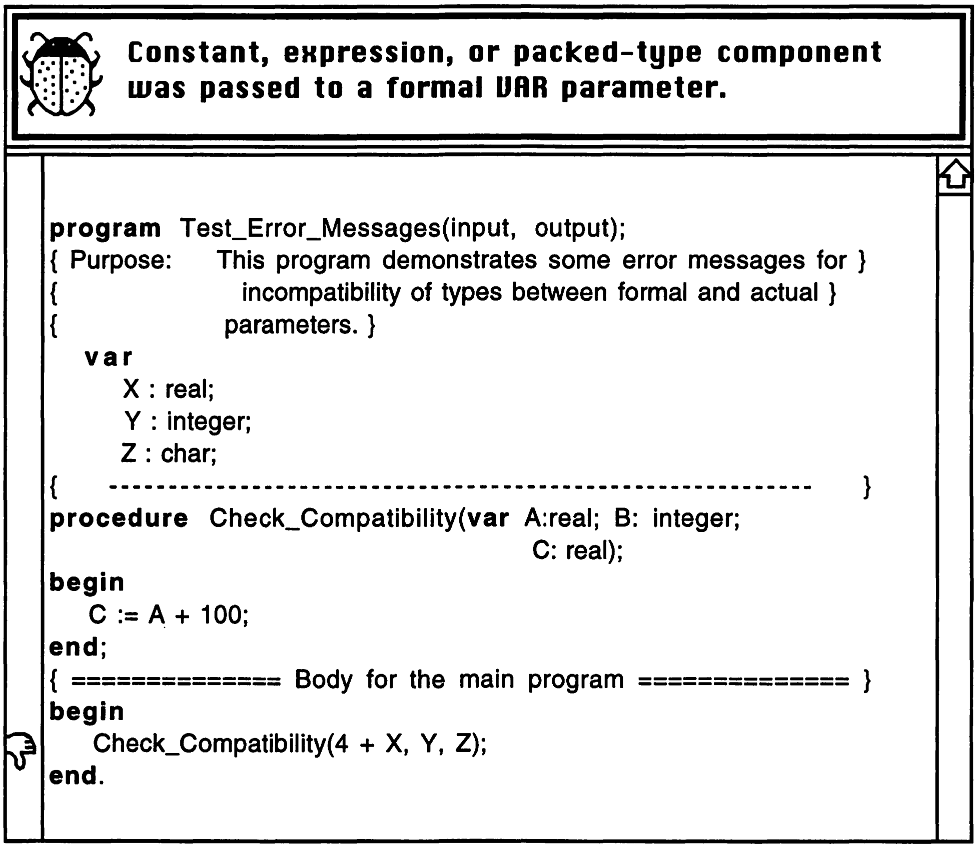
When a procedure is called, a formal parameter of a value type receives its initial value from its corresponding actual parameter. After this point, the procedure can include statements in its body that alter the value of the formal parameter. For example, consider the definition of the procedure Revised_Display_Character_Line:

```
procedure Revised_Display_Character_Line( C : char; N :
                                            integer);
{Purpose:   This procedure displays the character  C   N times }
{           across the Text window.}

begin
{ Display N characters across the Text window. }
   while N > 0 do
      begin
         write( C );
         N := N - 1
      end;
{ Terminate the print line. }
   writeln;
end;
```

As you can see, the formal (value) parameter N is used as a variable that is local to the body of the procedure. Instead of using the local variable Count, the parameter N is used as a control parameter for a **while** loop. Each time the **while** loop is executed, the value of N is decremented by 1, and once the value of N reaches zero, execution of the loop is terminated. While the value of the formal parameter changes as the loop is executed, the value of its corresponding actual parameter remains unchanged.

7.1.3 Passing Information with Formal Parameters: Variable Parameters

Procedures can be used to produce side effects on the values of actual parameters. This requires that the formal parameter of a corresponding actual parameter be declared a *variable* type by using the keyword **var** preceding the formal parameter name in the formal parameter list of the procedure-header. Within the body of the procedure, any reference to the formal variable parameter is in reality a reference to the memory location of its corresponding actual parameter. For any formal variable parameter, the corresponding actual parameter can only be a variable. If the actual parameter is a constant, or an expression other than a simple variable, an error message such as the one shown in Figure 7.7 will appear. In this example the formal variable parameter called A has for its corresponding actual parameter the expression 4 + X. This is improper syntax for any Pascal program. Notice that THINK Pascal catches this error during the compile step and in reporting the error points to the statement where the procedure is invoked.



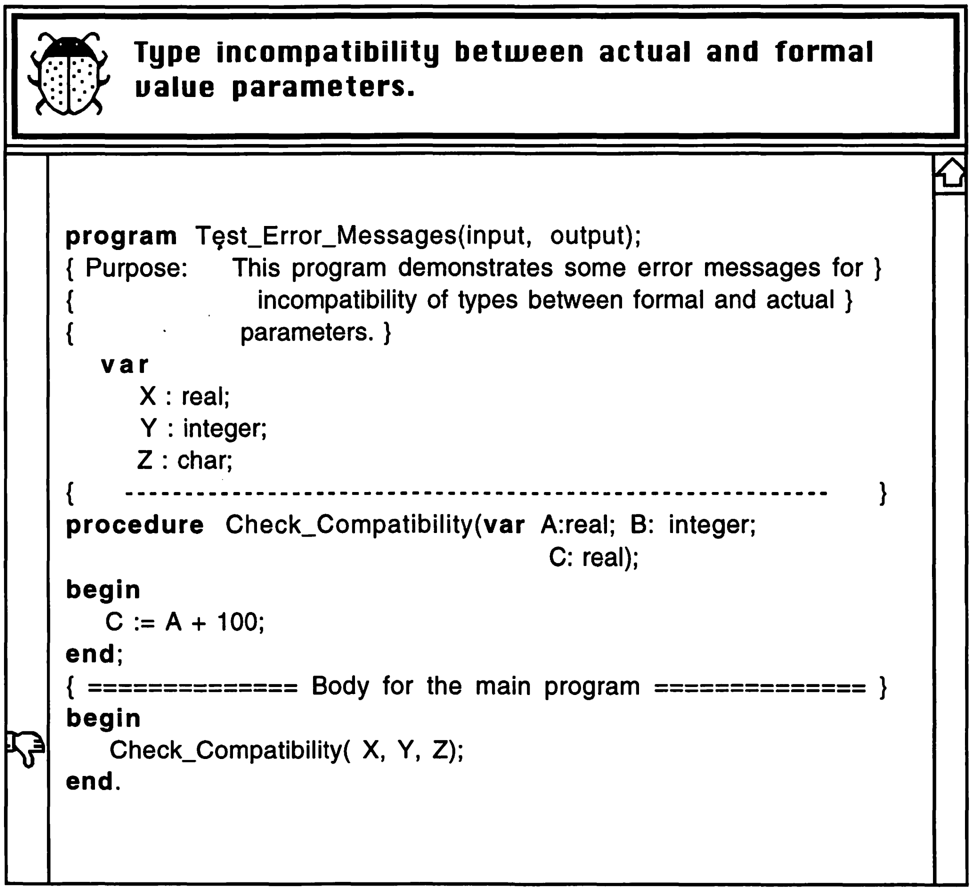
Constant, expression, or packed-type component was passed to a formal VAR parameter.

```
program Test_Error_Messages(input, output);
{ Purpose:   This program demonstrates some error messages for }
{              incompatibility of types between formal and actual }
{             parameters. }
  var
    X : real;
    Y : integer;
    Z : char;
{ -----------------------------------------------------------  }
procedure Check_Compatibility(var A:real; B: integer;
                                        C: real);
begin
  C := A + 100;
end;
{ ============= Body for the main program ============= }
begin
   Check_Compatibility(4 + X, Y, Z);
end.
```

Figure 7.7 The error message resulting from a formal variable parameter having an expression as its actual parameter.

It is important to understand that the data type for any formal parameter (value or variable) and its corresponding actual parameter be assignment-compatible. Figure 7.8 shows an example of type incompatibility between the formal parameter named `C` and the actual parameter `Z`. Again, the THINK Pascal compiler catches this error during the compile step by reporting the error and pointing to the statement that invokes the procedure. In Macintosh Pascal, type-checking of parameters for assignment compatibility is performed *only* when the program is executing, not when the program is being checked for proper syntax. If the data types fail to match, the program halts execution and reports an error.

Data types for formal parameters can be standard ordinal types such as `integer`, `longint`, `Boolean`, user-defined types for declared `enumerated` and `subrange` types, `real` types, or `string` types. All of these data types were discussed in detail in Chapter 3. Where a formal parameter is a `string` type, it can only be declared as a `string` and not as a `string` having a specific size. Including the attribute for string size results in a syntax error when the program is being checked. For a formal value parameter declared as a `string` type, the string size defaults to 255, even though the actual parameter has a smaller size. For a formal variable parameter declared as a `string` type, the size attribute of the formal parameter is that of its corresponding actual parameter. In addition to the requirement for assignment compatibility between corresponding actual and formal parameters, the number of formal parameters must agree with the number of actual parameters. In THINK Pascal this count is tested *only* when the syntax of the program is being checked; in Macintosh Pascal it is confirmed only when the program is being executed.



Type incompatibility between actual and formal value parameters.

```
program Test_Error_Messages(input, output);
{ Purpose:   This program demonstrates some error messages for }
{              incompatibility of types between formal and actual }
{            parameters. }
  var
    X : real;
    Y : integer;
    Z : char;
{ ---------------------------------------------------------- }
procedure Check_Compatibility(var A:real; B: integer;
                                      C: real);
begin
  C := A + 100;
end;
{ ============= Body for the main program ============= }
begin
  Check_Compatibility( X, Y, Z);
end.
```

Figure 7.8 An error mesage resulting from type incompatibility between the actual and formal parameters.

As an example of a procedure using variable parameters, let us consider a program that will prompt for three pieces of information— the full name of a person, tax number, and gross income— and write this information to an output file. Later this output file will be opened by a spreadsheet application to perform additional operations. The main program requires the following three steps:

1. Hide all windows, and then set and open the Text window for viewing information entered from the keyboard. This will be performed by a separate procedure named `Set_And_Show_Text_Window`, a parameterless procedure.
2. Prompt the user for a file name. This file name should be a string that will be passed back from a procedure called `Prompt_For_File_Name`. It will have only one parameter, the variable parameter for returning a file name.
3. Pass the file name to a third procedure called `Write_To_Data_File`. This procedure will be responsible for opening the file given by the file name from Step 2. This procedure continues to prompt for the three required pieces of information until the user is ready to quit. When the entry of data ends, the output file will be closed, and the procedure will terminate its execution and return control to the main program.

To make the program appear modular, the third procedure `Write_To_Data_File` will invoke a fourth procedure called `Prompt_For_Data`. This procedure will read three pieces of information entered from the keyboard: a person's full name, a person's tax number, and a person's gross income. All three pieces of information will be passed back to procedure `Write_To_Data_File`, so that the information can be passed on to the output file.

Figure 7.9 presents a special diagram for showing the relationship between procedures.

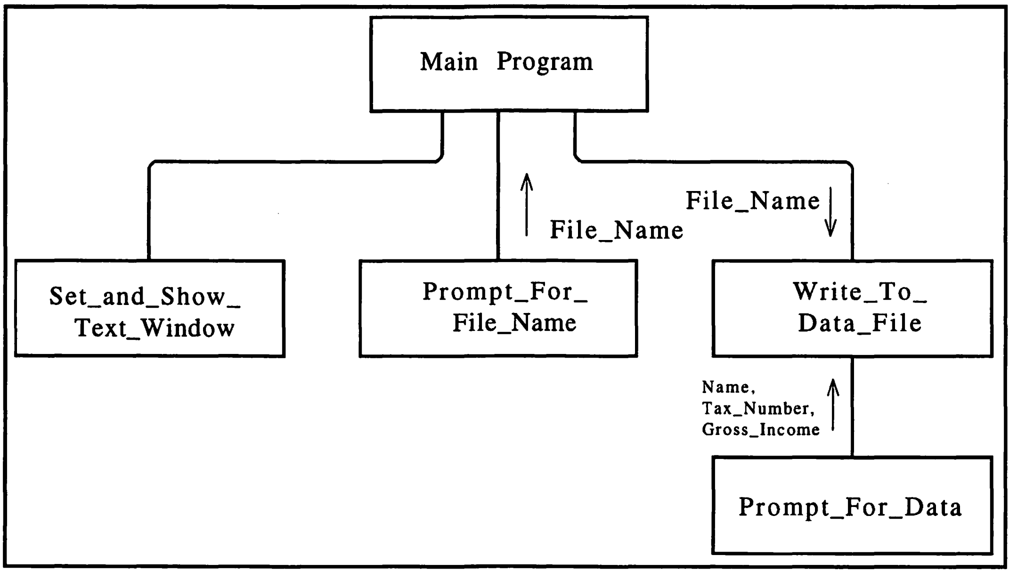


Figure 7.9 A structure chart showing the relationship between procedures and the passing of data between procedures.

This diagram is called a *structure chart.* The rectangle at the top represents the main program, and the rectangles below represent procedures that the main program depends upon. The lines connecting the rectangles represent the dependencies of the main program and its supporting procedures. In Figure 7.9 the body of the main program represents a superordinate body, and procedures Write_To_Data_File, Set_And_Show_Text_Window, and Prompt_For_File_Name are subordinate to main. In turn, the procedure Prompt_For_Data is subordinate to procedure Write_To_Data_File, because it serves to accept data and return it to its superordinate procedure.

The directed arrows in the structure chart show the direction in which information is sent. For example, the procedure Prompt_For_File_Name returns a copy of the file name entered from the keyboard. In our structure chart it is labeled File_Name. The arrow for File_Name points upward, telling us that this procedure returns information to the main program. In turn, the structure chart shows our file name being sent to procedure Write_To_Data_File, with nothing being returned by this procedure. The procedure Prompt_For_Data returns three pieces of data: the name of a person, the tax number, and the gross income. Here the arrow points upward, indicating the direction in which information flows.

The Pascal code for procedure Prompt_For_File follows.

```
procedure Prompt_For_File_Name (var File_Name: string);
{Purpose:   This procedure prompts for a file name and returns }
{           that name through the formal parameter Name. }
begin
{ Prompt user for a file name. }
   write(' Enter a file name for storing data: ');
   readln(File_Name);
end; { Prompt_For_File_Name }
```

The procedure prompts for and allows the user to enter a file name as a string. Notice that the formal parameter File_Name is a formal variable parameter and is used to return the actual file name to the body of the main program.

The procedure Write_To_Data_File requires several major steps. First, it will be sent the file name as a string and will use the file name to open an actual physical file by execution of the rewrite command. Second, it will write a header to the output file containing the labels Name, Tax Number, and Gross Income. A horizontal tab must be written between each label, because any spreadsheet reading a text file always treats a horizontal tab as a means to separate any column. For any spreadsheet reading a text file, an end-of-line always causes the spreadsheet to begin a new row. Third, a **repeat-until** loop is executed to continue prompting the user for information, and writing this information to the output file. Each time the loop is executed, the user is given the opportunity to quit by typing either the character 'Q' or 'q'. If any other character is entered, execution of the loop continues. The last step after leaving the loop is to close the output file. The following is the Pascal code for this procedure.

```
procedure Write_To_Data_File (File_Name: string);
{ Purpose : This procedure will open a file for writing data, }
{           and then write data to an output file }
{           represented by the variable parameter Output_File. }
   var
      Output_File: Text;
      Name, Tax_Number: string;
      Tab, Response: char;
      Gross_Income: real;
```

```
begin
{ Initialize the value for a horizontal tab. }
   Tab := chr(9);
{ Open the output file for writing data.  }
   rewrite(Output_File, File_Name);
{ Write column headers to the output file. }
   writeln(Output_File, 'Name', Tab, 'Tax Number', Tab, 'Gross
               Income');
{ Continue to prompt for data until the user is ready to quit. }
   repeat
   { Prompt for and enter name, tax number, and gross income. }
      Prompt_For_Data(Name, Tax_Number, Gross_Income);
   { Write these three entries to the data file. }
      write(Output_File, Name, Tab, Tax_Number, Tab);
      writeln(Output_File, Gross_Income : 6 : 2);
   { Prompt the user to either continue or quit. }
      write(' Press `Q` to quit, `C` to continue: ');
      readln(Response);
   until ((Response = 'Q') or (Response = 'q'));
{ Close the output file before exiting this procedure. }
   close(Output_File);
end; { Write_To_Data_File }
```

Notice that this procedure has only one formal variable: a value parameter named File_Name. This procedure needs only to receive a final name and does not change the way the name was entered, so the use of a formal variable parameter is unnecessary.

The last procedure is Prompt_For_Data, a simple procedure that only requires sequential execution of write and readln statements. As you can see in the code that follows, all input to this procedure comes from typing information at the keyboard. Both the formal parameters Name and Number are strings, because a tax number might be entered with characters other than digits. Notice that all three formal parameters are variable types, because they return values to the procedure Write_To_Data_File.

```
procedure Prompt_For_Data(var Name, Number: string; var
Income: real);
{ Purpose: This procedure prompts for and allows the entry of a }
{          person's full name, tax number, and gross income. }
begin
{ Prompt for person's name, tax number, and gross income. }
   writeln(' ---------------------------------------------- ');
   write(' Enter the last name of the person: ');
   readln(Name);
   write(' Enter the person`s tax number: ');
   readln(Tax_Number);
   write(' Enter the person` s gross income : $ ');
   readln(Income);
end; { Prompt_For_Data }
```

We can write the formal parameter list in procedure Prompt_For_Data in a different format from what is given above. For example, the following format is slightly longer but still declares all formal parameters to be variable types:

```
procedure  Prompt_For_Data(var Name: string; var Number:string;
                           var Income : real ) ;
```

If we failed to place a **var** before Number and Income, as in the following header,

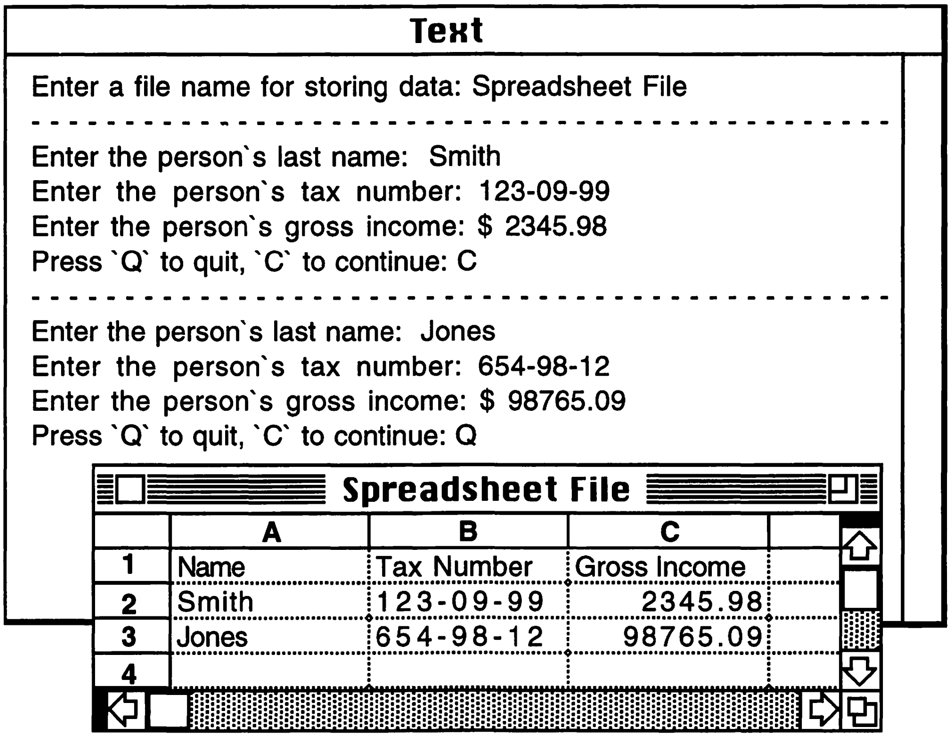
```
procedure  Prompt_For_Data( var Name  :  string ; Number  :  string;
                               Income : real ) ;
```

the side effects from executing Prompt_For_Data would not change the values of the actual parameters associated with Number and Income. Why? Although Name is a variable type, Number and Income are value-type parameters (since a **var** does not precede their names) and as such cannot change the values of their actual parameters. Simply starting the declaration with the keyword **var** does not make all other formal parameters variable types. While the procedure would execute, no values for Number and Income would be returned to their corresponding actual parameters, Tax_Number and Gross_Income. It is important to place the keyword **var** in the proper position to assure the declaration of a formal parameter as a variable type.

Why is it necessary to invoke the last procedure, Prompt_For_Data, from Write_To_Data_File? Would it not be easier to place the steps of this procedure in the body of Write_To_Data_File? Our purpose is to keep the procedure Write_To_Data_File highly functional (strongly cohesive). Its major function is to open a file for writing, and to write information to that file. Having a separate procedure to prompt and read data from the keyboard keeps the procedure Write_To_Data_File dedicated to its major purpose. Having a separate procedure such as Prompt_For_Data makes changing the body of this procedure easier, because the procedure is short and readable, and because it satisfies one major function: it prompts a user and accepts data entered from the keyboard.

A complete listing of this program follows. Figure 7.10 shows two windows: a Text window containing data from the keyboard, and a window representing a window in Excel, where our output file has been opened and displayed. The execution of the spreadsheet application is not a requirement of our program.

```
program Text_File_Program(input, output);
{Purpose :  This program creates a text file with three columns }
{           and several rows of information. The first column }
{           lists names, the second tax numbers, and the third, }
{           gross income. Once the file is closed, it can be }
{           opened by a spreadsheet application such as Excel. }
  var
     File_Name: string;
{ ------------------------------------------------------------- }
  procedure Prompt_For_File_Name (var File_Name: string);
  { Purpose:  This procedure prompts for file name and returns }
  {           that name through the formal parameter Name.}
{ Program listing continues after Figure 7.10 }
```



```
Enter a file name for storing data: Spreadsheet File
-----------------------------------------------------
Enter the person`s last name:  Smith
Enter the person`s tax number: 123-09-99
Enter the person`s gross income: $ 2345.98
Press `Q` to quit, `C` to continue: C
-----------------------------------------------------
Enter the person`s last name:  Jones
Enter the person`s tax number: 654-98-12
Enter the person`s gross income: $ 98765.09
Press `Q` to quit, `C` to continue: Q
```

	A	B	C
1	Name	Tax Number	Gross Income
2	Smith	123-09-99	2345.98
3	Jones	654-98-12	98765.09
4			

Figure 7.10 A display of the data generated by `Text_File_Program`. A spreadsheet file has been opened by Excel and is shown in the active window.

```
{ Continuation of listing of Text_File_Program }
   begin
   { Prompt user for a file name. }
      write(' Enter a file name for storing data: ');
      readln(File_Name);
   end; { Prompt_For_File_Name }
{ ------------------------------------------------------------ }
   procedure Prompt_For_Data (var Name, Number: string; var
                                        Income: real);
   { Purpose:  This procedure prompts for and allows the entry }
   {           of a person's full name, tax number, and gross }
   {           income. }
   begin
   { Prompt for person's name, tax number, and gross income. }
      writeln(' ---------------------------------------------- ');
      write(' Enter the last name of the person: ');
      readln(Name);
      write(' Enter the person`s tax number: ');
      readln(Number);
      write(' Enter the person` s gross income : $ ');
      readln(Income);
   end; { Prompt_For_Data }

{ -------------------------------------------------------------- }
   procedure Write_To_Data_File (File_Name: string);
   { Purpose : This procedure will open a file for writing }
   {           data, and then write data to an output }
   {           file represented by the variable parameter }
   {           Output_File. }
   var
         Output_File: Text;
         Name, Tax_Number: string;
         Tab, Response: char;
         Gross_Income: real;
   begin
   { Initialize the value of Tab for a horizontal tab. }
      Tab := chr(9);
   { Open the output file for writing data. }
      rewrite(Output_File, File_Name);
   { Write column headers to the output file.}
      writeln(Output_File, 'Name', Tab, 'Tax Number', Tab, 'Gross
               Income');
   { Continue to prompt for data until the user is ready }
   { to quit. }
      repeat
      { Prompt for and enter name, tax number, and gross income. }
         Prompt_For_Data(Name, Tax_Number, Gross_Income);
      { Write these three entries to the data file. }
         write(Output_File, Name, Tab, Tax_Number, Tab);
         writeln(Output_File, Gross_Income : 6 : 2);
     { Prompt the user to either continue or quit. }
         write(' Press `Q` to quit, `C` to continue: ');
         readln(Response);
      until ((Response = 'Q') or (Response = 'q'));
   { Close the output file before exiting this program. }
      close(Output_File);
   end; { Write_To_Data_File }
 { ------------------------------------------------------------- }
   procedure Set_And_Show_Text_Window;
   { Purpose:  This procedure sets the boundary of and opens }
   {           the Text window for viewing. }
      var
         Border: Rect;
   begin
   { Hide all windows before establishing and showing the Text }
   { window. }
      HideAll;
   { Establish the boundaries for displaying the Text window. }
      SetRect(Border, 10, 40, 400, 250);
      SetTextRect(Border);
   { Show the Text window for viewing output.}
      ShowText;
   end; { Set_And_Show_Text_Window }
{ ================= Body for the main program. ================ }
begin
{ Hide all windows and then open the Text window for viewing }
{ prompts and responses. }
   Set_And_Show_Text_Window;
{ Prompt user for a file name. }
   Prompt_For_File_Name(File_Name);
{ Now write data to the output file represented by file name.}
   Write_To_Data_File(File_Name);
end.
```

Before ending this section, we must make one additional comment. The way the Text_File_Program has been written makes it important to declare the procedure Prompt_For_Data before Write_To_Data_File. Procedure Write_To_Data_File invokes Prompt_For_Data, so declaring procedure Prompt_For_Data after Write_To_Data_File results in a compile-error message indicating that the name "Prompt_For_Data" has not been declared. The error occurs in the body of Write_To_Data_File where Prompt_For_Data is invoked.

An additional example, where values of actual parameters are affected by actions on corresponding formal parameters, is Utility_Program, which features three procedures: Computation_On_Average_Cost, Report_On_Cost, and Prompt_User_To_Continue.

```
program Utility_Program(input, output);
{ Purpose:  This program computes the average cost and }
{           consumption for 12 utility bills and reports the }
{           average values of consumption and cost in the Text }
{           window. }
    var
      AverageCost,  AverageConsumption : real;
      Answer : char;
{ ------------------------------------------------------------- }
   procedure Set_And_Show_Text_Window;
   { Purpose:  This procedure sets the boundary of and opens }
   {           the Text window for viewing. }
      var
         Border: Rect;
   begin
   { Hide all windows before establishing and showing the Text }
   { window. }
      HideAll;
   { Establish the boundaries for displaying the Text window. }
      SetRect(Border, 10, 40, 400, 250);
      SetTextRect(Border);
   { Show the Text window for viewing output.}
      ShowText;
   end; { Set_And_Show_Text_Window }
{ ------------------------------------------------------------ }
   procedure Computation_Of_Average_Cost(var Average_Cost,
                              Average_Consumption : real);
  { Purpose:  Enter 12 months of cost and consumption values }
  {           from the keyboard. Then compute the average cost }
  {           and average consumption. }
     var
        Counter, Total_Consumption, Total_Cost : integer;
        Consumption,  Cost : integer;
  begin
  { Initialize the totals and control variable Counter. }
     Counter := 1;
     Total_Consumption := 0;
     Total_Cost := 0;
  { Repeat entry of consumption and cost data until counter }
  { exceeds 12.}
     repeat
     { Prompt for and accept data monthly consumption and cost }
     { from the keyboard. }
        writeln;
        write('Enter consumption: ');
        readln(Consumption);
        write('Enter cost:  $ ');
        readln(Cost);
     { Compute partial sums for consumption and cost. }
        Total_Consumption := Total_Consumption + Consumption;
        Total_Cost := Total_Cost + Cost;
        Counter := Counter + 1;
     until (Counter > 12);
  { Compute the average values for consumption and cost. }
     Average_Consumption := Total_Consumption / 12;
     Average_Cost := Total_Cost / 12;
   end; { Computation_Of_Average_Cost }
{ --------------------------------------------------------------- }

  procedure Report_On_Cost( Average_Cost, Average_Consumption
                                 : real);
  { Purpose:  This procedure reports on the average cost and }
  {           consumption.}
  begin
  { Display the average values, cost and consumption, to the }
  { screen.}
     writeln;
     writeln('Average monthly consumption: ', Average_Consumption
             : 7 : 2);
     writeln('Average monthly cost:  $', Average_Cost : 6 : 2);
  end; { Report_On_Cost }
 { -------------------------------------------------------------- }
  procedure Prompt_User_To_Continue(var Response : char);
  { Purpose:  This procedure prompts the user to see if he/she }
  {           wishes to enter an additional set of values. }
   begin
  { Prompt the user to repeat the entry of data or to quit.}
```

```
      writeln;
      writeln('Do you wish to enter another set of data?');
      write('Enter "Y" for yes, "N" for no, and then press the
               Return key:');
      readln(Response);
   end; { Prompt_User_To_Continue }
{ ================ Body of the main program. ================== }
begin
{ Hide all windows and then open the Text window for viewing }
{ prompts and responses. }
   Set_And_Show_Text_Window;
{ Repeat entering sets of data until user decides to quit. }
   repeat
   { Enter set of cost and consumption values. }
      Computation_Of_Average_Cost(AverageCost,AverageConsumption);
   { Report average cost and consumption and prompt user to }
   { continue. }
      Report_On_Cost(AverageCost, AverageConsumption);
   { Prompt user to either quit or enter another set of data. }
      Prompt_User_To_Continue(Answer);
   until(Answer = 'N');
end.
```

Figure 7.11 shows a structure chart for this example. This program is based on an early program called `Electric_Bill`, but with the instructions in the body of the

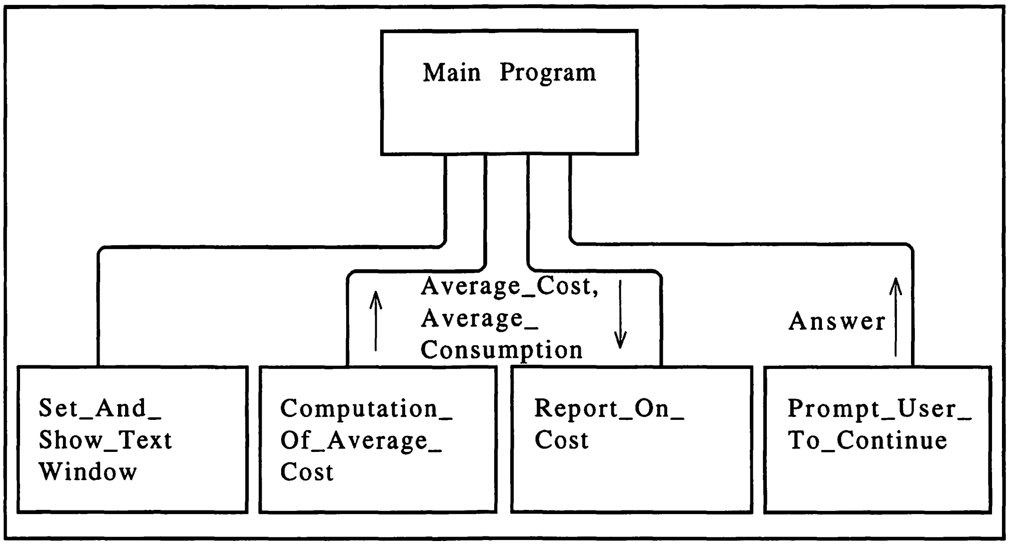


Figure 7.11 A structure chart for `Utility_Program`, showing the relationships between procedures and the passing of parameters between main program and procedures.

main program divided among three procedures: `Computation_Of_Average_Cost`, `Report_On_Cost`, and `Prompt_User_To_Continue`. Notice that the formal parameters `Average_Cost` and `Average_Consumption` of procedure `Computation_Of_Average_Cost` have a one-to-one correspondence with their actual parameters `AverageCost` and `AverageConsumption`, respectively. Both formal parameters are variable types. When either of the formal parameters `AverageCost` or `AverageConsumption` is assigned a new value by an assignment statement, the value of the corresponding actual parameter is also changed. If we had forgotten to place **var** before the formal parameters, they would be value types, and there would be no side effects on the corresponding actual parameters. Whatever values the actual parameters had before execution of the procedure `Computation_Of_Average_Cost` would remain. The second procedure, `Report_On_Cost`, has two formal parameters, `Average_Cost` and `Average_Consumption`, both being value types. Why? Because we are only interested in passing the values of the actual parameters `Average_Cost` and `Average_Consumption` to procedure `Report_On_Cost`, there is no need to change these values during the execution of this procedure.

The third procedure, `Prompt_User_To_Continue`, prompts the user to either repeat the actions of the program or quit. The `readln` statement in this procedure assigns a value to the formal parameter `Response`. Because `Response` is a formal variable parameter, it passes this value back to the actual parameter called `Answer`. If we had not placed **var** before `Response`, this formal parameter would be a value type, and the value of the actual parameter `Answer` would remain unchanged. Unfortunately, the program would never terminate execution, because the proper response for quitting could never be returned.

Following is a brief review of some of the concepts we have discussed in relation to procedures and parameters.

1. Procedures are used for their effects. In some instances these may include changing the values of actual parameters while the procedure is executing; in other instances they simply report information.
2. The link binding the actual parameter list and the formal parameter list provides a path by which information can pass between where the procedure is called and where the procedure is executed. This link is broken once the procedure ends execution.
3. The data type of any formal parameter must agree with its corresponding actual parameter, and the number of formal parameters must agree with the number of actual parameters.
4. A value for an actual parameter can be returned from its corresponding formal parameter only if the formal parameter has been declared a variable type in the formal parameter list.
5. An actual parameter that passes a value to its corresponding formal parameter can be either a simple variable or an expression.
6. An actual parameter that can both pass and receive a value from its corresponding formal parameter can only be a simple variable. It cannot be an expression. If it is, it results in a compilation error.
7. Formal parameters are always represented by simple variables and can never be represented as expressions.
8. When a procedure is called, the present executable environment is interrupted, the link between actual and formal parameters is created, and the body of the procedure is executed.

9. When the last **end** of the procedure is executed, the link between the actual and formal parameters is broken, and execution resumes at the statement following the statement that called the procedure.
10. Procedures can also be defined without a formal parameter list. In this case the procedure-header must have the form

```
procedure Procedure_name;
```

Declaring a header in the form

```
procedure Procedure_name();
```

results in an error message indicating an invalid formal parameter list.

11. With procedures, a Pascal program becomes a series of highly functional steps, because each procedure is written to serve one or two major activities. The body of the main program becomes an excellent description of the solution to a problem, with the detail of each step left to the body of the procedure executing the step.
12. It is important to write a procedure as a highly cohesive unit. Its level of cohesion must be chosen in terms of the solution to the problem and of the purpose of the procedure.

7.2 PASCAL FUNCTIONS

A Pascal function is similar in concept and format to a Pascal procedure. Syntactically, the format for declaring a function is shown in Figure 7.12.

```
function Function_Name ( {formal parameter list} ): result-type;

{ list of label declarations known only to this function }
{ list of constant declarations known only to this function }
{ list of user-defined types known only to this function }
{ list of variable declarations known only to this function }
{ list of procedure and/or function declarations known only }
{ to this function }
{ directive }
{ in-line body }

begin
{ executable body of the function }
end;
```

Figure 7.12 Pascal function format.

There are four major differences between a Pascal function and a Pascal procedure. First, a value having a data type is returned through the name of the function. This implies that the name of the function has data associated with the result-type specified in the header.

Second, the name of the function can receive a value only if the function name is assigned a value in the executable body of the function itself and before the function terminates execution. Rules for naming a function are the same as for any other identifier in Pascal. Third, a function can be used both for its effect and for returning a value, because the formal parameter list of the function-header can contain both value and variable-type parameters. Fourth, a function can only be called from within an expression; it cannot be invoked like a procedure.

As an example of writing a function, consider the following problem. Standard Pascal does not support an exponentiation operator, that is, an operator for raising a value x to a power y. To provide this capability, we will develop a function called `Exponentiation` for taking an `integer`value x and raising it to an `integer` value y. Here we assume that y must be positive. First, let us consider some special cases for values of x and y, and if the conditions are proper, the basic steps for computing x^y :

1. If y is negative, or if both x and y are zero, then no result exists. For convenience we will define the result to be zero. The need to report an error message is a matter of judgment.
2. If y is zero, the result is 1, because x at this step must be nonzero.
3. Otherwise, the result is the product of x multiplied y times.

Here is a more detailed algorithm for this function:

```
function Exponentiation { returns an integer value };
{ Comment on variables:
  X, Y are formal parameters whose values are passed into this
  function. Product and Counter are integer variables local to
  the executable body of this function. }
begin
{ First, check for special conditions on X and Y. }
   if ( Y < 0 ) or (( X=0) and (Y=0 ))  then
      begin
   { If important display a message that no result can exist.}
         Exponentiation <-- 0
      end
   else  { Y must be  greater than or equal to zero. }
      if ( Y = 0 )  then  { X must be nonzero }
         Exponentiation <-- 1
      else
      { Initialize control variable Counter and the partial
         product Product, and compute X raised to power Y.  }
         begin
            Counter <-- 1 ;
            Product <-- 1;
            repeat
               Product <-- Product * X;
               Counter  <-- Counter + 1;
            until ( Counter > Y );
            Exponentiation <-- Product
         end;
end;  { Exponentiation }
```

The program `Testing_Exponentiation` shows our algorithm as a Pascal function. Note that the function `Exponentiation` is invoked from an expression in the body of the main program, and that the value returned from function `Exponentiation` is assigned to a variable called `Result`.

```
program Testing_Exponentiation(input, output);
{ Purpose:  This program allows the testing of the function }
{           Exponentiation. }
   var
      X, Y, Result : integer;
      Response : char;
{ --------------------------------------------------------------- }
   procedure Set_And_Show_Text_Window;
   { Purpose:  This procedure sets the boundary of and opens }
   {           the Text window for viewing. }
      var
         Border: Rect;
   begin
   { Hide all windows before establishing and showing the Text }
   { window. }
      HideAll;
   { Establish the boundaries for displaying the Text window. }
      SetRect(Border, 10, 40, 400, 250);
      SetTextRect(Border);
   { Show the Text window for viewing output.}
      ShowText;
   end; { Set_And_Show_Text_Window }
{ --------------------------------------------------------------- }
   function Exponentiation ( X, Y : integer) : integer;
   { Purpose:  This function raises the value of X to the }
   {           power Y. }
      var
         Counter, Product : Integer;
   begin
   { First, check for special conditions on X and Y. }
      if (Y < 0) or ((X = 0) and (Y = 0)) then
         begin
            writeln;
            write(' No result exists, since either Y < 0 or');
            writeln(' both X and Y are 0.');
            Exponentiation := 0
         end
      else if (Y = 0) then { X must be  nonzero }
         Exponentiation := 1
      else
      { Initialize control variable Counter and temporary }
      { variable Product, and compute X raised to the power Y. }
         begin
            Counter := 1;
            Product := 1;
            repeat
```

```
              Product := Product * X;
              Counter := Counter + 1;
              until (Counter > Y);
           Exponentiation := Product
        end
   end; { Exponentiation }
{ ================= Body of the main program. ================= }
begin { Body of the main program. }
{ HideAll windows and then set and show the Text window for }
{ viewing test data.}
   Set_And_Show_Text_Window;
{ Continue to execute the function Exponentiation until the }
{ person testing this function is ready to quit. }
   repeat
   { Prompt for the base and the exponent. }
      writeln;
      write('Enter a value for X followed by a value for Y: ');
      readln(X, Y);
   { Compute X raised to power Y using the function }
   { Exponentiation. }
      Result := Exponentiation(X, Y);
   { Display the results from execution of Exponentiation. }
      writeln;
      write('Value of ', X:1, ' raised to the power ');
      writeln( Y:1,' is ', Result :1 );
   { Prompt the person testing the function Exponentiation to }
   { either stop or continue.}
      write('Enter "S" , and press RETURN key to stop testing: ');
      readln(Response);
   until (Response = 'S')
end.
```

Why not eliminate the local variable `Product` in the executable body of the function `Exponentiation` and replace the sentence `Product := Product * X` with `Exponentiation := Exponentiation * X`? Doing this would result in an error message during the compile step, because THINK Pascal treats the name of the function on the right side of an assignment operation as a direct call on itself. In this particular case, the THINK Pascal compiler responds with a dialog window having a message that too few parameters are used in a procedure or function call. For the body of any Pascal function, the function name appearing on the left side of an assignment operator is always treated as a simple variable, whereas on the right side it acts as a call upon itself.

How could the exponentiation operation be performed by a procedure instead of a function? Below is a procedure called `Exponentiation_Revised` for raising an integer *x* to a power *y*. As you can see, a third formal parameter called `Product` is necessary for returning the value of the exponentiation operation. The algorithm for computing exponentiation is basically the same.

```
procedure Exponentiation_Revised ( X, Y : integer; var Product:
```

```
                                        integer);
{ Purpose:  This procedure raises the value of X to the power Y. }
   var
      Counter : integer;

begin
{ First check for special conditions on X and Y. }
   if (Y < 0) or ((X = 0) and (Y = 0)) then
      begin
         writeln;
         write(' No result exists, since either Y < 0 or');
         writeln(' both X and Y are 0. ');
         Product := 0
      end
   else if (Y = 0) then { X must be  nonzero }
      Product  := 1
   else
   { Initialize control variable Counter and temporary variable }
   { Product, and compute X raised to the power Y. }
      begin
         Counter := 1;
         Product := 1;
         repeat
            Product := Product * X;
            Counter := Counter + 1;
         until (Counter > Y);
      end
end; { Exponentiation_Revised }
```

If we use the procedure Exponentiation_Revised in place of its corresponding function, the assignment statement in the body of the main program Testing_ Exponentiation,

```
Result := Exponentiation(X, Y);
```

must be replaced by the procedure call

```
Exponentiation_Revised (X, Y, Result);
```

Whether to use a procedure or function can be a matter of preference by the author of the algorithm. Some people are comfortable with defining subprogram units in terms of procedures, and others prefer functions. In the programming language C, all subprogram units are functions, and although C does not support a subprogram unit called a *procedure*, it is possible in C to write a function and have it take on the characteristics of a procedure.

7.3 GLOBAL VERSUS LOCAL IDENTIFIERS

In standard Pascal a program unit is represented by a declaration section followed by an executable section. The declaration section allows labels, constants, user-defined types, variables, procedures, and functions to be declared. What is important is the scope or extent of these declared identifiers in relation to other internal procedures and functions. The scope for an identifier is important because it defines the valid limits of an identifier's accessibility within the boundary of any Pascal program. Basically, the scope of an identifier extends from where the identifier is initially declared to the end of the program unit, with the exception of those nested program units that redeclare the identifier. In this situation the identifier acts as a local identifier to the program unit in which it is declared. If a nested program unit uses an identifier without redeclaring it, the identifier is considered global to the borders of the nested program unit. The rules of scope help limit the declaration of an identifier to that part of the Pascal program where the identifier is employed. They can also result in an identifier being used in several different parts of a program and having a different meaning in each part. In turn, they allow the value of an identifier to be shared among several nested program units. As an example, consider the following program called Scoping.

```
program Scoping (input, output);
   var A, B, C : real;  D, E, F : integer;
{ ------------------------------------------------------------ }
   procedure One (X, Y : integer);
      var A, E : Boolean;
   begin   { Body of outer procedure One }
   { Identifiers A and E have been redeclared and act as local }
   { identifiers to this block.  The real identifiers, B, C, and }
   { integer identifiers, D, F, in the main program are global }
   { to this block. Only this outer procedure One is known to }
   { this block, since it was declared before procedure Two.}
    end; { One }
{ ------------------------------------------------------------ }

procedure Two (var A : integer; F : Boolean);
      var B, C : integer;
   { +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ }
      procedure One;
         begin { Body of Inner procedure One }
         { Parameters A and F in procedure header Two as well }
         { as the integer identifiers B and C are global to }
         { this block. Both variables B and C are associated }
         { with the two variables declared at the beginning }
         { of procedure Two and not with the identifiers B and C }
         { declared at the beginning of the program. The real }
         { identifiers D and E declared at the beginning of the }
         { main program are global to this block. The  outer }
         { procedure Two and the inner procedure One are both }
         { known to this block. The declaration of outer }
         { procedure One is redefined by the present declaration }
         { of One. }
         end; { One }
   { +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ }
   begin { Body of procedure Two }
   { Parameters A and F as well as the integer identifiers B }
   { and C are local to this block, while the identifiers D and }
   { E declared in the main program are global to this block. }
   { The outer procedure Two and the inner procedure One nested }
   { within procedure Two are known to this body.}
   end; { Two }
{ ================== Body of the main program. ================ }
begin
{ The identifiers A, B, C, D, E, and F declared at the beginning }
{ of the program are local to this block.  Only the outer two }
{ procedures, One and Two, can be called from this block. }
end.
```

Figure 7.13 is a contour diagram showing the boundaries of program Scoping and subprograms One and Two.

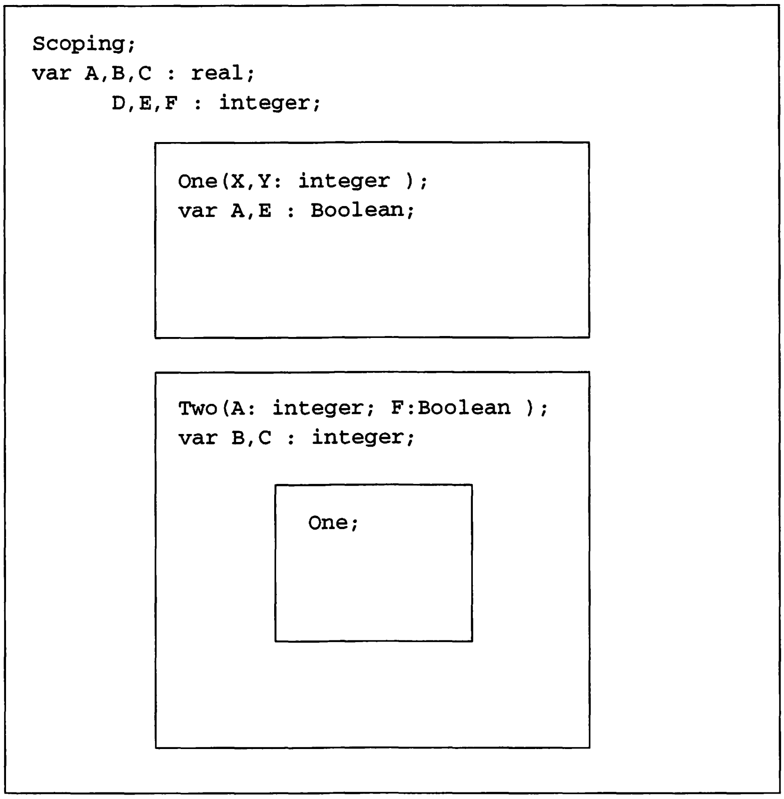


Figure 7.13 Contour diagram showing the boundaries for the main program Scoping and subprograms One and Two.

The identifiers defined in the block Scoping are accessible in blocks One and Two, and within the nested block One declared in block Two if those identifiers have not been redeclared. For example, the variables B, C, D, and F are global to block One, and variables D, E, and F are global to block Two as well as block One nested within block Two. In turn, block One nested within block Two is accessible to identifiers D, E, and F from block Scoping, and identifiers A, B, and C from block Two. For block One, nested within block Two, the parameter A and variables B and C are global within the contour of One. Note that for block Two, nested block One is the only procedure accessible by block Two, not procedure One nested in block Scoping. If procedure One is invoked from within the body of Two, it is the procedure One nested within Two that is executed, not One nested within Scoping. Identifiers D and F have scope throughout the entire block Scoping as well as blocks One and Two.

Using global identifiers can sometimes lead to bad side effects not predicted at the time of writing a program. Consider the following example, titled Bad_Habits:

```
program Bad_Habits( input, output );
{ Purpose:  This program demonstrates a basic problem in using }
{           global identifiers.}
   var
      N : integer;
      Sum : longint;
{ ------------------------------------------------------------ }
   function Total (X : integer) : integer;
   { Purpose:  This function computes the sum of }
   {           1 + 2 + 3 + . . . + X. }
   begin
      N := X;
      Sum := 0;
      repeat
         Sum :=  Sum + N;
         N := N - 1;
      until (N = 0);
      Total :=  Sum
   end; { Total }
{ ================= Body of the main program. ================= }
begin
   N := 1;
   Sum := 0;
   repeat
      Sum :=  Sum + Total(N);
      writeln(N, Sum);
      N := N + 1;
   until N  > 100;
end.
```

This program never ends execution. Although the function Total is defined to compute the sum of (1 + 2 + ... + x) for a given value of x, and the body of the main program is defined to compute the summations of 1 + (1 + 2) + (1 + 2 + 3) + ... + (1 + 2 + 3 + ... + 100), this program never seems to reach an end to its summations. Unfortunately, the person who wrote the program failed to declare N and Sum as local variables in the body of the function Total. These errors result in the function Total assigning N the value 0 each time it is executed. In the body of the main program, N becomes incremented by 1, with the value of N returning to the value 1. N never exceeds the value of 100 (it will always be 1 when the condition N > 100 is tested), so the program continues to loop forever. As for the value of Sum, it will always be 2 when the writeln statement is executed, because Sum is a global variable in the function Total. Its value is returned to 0 each time the function Total is executed.

If Pascal supports procedures defined only within the boundary of the main program unit, should we use global identifiers for procedures, rather than linking values through an actual parameter, formal parameter list? No definite answer favors global variables over a formal parameter list. One rule of thumb to follow when defining procedures or functions is to use only local variables and formal parameters within the executable body. Use formal parameters for passing a value to a formal parameter or for causing a side effect on an actual parameter. Let global identifiers be limited to constants and types, but never variables.

If you follow this rule, you will be enforcing the principle of loose coupling, or data coupling. With loose coupling, only the required amount of information is passed between the body of the main program and the subprogram unit. It is the most desirable kind of coupling since we are assured of knowing what information is being passed and what is required in return from the subprogram unit. No attempt is made to change the values of any other identifiers beyond the boundary of a subprogram unit. The opposite extreme is common coupling. Common coupling occurs when a subprogram unit makes direct references to an identifier outside its boundaries, such as a global identifier. This is the case for variables N and Sum in function Total. Once this occurs, the subprogram independence that we are attempting to achieve is lost. Applying the rule of loose coupling would keep our unknown programmer from creating the problems shown in Bad_Habits.

7.4 FORWARD DECLARATIONS

Both THINK Pascal and Macintosh Pascal are one-pass translators. This means that as a Pascal program is being translated, names of declared identifiers are added to an internal data object called a *symbol table*. If the translator encounters a reference to an identifier not in the symbol table, a translation error specifying that an identifier is undefined is raised. This often happens when a procedure is referenced before being defined. Consider the example given in the foll ·ving program called Forward_Example.

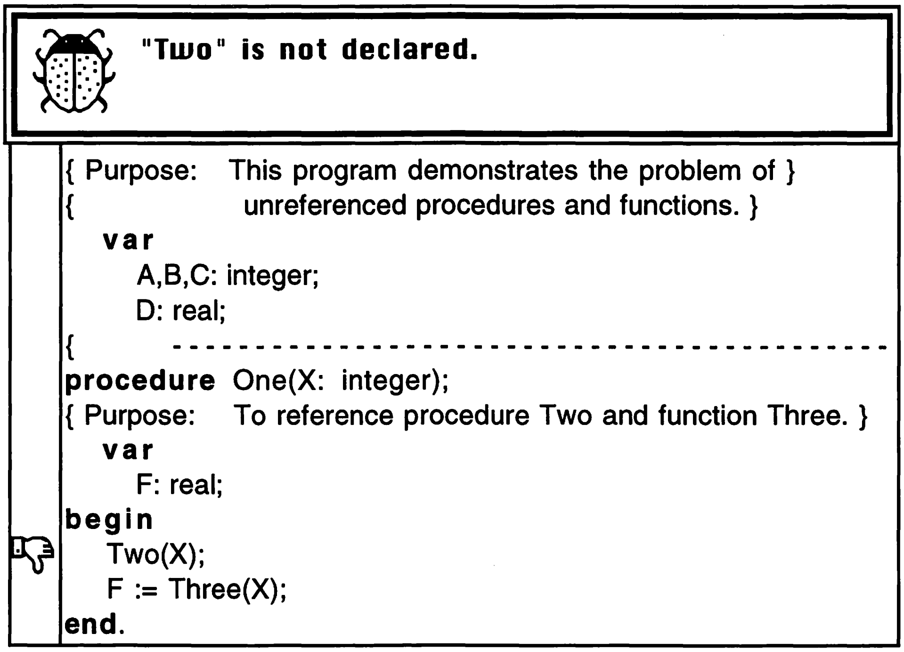
```
program Forward_Example(input, output);
{ Purpose:  This program demonstrates the problem of }
{           unreferenced procedures and functions. }
  var
    A, B, C : integer ;
    D : real ;

{ ------------------------------------------------------------ }
  procedure One(X : integer);
  { Purpose: To reference procedure Two and function Three.}
    var
      F : real;
```

```
   begin
      Two(X);
      F := Three(X);
   end;

{ ------------------------------------------------------------- }
   procedure Two(X : integer);
   { Purpose: To reference procedure One and function Three.}
      var
         F : real;
   begin
      One(X);
      F := Three(X)
   end; { Two }

{ ------------------------------------------------------------- }
   function Three(Z : integer) : real;
   { Purpose: To reference procedures One and Two.}
   begin
      One(Z);
      Two(Z);
      Three := Z
   end; { Three }

 { ================ Body for the main program. ================ }
begin
{ No executable statements.}
end.
```

In the body of procedure One, procedure Two and function Three are referenced; in the body of procedure Two, procedure One and function Three are referenced; and in the body of function Three, both procedures One and Two are referenced. Although the program appears to be correct, translating it with the THINK Pascal compiler (as well as the Macintosh Pascal translator) results in a dialog box like the one shown in Figure 7.14.

This error is generated because identifier Two (as well as Three), referenced from within the body of procedure One, appears to be an undefined identifier to the translator. Standard Pascal requires all identifiers be declared before being referenced by other declarations or executable statements. Placing procedure One before procedure Two produces this error. The definition of procedure One could be placed below Two and Three, but this would not rectify the problem, because now identifiers One and Three would appear as undefined references within the body of procedure Two.

Standard Pascal allows us to solve this problem by using a forward declaration. Sometimes referred to as a *directive*, the forward declaration allows a procedure-header or function-header to be declared, leaving the remainder of the body for later definition. This allows other procedures and functions to be declared between them, in particular those that reference the forwarded procedures and functions. The following shows the application of forward declarations to our previous program. The subprogram units have been listed in alphabetical order by name.



"Two" is not declared.

```
{ Purpose:   This program demonstrates the problem of }
{              unreferenced procedures and functions. }
   var
      A,B,C: integer;
      D: real;
{       -----------------------------------------
procedure One(X: integer);
{ Purpose:   To reference procedure Two and function Three. }
   var
      F: real;
begin
   Two(X);
   F := Three(X);
end.
```

Figure 7.14 An example of the error message resulting from an identifier that cannot be referenced.

```
program Forward_Example_Revised(input, output);
{ Purpose:  This program demonstrates the use of forward }
{           directives.}
   var
      A, B, C : integer;
      D : real;
{ ---------------- List of forward directives. -------------- }
   procedure One(X : integer);
      forward ;
   function  Three(Z : integer): real;
      forward;
   procedure Two(X : integer);
      forward;
{ --------- Definitions of procedures and functions. --------- }
   procedure One; { X : integer }
   { Purpose:  This procedure references procedure Two and }
   {           function Three.}
      var
         F : real;
   begin
      Two(X);
      F := Three(X);
   end; { One }
{ ---------------------------------------------------------- }
```

```
   function Three;    { Z : integer;  returns real }
   { Purpose: This function references procedures One and Two. }
   begin
      One(Z);
      Two(Z);
      Three := Z;
   end; { Three }
{ ------------------------------------------------------------- }
   procedure Two;    { X : integer }
   { Purpose:  This procedure references procedure One and }
   {           function Three.}
      var
         F : real;
   begin
      One(X);
      F := Three(X)
   end; { Two }
{ ================= Body of the main program. =============== }
begin
{ No executable statements.}
end.
```

With the forward directives appearing before the body of any procedure or function, program `Forward_Example_Revised` successfully compiles. With these three forward declarations, the names of identifiers `One`, `Two`, and `Three` are added to the symbol table before the translator examines the body of any procedure or function. Now when procedure body `One` is examined, the identifiers `Two` and `Three` are accessible and can be referenced. The body of the main program makes no calls upon any of the subprograms, so the program successfully executes. Understand that when a forward declaration is given, only the reserved word **procedure** or **function** followed by the identifier name is defined prior to the definition of the body. Repeating a complete procedure or function heading can result in a translation error. One method for remembering the formal parameters is to insert a copy of the parameter list as a comment before defining the remainder of the procedure or function.

7.5 PROCEDURAL AND FUNCTIONAL PARAMETERS

In Pascal the name of a procedure or function can be passed as an actual parameter to the name of a corresponding formal parameter. This implies that a formal parameter in a subprogram can represent the name of the actual procedure or function during execution of the subprogram itself. This allows a subprogram to serve as a host for several different operations. A particular operation being performed when the subprogram is in execution and the name of a procedure or function that is passed to a formal parameter is called upon by invoking the formal parameter within the body of the subprogram.

This requires that the actual and formal parameter lists be compatible. This means that the corresponding formal parameter (the formal parameter receiving the name of a procedure or function) in the formal parameter list must have a declaration that is identical in form with the procedure or function heading of its corresponding actual parameter. As an example, consider the following sample program, where three new procedures are defined: `Sine`, `Cosine`, and `Tangent`:

```
program Passing_Names (input, output);
{ Purpose:  This program serves to test the passing of the names }
{           of procedures as values to formal parameters. }
   var
      Answer, Radian_Angle: real;
      Degree_Angle: integer;

{ ------------------------------------------------------------ }
   procedure Sine (X: real; var S: real);
   { Purpose:  This procedure computes the sine of an angle, }
   {           using the standard Pascal function sin. }
   begin
      S := sin(X)
   end; { Sine }

{ ------------------------------------------------------------ }
   procedure Cosine (X: real; var C: real);
   { Purpose:  This procedure computes the cosine of an angle, }
   {           using the standard Pascal function cos. }
   begin
      C := cos(X)
   end; { Cosine }

{ ------------------------------------------------------------ }
   procedure Tangent (A: real; procedure F (Y: real; var V:
                           real);
      procedure G (Y: real; var V: real);
                     var T: real);
      { Purpose:  This procedure computes the tangent of an }
      {           angle A, using the newly defined procedures }
      {           represented by the formal parameters }
      {           F and G. }
         var
            S, C: real;
   begin
   { Compute the sine of angle A by calling on procedure Sine, }
   { using the formal parameter F. Variable S receives the sine }
   { of A. }
      F(A, S);
   { Compute the cosine of angle A by calling on procedure }
   { Cosine, using the formal parameter G. Variable C receives }
   { the cosine of A. }
      G(A, C);
   { Compute the tangent of angle A, using values of S and C. }
      T := S / C
   end; { Tangent }

{ ------------------------------------------------------------ }
   procedure Set_And_Show_Text_Window;
   { Purpose:  Sets the boundary of and opens the Text window }
   {           for viewing. }
      var
         Border: Rect;
   begin
   { Hide all windows. }
      HideAll;
   { Establish the boundaries for displaying the Text window. }
      SetRect(Border, 0, 20, 500, 300);
      SetTextRect(Border);
   { Show the Text window for viewing output.}
      ShowText;
   end; { Set_and_Show_Text_Window }

{ ================== Body of the main program. ================ }
begin
{ After hiding windows, establish the boundary of and show the }
{ Text window for viewing output. }
   Set_And_Show_Text_Window;
{ Display a header for angle in degrees and the tangent of }
{ the angle. }
   writeln('  Angle (Degrees)    ||    Tangent(Angle)');
   writeln(' ----------------------------------------');
{ Compute the tangent for angles from 0 degrees to 45 degrees. }
   for Degree_Angle := 0 to 45 do
      begin
         Radian_Angle := Pi * Degree_Angle / 180.0;
         Tangent(Radian_Angle, Sine, Cosine, Answer);
         writeln(Degree_Angle : 10, Answer : 35 : 7)
      end
end.
```

The purpose of this program is to compute the tangent of an angle the hard way. Notice that each of the procedures Sine and Cosine have one value parameter and one variable parameter. The procedure Tangent is different, because its first formal parameter on the left receives a value from the actual parameter Angle, and the second and third formal parameters are declared as procedures. They are also value-type parameters because they receive the name Sine for F and Cosine for G. Full procedure-header declarations are required for the two formal parameters F and G, because they must be identical with the procedure-headers for the procedure names being passed to Tangent, that is, Sine and Cosine, respectively. If not, THINK Pascal reports that a type incompatibility exists between an actual and formal value parameter; Macintosh Pascal reports that an erroneous Procedural or Functional parameter has been encountered where Tangent is invoked.

For procedure Tangent, execution of the statement F(A,S) causes the procedure to be interrupted, with execution now being controlled by the name of the procedure passed to F, which is Sine. When procedure Sine has completed its execution, execution of procedure Tangent continues, with procedure Cosine being called through execution of the statement G(A,C). When the procedure Cosine has completed its execution, execution of procedure Tangent continues with the assignment statement T := S / C computing the value of Tangent. Passing_Names_Revised shows the same basic example, except that functions replace procedures.

```
program Passing_Names_Revised (input, output);
{ Purpose:  This program serves to test the passing of the names }
{           of functions as values to formal parameters. }
   var
      Answer, Radian_Angle: real;
      Degree_Angle: integer;
{ -------------------------------------------------------------- }
   function Sine (X: real): real;
   { Purpose:  This function computes the sine of an angle, }
   {           using the standard Pascal function sin. }
   begin
      Sine := sin(X)
   end; { Sine }
{ -------------------------------------------------------------- }
   function Cosine (X: real): real;
   { Purpose:  This function computes the cosine of an angle, }
   {           using the standard Pascal function cos. }
   begin
      Cosine := cos(X)
   end; { Cosine }
{ -------------------------------------------------------------- }
   function Tangent (A: real; function F (Y: real): real;
                      function G (Y: real): real): real;
   { Purpose:  This function computes the tangent of an angle }
   {           A, using the newly defined functions represented }
   {           by the formal parameters F and G. }
   begin
      Tangent := F(A) / G(A)
   end; { Tangent }
{ -------------------------------------------------------------- }
   procedure Set_And_Show_Text_Window;
   { Purpose:  Sets the boundary of and opens the Text window }
   {           for viewing. }
      var
         Border: Rect;
   begin
   { Hide all windows. }
      HideAll;
   { Establish the boundaries for displaying the Text window. }
      SetRect(Border, 0, 20, 500, 300);
      SetTextRect(Border);
   { Show the Text window for viewing output.}
      ShowText;
   end; { Set_And_Show_Text_Window }
{ ==================== Body of the main program. ============= }
begin
{ After hiding windows, establish the boundary of and show the }
{ Text window for viewing output. }
   Set_And_Show_Text_Window;
{ Display a header for angle in degrees and tangent of the angle }
```

```
   writeln('  Angle (Degrees)    ||    Tangent(Angle)');
   writeln(' ---------------------------------------');
{ Compute the tangent for angles from 0 degrees to 45 degrees. }
   for Degree_Angle := 0 to 45 do
      begin
         Radian_Angle := Pi * Degree_Angle / 180.0;
         Answer := Tangent(Radian_Angle, Sine, Cosine);
         writeln(Degree_Angle : 10, Answer : 35 : 7)
      end;
end.
```

Rules for formal parameters taking on the names of functions are similar to those for procedures. As a second example, consider writing a Pascal function for computing the sum of discrete values for a general function specified as $f(k)$. The summation is represented as follows:

$$\sum_{k = \text{Lower\_Limit}}^{\text{Upper\_Limit}} f(k)$$

In Pascal this summation is given by the following function:

```
function Summation ( function f( K: Datatype ) : real;
                     Lower_Limit, Upper_Limit : Datatype ): real;
{ Purpose:  This function computes the summation of f(K) for }
{           values of K starting at a lower limit and ending at }
{           an upper limit. }
   var
      K :  Datatype;
      Total_Sum : real;
begin
{ Initialize  Total_Sum to zero. }
   Total_Sum := 0.0;
{ Compute the summation of f(i) from Lower_Limit to Upper_Limit. }
   K = Lower_Limit;
   while K <= Upper_Limit do
      begin
         Total_Sum := Total_Sum + f(K);
         K = K + 1;
      end;
   Summation := Total_Sum
end;
```

We can now apply the function `Summation` to compute the following:

$$\sum_{k=1}^{N}(2k-3)^2$$

where $f(k) = (2k - 3)^2$. The Pascal definition for function $f(k)$ follows:

```
function  F_of_K( K : Datatype )  : real;
{ Purpose:  This function represents an exact definition for }
{           f(k). }
begin
   F_of_K := sqr( 2 * K - 3 );
end;
```

Here is a program for computing the summation of $f(k)$ for values of k starting at a lower limit and ending at an upper limit.

```
program Testing_Function_F(input, output);
{ Purpose:  This example illustrates the concept of passing the }
{           name of a function as a value to a formal parameter. }
   type
      Datatype = integer;
   var
      Lower_Bound, Upper_Bound: Datatype;
{ ------------------------------------------------------------- }
   function Summation(function f(K: Datatype): real;
                      Lower_Limit, Upper_Limit: Datatype): real;
   { Purpose:  This function computes the summation of f(K) for }
   {           values of K starting at a lower limit and ending }
   {           at an upper limit. }
      var
         K: Datatype;
         Total_Sum: real;
   begin
   { Initialize  Total_Sum before computing the value for }
   { Summation.}
      Total_Sum := 0.0;
   { Compute the summation of f(k) from Lower_Limit to }
   { Upper_Limit. }
      K := Lower_Limit;
      while K <= Upper_Limit do
            begin
               Total_Sum := Total_Sum + f(K);
               K := K + 1;
            end;
      Summation := Total_Sum
   end; { Summation }
{ ------------------------------------------------------------- }
   function F_of_K(K: Datatype): real;
   { Purpose: This represents the function f(K).}
   begin
      F_of_K := sqr(2 * K - 3);
   end; {  F_of_K }
{ ===================== Body of the main program. ============== }
begin { Body of the main program. }
   ShowText;
   write(' Enter lower bound: ');
   readln(Lower_Bound);
   write(' Enter upper bound: ');
   readln(Upper_Bound);
   writeln(Summation(F_of_K, Lower_Bound, Upper_Bound) : 10 : 1);
end.
```

When the `writeln` statement in the main program is executed, the function `Summation` is called, with the formal parameter `F` receiving a copy of the function name `F_of_K`, the formal parameter `Lower_Limit` receiving a copy of the value `Lower_Bound`, and the formal parameter `Upper_Limit` receiving a copy of the value `Upper_Bound`. While executing the function `Summation`, the statement

```
Total_Sum := Total_Sum + f(K);
```

calls the function `F_of_K`, with the actual parameter having a value that is presently assigned to the local variable `K`. The parameter `K` listed in the function-header of `Summation` is simply a dummy argument for the purpose of proper syntax and could therefore use any identifier name other than `K`.

7.6 RECURSIVE FUNCTIONS AND PROCEDURES

Recursion is an important aspect of both mathematics and computer science. The *American Heritage Dictionary of the English Language* defines the term *recursion* as "pertaining to, or designating (a) a mathematical expression, each term of which is determined by application of a formula to preceding terms, (b) a formula that generates the successive terms of such an expression." In brief, a recursive definition is one that is circular by being defined in terms of itself.

For example, consider the factorial of a positive integer number n. In general, it is usually defined as

```
n!  = 1 * 2 * 3 * 4 * 5  . . .  * ( n -1 ) * n   for  n > 0
```

where 0! is understood to be 1. If we write separately each of the factorials for the integer numbers 0 through n , we see a pattern emerge that can define the factorial recursively:

```
0! =  1
1! =  1 * 1 = 1 * 0!
2! =  2 * 1 = 2 * 1!
3! =  3 * 2 * 1 = 3 * ( 2 * 1) = 3 * 2!
4! =  4 * 3 * 2 * 1 = 4 * ( 3 * 2 * 1 ) = 4 * 3!
5! =  5 * 4 * 3 * 2 * 1 = 5 * ( 4 * 3 * 2 * 1 ) = 5 * 4!
   . . .
```

```
n! =  n  * [ ( n - 1 ) * ( n - 2 ) * ( n - 3 ) * . . . *  2 * 1 ]
   =  n  * ( n - 1 )!
```

In short, we can express the recursive definition for the factorial of a positive integer in two lines:

```
(a) 0! = 1
(b) n! = n * ( n - 1 )!  for  n  > 0.
```

It is recursive because the factorial of n has been defined in terms of the factorial of $n - 1$, that is, in terms of itself. As an example of applying this definition, consider the following steps for showing the levels of recursion in computing the value of 5! :

```
5!  = 5 * 4!       ( level  1 )
4!  = 4 * 3!       ( level  2 )
3!  = 3 * 2!       ( level  3 )
2!  = 2 * 1!       ( level  4 )
1!  = 1 * 0!       ( level  5 )
0!  = 1            ( level  6 )
```

Although the step of computing 0! has halted the recursion, it has not provided an answer to the value of 5!. To determine this value, we must now unravel the levels of recursion by computing each of the recursive steps starting with 0!. We obtain the following:

```
0!  = 1                           ( level  6 )
1!  = 1 * 0! = 1                  ( level  5 )
2!  = 2 * 1! = 2 *  1  =   2      ( level  4 )
3!  = 3 * 2! = 3 *  2  =   6      ( level  3 )
4!  = 4 * 3! = 4 *  6  =  24      ( level  2 )
5!  = 5 * 4! = 5 * 24  = 120      ( level  1 )
```

Although we have not discussed the programming aspects of recursion, we can see some important points in the concept of recursion. First, it allows us to divide the computation of a function into smaller steps, so that we can build on the computation with simple answers. Second, we need one or more trivial cases to end the recursion. It is important to reach a point at which the recursion stops, and we can begin the process of unraveling the recursion. With factorial, computing 0! provides the necessary step for ending the recursion. Third, we need to unravel the recursion to arrive at a final answer. This unraveling is a backward trace through the recursive steps, substituting the computed results from the levels of recursion that have followed. Eventually our backward tracing reaches the initial level of recursion, ending our computation. Fourth, each level of our backward tracing requires saving the result from a computation of a recursion that has followed. In the factorial example, we have written these intermediate values on paper. When programming, the computer software and hardware perform this action for us. Fifth, writing recursive functions requires a different way of thinking, one that forces us to forget defining functions by iteration and to think recursively.

As a second example, consider computing the sum in the following equation:

```
Sum(n)  = 1 + 2 + 3 + 4 + 5 +  . . .  + ( n - 1 ) + n
```

We can derive a recursive definition by writing the following steps:

```
Sum(1) = 1
Sum(2) = 2 + 1 = 2 + Sum(1)
Sum(3) = 3 + 2 + 1 = 3 + ( 2 + 1 ) = 3 + Sum(2)
Sum(4) = 4 + 3 + 2 + 1 = 4 + ( 3 + 2 + 1 ) = 4 + Sum(3)
   . . .
Sum(n) = n + [ ( n - 1) + ( n - 2 ) + . . .  + 2 + 1 ]
       = n + Sum(n -1)
```

In short, we can express the recursive definition for Sum(*n*) in two lines:

```
(a) Sum(1) = 1
(b) Sum(n) = n  + Sum(n -1)  for  n > 1.
```

The initial steps for computing Sum(5) recursively follow:

```
Sum(5) = 5 + Sum(4)
Sum(4) = 4 + Sum(3)
Sum(3) = 3 + Sum(2)
Sum(2) = 2 + Sum(1)
Sum(1) = 1
```

In unraveling the recursion, the following steps complete the computation of Sum(5):

```
Sum(1) = 1
Sum(2) = 2 + Sum(1) = 2 +  1 =  3
Sum(3) = 3 + Sum(2) = 3 +  3 =  6
Sum(4) = 4 + Sum(3) = 4 +  6 = 10
Sum(5) = 5 + Sum(4) = 5 + 10 = 15
```

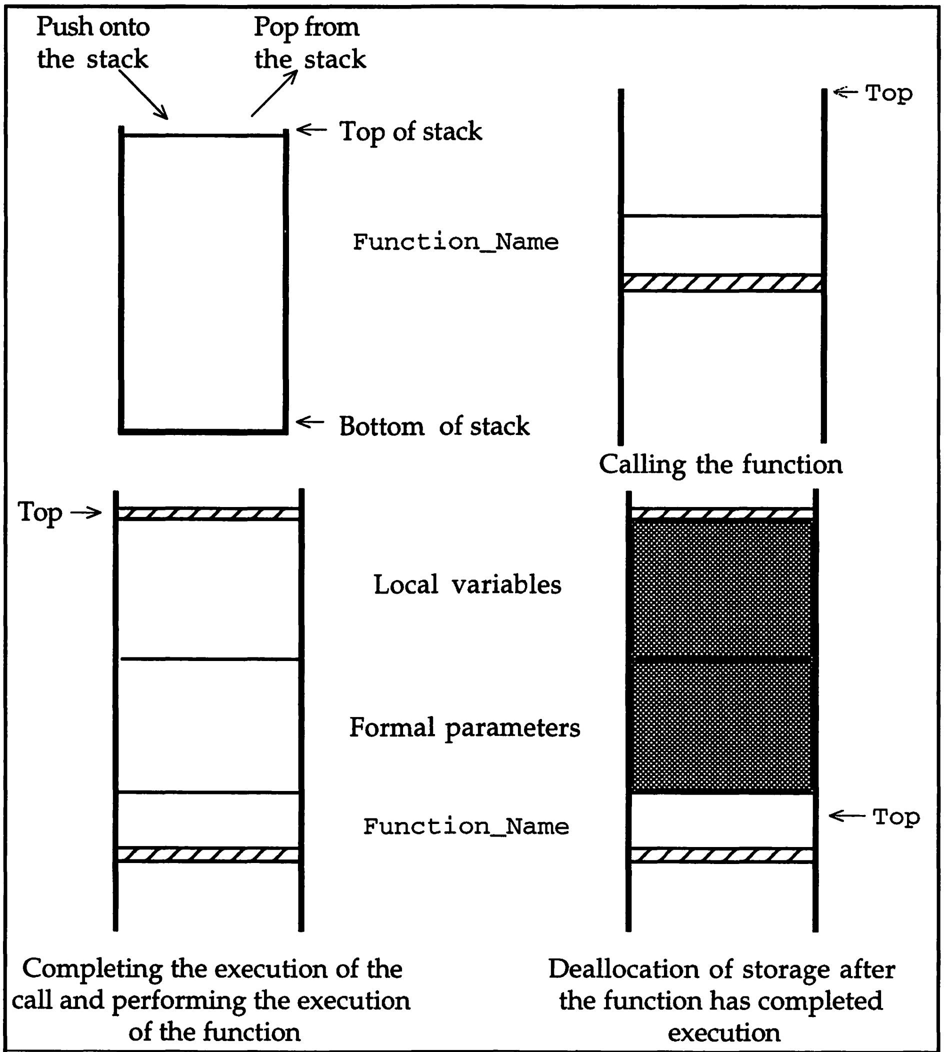
These trivial examples clearly show how easy it is to define recursive functions.

In Pascal a function or procedure is said to be *directly recursive* if the function or procedure calls directly on itself from within the body of the function or procedure. For the factorial function, the Pascal definition follows:

```
function Factorial( N : integer ) : longint;
{ Purpose:  Computes the factorial of a positive integer value. }
begin
{ Test if a trivial case exists. }
   if N = 0 then
      Factorial := 1
   else { compute (n-1)! }
      Factorial := N * Factorial( N - 1 );
end; { Factorial }
```

The formal parameter N needs only to receive a value and never causes any side effect on its corresponding actual parameter, so it is declared as a value parameter in the parameter list. In addition, the function is declared as a `longint` value, because we must accommodate very large `integer` results.

When executing a recursive function, we need temporary storage for each recursion that is performed. In both THINK and Macintosh Pascal, this is provided by a special data structure called a *system stack*. A system stack is a unique data object: information is pushed onto the stack at only one end, and at this same end information is removed by popping it off the stack. The push and pop operations are performed in a first-in, last-out manner. The first object pushed onto the top of the system stack is the last object popped off the stack. Figure 7.15 shows an example of a stack.



Note: Assume that the pointer called Top is pointing to the last object pushed onto the stack and the next object ready to be popped from the stack.

Figure 7.15 Representation of the system stack as it applies to the computation of a function.

When a function is called in Pascal, a memory-cell address is pushed onto the top of the system stack, representing the location where the value of the function will be stored. This is now followed by the allocation of cells for formal parameters, followed by local variables. When the function has completed execution, memory allocations for both local variables and formal parameters are popped, leaving at the top of the stack the location of the cell containing the value of the function. When the value of the function is accessed to complete the evaluation of an expression, the value contained in the memory cell is copied and applied to the name of the function in the expression. The memory cell associated with the function presently at the top of the stack is now popped.

As an example, let us consider the computation of 5! using the Pascal function `Factorial` defined earlier. Figure 7.16 provides a brief pictorial view of instances of execution of `Factorial` at six different levels of recursion. Two views are presented at each level: on the left we represent the body of the function at the point that the function `Factorial` is to be executed recursively, and on the right we present the contents of the stack. The first time `Factorial` is executed, a memory-cell address is pushed onto the top of the stack for storing the value of `Factorial(5)`. On top of this is pushed an address for storing the formal value parameter `N`. `N` is greater than zero, so the expression `5 * Factorial(4)` is executed. Before we can complete execution of this expression, it is called recursively by executing `Factorial` with `N` having the value 4. Notice that another address is now pushed onto the stack for storing the value of `Factorial(4)`, and an address is pushed on the stack for storing the formal value parameter `N`. `N` is again greater than zero, so these steps are repeated, with addresses being pushed on the stack for storing the intermediate values of `Factorial(3)`, `Factorial(2)`, and `Factorial(1)`.

On the last recursive execution of the function `Factorial`, the formal value parameter `N` is zero. This ends the recursion by assigning a value of 1 to the memory cell of `Factorial(0)`. The recursion is now unraveled by backtracking through the preceding levels. Figure 7.17 shows the results as execution of the expression `N * Factorial( N - 1 )` is completed. On completing execution of `Factorial(0)`, the address for the formal value parameter `N` is popped from the stack. Control now returns to the body of the function `Factorial` for computing `Factorial(1)`. At this point the value of `Factorial(0)` is copied, and then its address is popped from the stack. The value for `Factorial(0)` is now used in evaluating the expression `1 * Factorial(0)`, and the result is assigned to `Factorial(1)`. As you can see from Figure 7.17, these steps are repeated for the preceding levels of recursion: `Factorial(2)`, `Factorial(3)`, and `Factorial(4)`. After execution of the body of `Factorial` for computing `Factorial(4)`, we have reached the point from which the function `Factorial` was first called. At the top of the stack is the address containing the value of `Factorial(5)`.

As you can see, the system stack is a useful object for describing the dynamic execution of a recursive routine. By using a system stack, we can show the dynamic paths of execution between procedures and functions. This is different from reading a program listing, which is a static object indicating the semantic actions in the body of the routine. It does not adequately display the dynamic actions among several bodies.

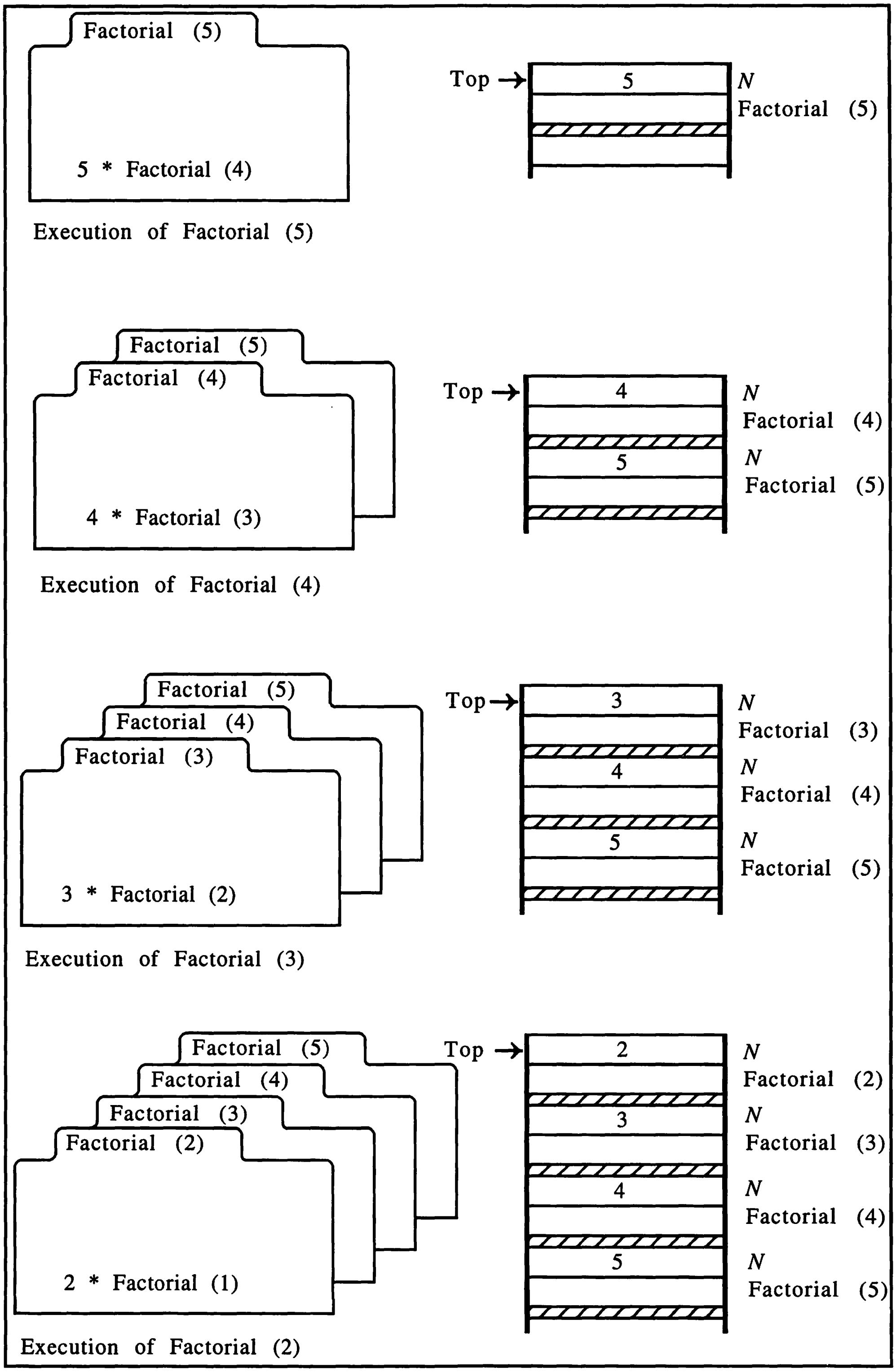


Figure 7.16 Execution of a recursive function, `Factorial`.

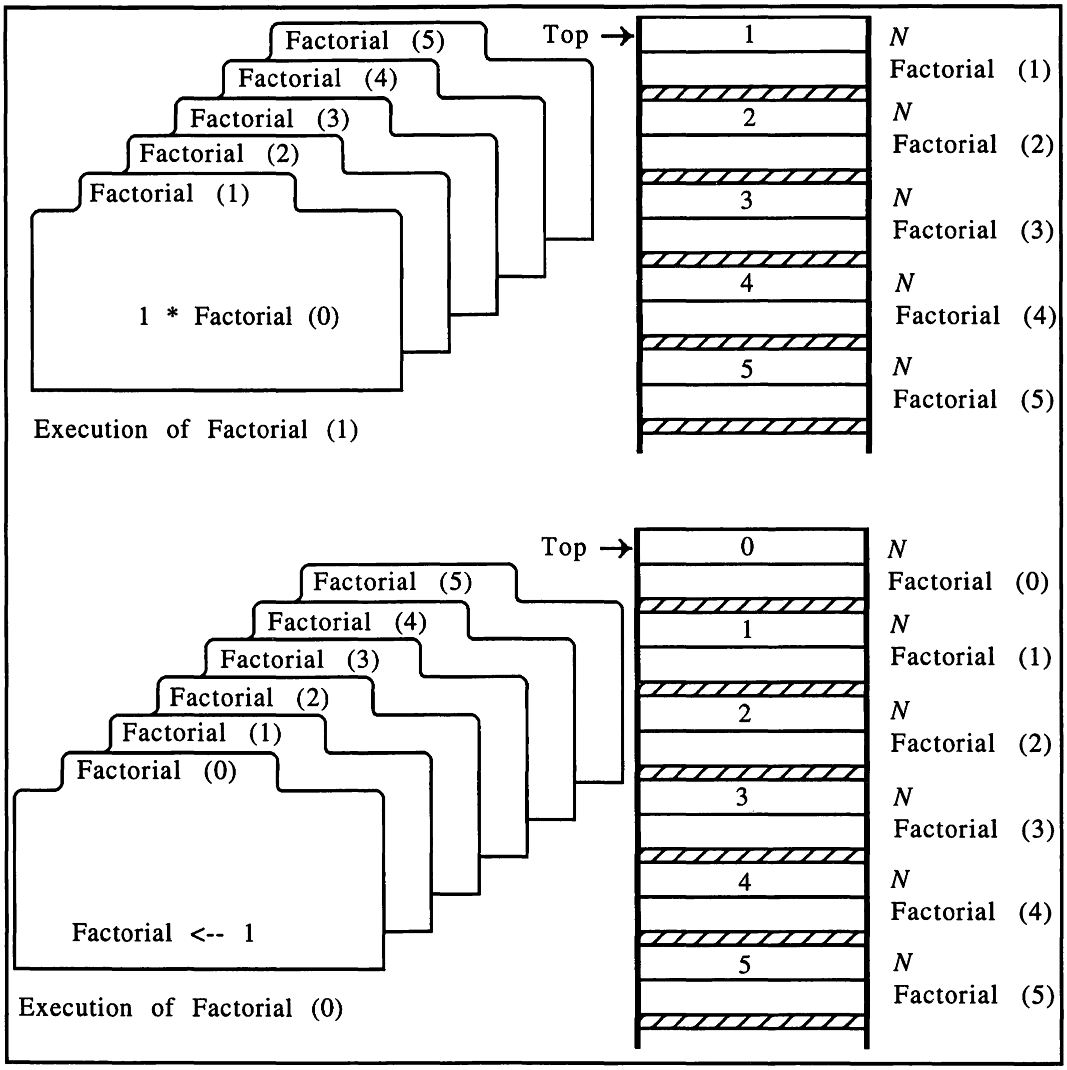


Figure 7.16 (continued)

The following is an example of a procedure that is directly recursive. As you can see, it computes the factorial of a positive integer.

```
procedure Factorial( N : integer; var Fact : longint );
{ Purpose:  This procedure computes the factorial of a positive }
{           integer value. }
begin
{ Test if the trivial case exists. }
   if N = 0 then
      Fact := 1
   else
```

```
      begin
         Factorial( N-1, Fact );
         Fact := N * Fact
      end;
end;
```

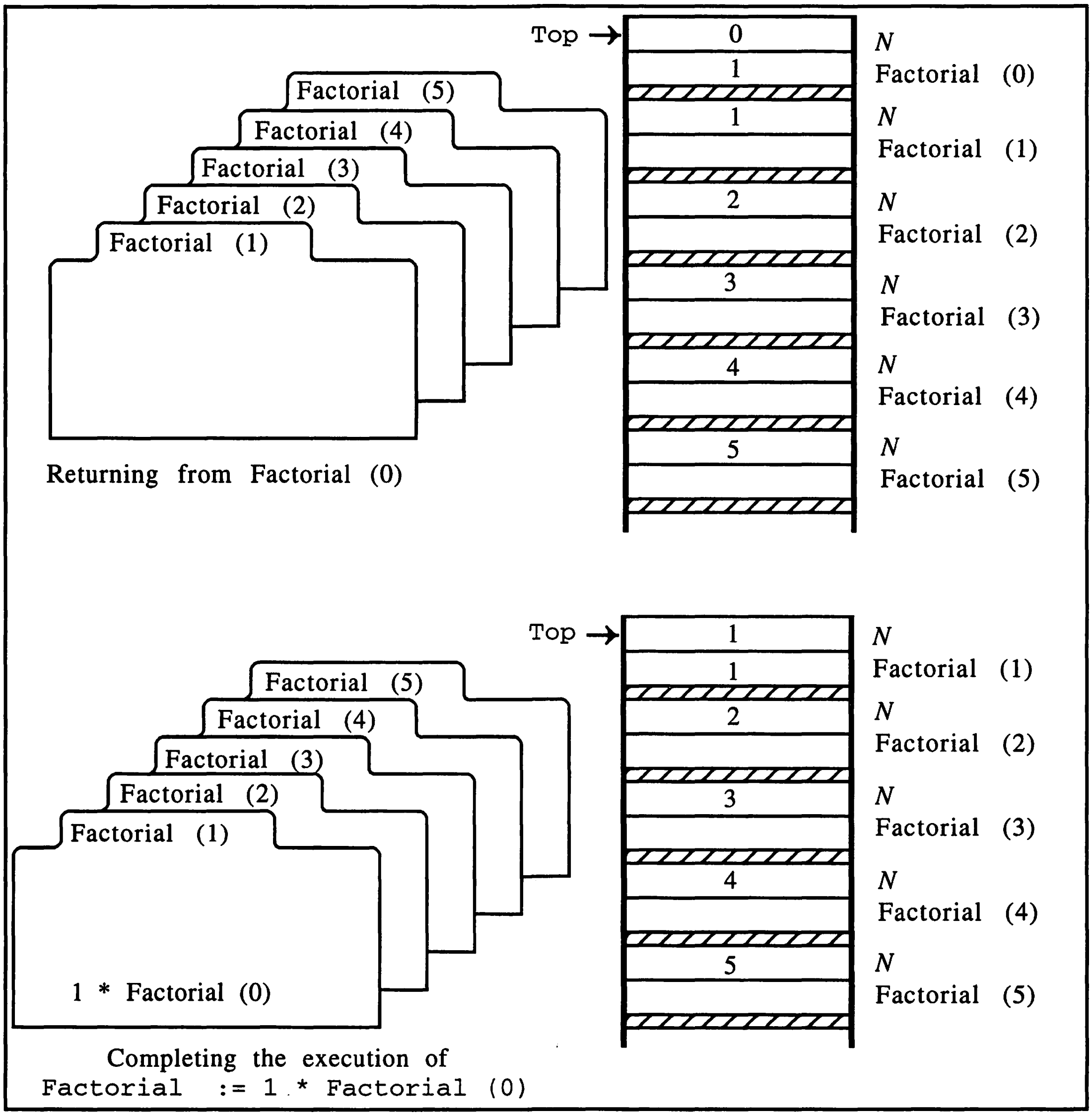
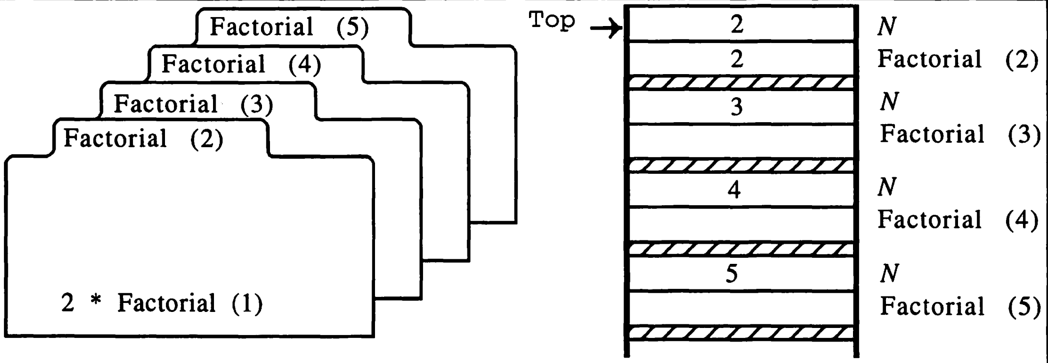
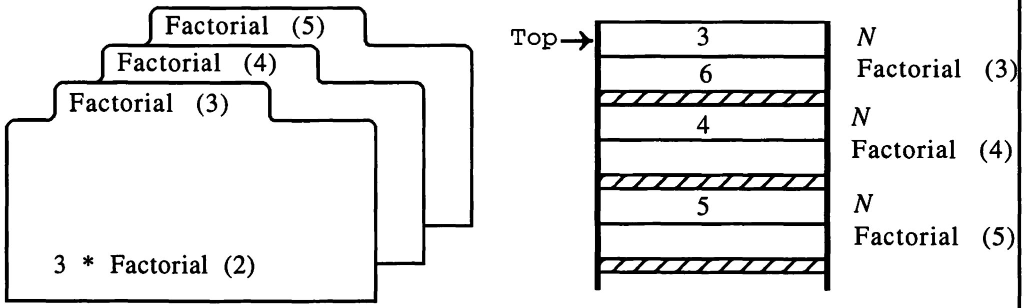


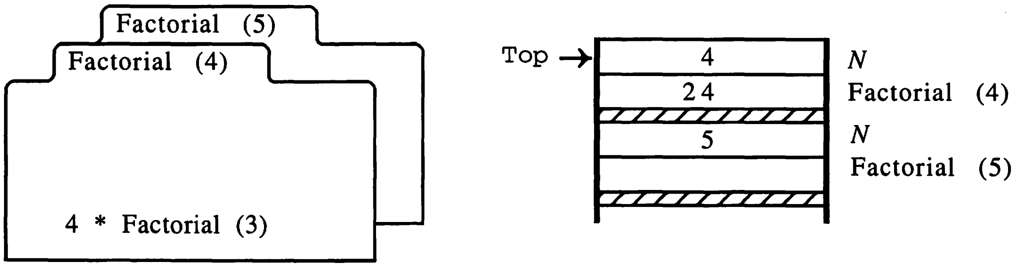
Figure 7.17 Unraveling the recursive function `Factorial`.



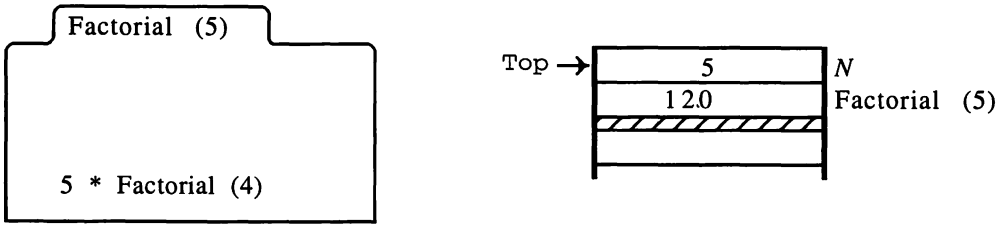
Completing the execution of
`Factorial := 2 * Factorial (1)`



Completing the execution of
`Factorial := 3 * Factorial (2)`



Completing the execution of
`Factorial := 4 * Factorial (3)`



Completing the execution of
`Factorial := 5 * Factorial (4)`

Figure 7.17 (continued)

This particular example requires two formal parameters: one for storing the value of N and a second variable parameter, Fact, for returning the computed value of Factorial. When using a stack to trace the dynamic paths of execution, the formal value parameter N will have a memory-cell address for storing the value of its corresponding actual parameter. The formal variable parameter Fact will have an address for storing the memory location of its corresponding actual parameter. In the case of our recursive procedure listed earlier, the formal parameter Fact will contain the memory location of the corresponding actual parameter when Factorial is first called. Below are some useful heuristics for defining a recursive function or procedure:

1. Always start with a trivial case or a terminal case in which the final solution is known, for example, results that terminate execution for a value of a data object becoming 1 or 0.
2. For the nontrivial case, try to define the steps needed to reduce the value nearer to the trivial case.
3. Combine the trivial case with one or more other nontrivial cases by preceding the nontrivial cases with a conditional statement employing the trivial case.
4. Check your definition with several nontrivial examples.

To understand these steps, consider a recursive function for computing the *n*th term of a Fibonacci series. A Fibonacci series is a sequence of integers, the first two terms of which are 1 and 1, respectively. Each of the remaining terms of the series is the sum of the preceding two terms. Thus the initial terms of a Fibonacci series are 1, 1, 2, 3, 5, 8, 13, 21, ... First, what are the trivial cases with respect to a Fibonacci series? If *n* is 1, then the *n* th term is 1. If *n* is 2, the *n*th term is again 1. For the nontrivial cases, we need to add the values of the $(n - 1)$th and $(n - 2)$th terms to obtain the value of the *n*th term.

Consider now a more formal definition for performing these steps:

```
Function Fibonacci;
{ This function computes the Nth term of the Fibonacci series,
  requires one formal parameter, N, and will return one value. }

begin
{ Consider the trivial cases where N is 1 or 2. }
   if ( N = 1 ) or ( N = 2 )  then
      Fibonacci <-- 1;
   else { Compute the sum of the N-1th and N-2th terms. }
      Fibonacci <-- Fibonacci( N-1 ) + Fibonacci( N-2 )
end; { Fibonacci }
```

As you can see, defining the step in the nontrivial case is almost instantaneous. For the trivial cases where N is equal to 1 or 2, the value for function Fibonacci is obvious. For N equal to 3, the following steps are executed:

1. Fibonacci is called, with N having the value 3.
2. Because N does not equal 1 or 2, the expression Fibonacci(N-1) is called recursively, with the formal parameter N having the value 2.

3. Because the function Fibonacci is invoked recursively with the formal parameter N having a value 2, the function is assigned the value 1, with execution returning to the expression Fibonacci(N-1) + Fibonacci(N-2).
4. Before addition can be performed, the function Fibonacci is called, the value 1 being passed to the formal parameter N.
5. Because the function Fibonacci is invoked recursively with the formal parameter N having a value 1, the function is assigned the value 1, with execution again returning to the expression Fibonacci(N-1) + Fibonacci(N-2).
6. The sum of Fibonacci(N-1) + Fibonacci(N-2) is computed and assigned to the name of the function.
7. Execution of the function Fibonacci terminates.

It is important to note that the parameter N must be a value type. It cannot be a variable type, because the actual parameter involves an expression. In addition, we called the function Fibonacci with N being fixed. No side effect was defined on the original value of N. Here is the test program for the function Fibonacci:

```
program Computing_Nth_Fibonacci(input, output);
{ Purpose:  This program provides a short demonstration of a }
{           recursive function. }
   var
      N: integer;

{ ------------------------------------------------------------- }
   function Fibonacci(N: integer): integer;
   { Purpose:  This function will compute the Nth term of the }
   {           Fibonacci series. It requires one formal }
   {           parameter, N, and will return one value.}
   begin
   { Consider the trivial cases where N is 1 or 2. }
      if (N = 1) or (N = 2) then
         Fibonacci := 1
      else { add the N-1th and N-2th terms }
         Fibonacci := Fibonacci(N - 1) + Fibonacci(N - 2);
   end; { Fibonacci }

{ ------------------------------------------------------------- }
   procedure Set_And_Show_Text_Window;
   { Purpose:  Sets the boundary of and opens the Text window }
   {           for viewing. }
      var
         Border: Rect;
   begin
   { Hide all windows. }
      HideAll;
   { Establish the boundaries for displaying the Text window. }
      SetRect(Border, 0, 20, 500, 300);
      SetTextRect(Border);
   { Show the Text window for viewing output.}
```

```
      ShowText;
   end; { Set_And_Show_Text_Window }

{ ================ Body of the main program ================= }
begin
{ Hide all windows and then establish the Text window for }
{ viewing output. }
   Set_And_Show_Text_Window;
{ Establish a header for viewing values of N and Fibonacci }
{ numbers. }
   writeln(' Value of N       ||      Fibonacci(N)');
   writeln(' ----------------------------------');
   for N := 1 to 21 do
      writeln(N : 9, Fibonacci(N) : 30);
end.
```

Unfortunately, the execution time of `Fibonacci(N)` is slow when `N` becomes large because of the way the function has been defined. First, the function makes no attempt to save the last two terms of the series. Rather it requires the complete series to be recomputed for each value of the formal parameter `N` as the function executes recursively. Second, the expression `Fibonacci(N-1) + Fibonacci(N-2)` first requires $n-1$ terms of the series to be recomputed, followed by $n-2$ terms. This is not to say that all recursive definitions require excessive execution time. A recursive definition may be more elegant in terms of understanding a solution, but it may not guarantee fast execution time. Further analysis may be required to reduce execution time.

A second form of recursion that Pascal functions and procedures can take is called *indirect recursion.* A procedure or function is said to be *indirectly recursive* if it calls on some other procedure or function, which in turn calls directly or indirectly on the original procedure or function. Indirect recursion requires one or more forward declarations with one or more procedure-headers or function-headers. This is so because within the body of an indirectly recursive routine, a call is made to a routine that has not yet been declared, and at this point, as the translator is checking, the syntax and names of identifiers are still unknown. The name of this unknown routine is made available through the use of the forward attribute.

For example, consider the program titled `Indirect_Recursion`.

```
program Indirect_Recursion(input, output);
{ Purpose:  This program provides a short example of indirect }
{           recursion. }
   var
      Answer : longint;
      Number : integer;

   procedure Compute_Expression (N : integer;  var Fact :
                                     longint);     forward;

{ ----------------------------------------------------------- }
   function Factorial(N : integer) : longint;
   { Purpose: Computes the factorial of a positive integer N. }
      var
```

```
        Fact : longint;
   begin
   { Test if a trivial case exists. }
      if N = 0 then
         Factorial := 1
      else
         begin { compute (N-1)! }
            Compute_Expression(N, Fact);
            Factorial := Fact;
         end
   end; { Factorial }

{ ------------------------------------------------------------------ }
   procedure Compute_Expression; { N : integer;  var Fact :  }
                                   { longint }
   { Purpose: This procedure computes the expression N * (N-1)!. }
   begin
      Fact := N * Factorial(N - 1);
   end; { Compute_Expression }

{ =================== Body of the main program. =============== }
begin { Body of the main program. }
   ShowText;
{ Prompt user for a nonnegative integer number. }
   write('Enter a positive whole number: ');
   readln(Number);
   if Number > 0 then
      begin
         Answer := Factorial(Number);
         writeln(Answer : 11)
      end;
end.
```

This is not the best approach to defining `Factorial` recursively, but it does provide a simple example of indirect recursion. In the body of the function `Factorial`, a call is made to the procedure `Compute_Expression`. This procedure has only one executable statement in its body, which computes the expression `N * Factorial(N-1)`. The value for this expression is passed back through the formal variable parameter `Fact`. This procedure provides indirect recursion by calling on the function `Factorial`. As you can see, a forward declaration is needed for `Compute_Expression`, because its body is declared after the function `Factorial`.

7.7 DEVELOPING MODULAR PROGRAMS THROUGH STEPWISE REFINEMENT

The purpose of procedures and functions is to allow the development of modular programs, in which parts of a larger program are broken into separate units called *modules*. These modules are developed, assembled, and tested as separate units and then integrated into the framework of a larger program. In relation to developing an algorithm, calling on a module is equivalent to stating a major step in the solution of a problem without having to express the detailed steps. The definition of the module provides the detail for a major step, because it becomes the vehicle for specifying the supporting substeps. Our approach to problem solving has been based on the concept of *stepwise refinement*. Basically, this represents a problem-solving abstraction that can contain several levels. At the top level of the abstraction we are concerned with defining the specifications and requirements of a problem. This approach allows us to practice the principle of top-down design.

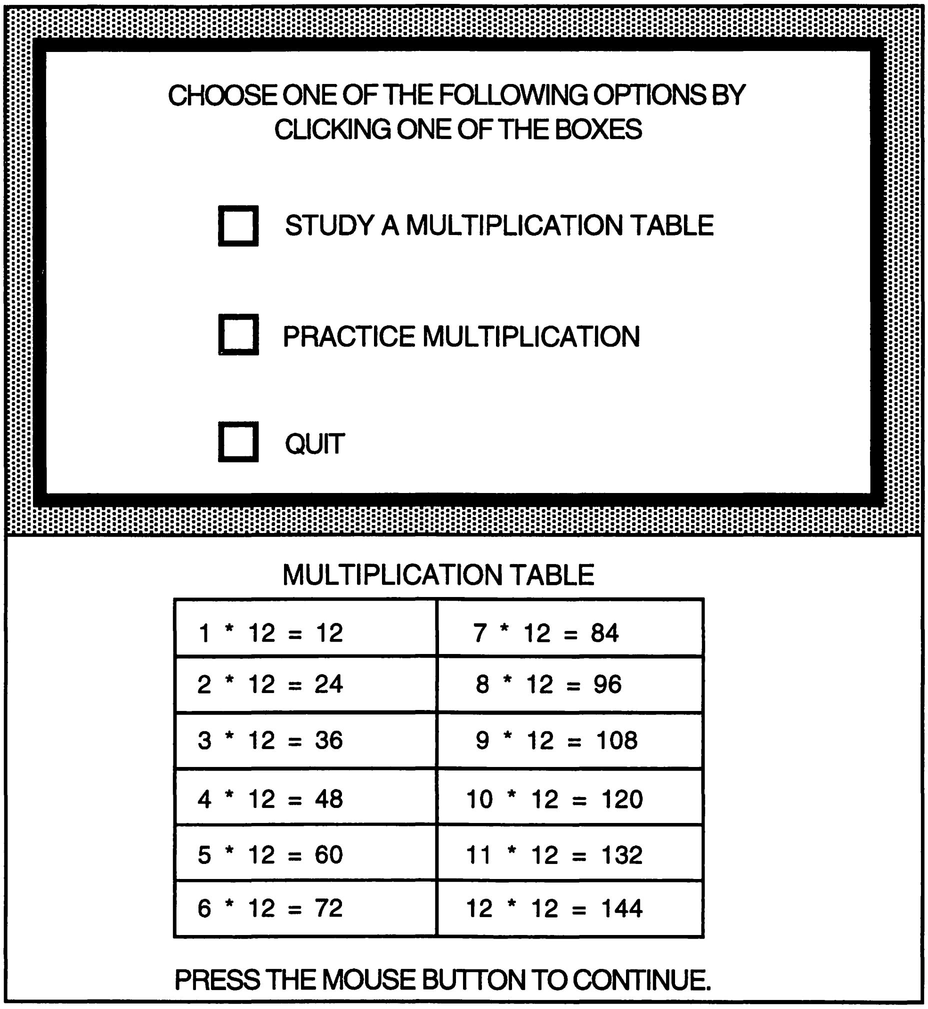
With top-down design, we develop an application from general to specific steps. It is a hierarchical approach in which requirements are first specified, then followed by the definition of major functions, the development of modules supporting the design, and finally coding and testing. The alternative to top-down design is bottom-up design, which begins with the lowest level of software components of development and proceeds through progressively higher levels to the top level of the design. In bottom-up design we often press ourselves to develop the support modules before the top level of the design is completed. In real programing environments, a combination of top-down and bottom-up design is often practiced. As an example of top-down design, consider the following example.

7.7.1 Abstraction 1

The program to be developed is a simple tutorial system for teaching multiplication tables. The student is given a menu having the options (1) for reviewing a multiplication table, (2) practicing with a set of multipliers, or (3) terminating the application. Once the option for either reviewing or practicing is selected, the student is asked to provide a value for the multiplicand, which must be in the range 1 to 100. At the end of a practice session, the student is given a short report on the number of questions answered correctly and a performance rating. The program must select multipliers in the range 1 through 12. Input is either from the keyboard or by clicking the mouse button. Output is only the text drawn in the Drawing window rather than the Text window. Figure 7.18 shows two of five window formats used by this application: at the top is an example of the Drawing window for selecting an option, and at the bottom is the format displaying a table of multipliers.

At the first level of abstraction we define a solution in terms of the problem environment but without any details of the actual design. At this point we understand who interacts with the application, what is expected for input, and what is expected for output. We understand that only Drawing windows display information and that the student enters data either by pressing keys at the keyboard or by clicking the mouse button. If necessary we might test these requirements by thinking of one or more scenarios for interacting with the application, even though it has not been programmed.

The second level of the abstraction moves away from the problem environment and employs a natural language to describe the major processes associated with the application.



CHOOSE ONE OF THE FOLLOWING OPTIONS BY CLICKING ONE OF THE BOXES

☐ STUDY A MULTIPLICATION TABLE

☐ PRACTICE MULTIPLICATION

☐ QUIT

MULTIPLICATION TABLE

1 * 12 = 12	7 * 12 = 84
2 * 12 = 24	8 * 12 = 96
3 * 12 = 36	9 * 12 = 108
4 * 12 = 48	10 * 12 = 120
5 * 12 = 60	11 * 12 = 132
6 * 12 = 72	12 * 12 = 144

PRESS THE MOUSE BUTTON TO CONTINUE.

Figure 7.18 Screen formats for Tutor_System.

7.7.2 Abstraction 2

Here is a general algorithm for Tutor_System:

```
Algorithm Tutor_System;
begin
   Initialize the Drawing window for displaying a choice;
   repeat the following steps until the student is ready to quit:
   Select a choice from a menu having three options;
      case of Choice
         Tutor       :  Study a multiplication table;
         Practice :  Practice with multipliers;
         Quit      :  Choose the option to stop;
```

```
end. { Tutor_System }
```

As you have seen, structure charts are simple block diagrams showing the hierarchical relationship between blocks (procedures and functions) in an application. The term *module*, commonly applied in the field of software engineering, refers to a block. Figure 7.19 shows an initial structure chart for `Tutor_System` in which complete control is represented by the main module at the top of the diagram.

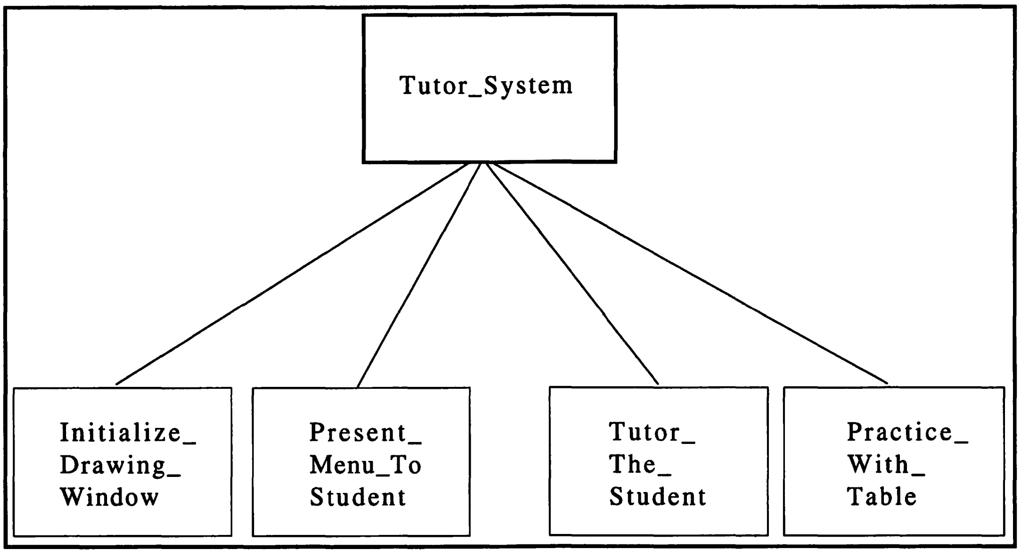


Figure 7.19 The hierarchial organization of `Tutor_System`.

Each module at the second level of the structure chart represents a major step in the solution to the problem. The topmost module acts as a supervisor (superordinate module) for the modules below it. The purpose of the main module is to make major decisions in solving a problem by performing the major steps while delegating the details of execution to subordinate modules. You can compare the main module to a supervisor in a company who delegates authority and procedures to employees working under his or her control. In relation to `Tutor_System`, the top module represents the main program, and modules at lower levels of the structure chart represent procedures and/or functions. Even at this level, testing can be conducted, but only in a broad sense. For example, assuming a value is picked for `Choice` by executing the menu, we can trace the algorithm to its next executable step (assuming that all major steps are executed).

At the next level of abstraction, we employ more structured terminology and procedural representation to describe the steps of the solution in greater detail.

7.7.3 Abstraction 3

Now that we have defined our initial algorithm and traced it to see that it is correct, we are ready to define the detailed algorithm for `Tutor_System`. The following shows the main module with comments and the names of modules representing major steps in our algorithm.

```
Algorithm Tutor_System;
{ Purpose:  Provide the student with the options of studying or
            practicing with a multiplication table. This module
            has only two variables: Choice and Screen. }
begin
{ Initialize the Drawing window for displaying a choice. }
   Initialize_Drawing_Window ( Screen );
{ Continue with the following steps until the student is ready
  to quit. }
   repeat
   { Prompt the student for a choice. }
      Present_Menu_To_Student(Screen, Choice);
   { Choose the option to Tutor, Practice, or quit.}
      case Choice of
         Tutor    :  Tutor_The_Student (Screen);
         Practice :  Practice_With_Table (Screen);
         Quit     :  HideAll;
      end;
   until Choice = Quit ;
end. { Tutor_System }
```

The modules `Present_Menu_To_Student`, `Tutor_The_Student`, and `Practice_With_Table` require the Drawing window for displaying information, so we use the variable `Screen` for passing the boundary information of the Drawing window between modules. This is necessary because the background of the drawing window will alternate between light-gray and white. Also note that while the comments added to the algorithm appear to be redundant, they are important for understanding the steps of the algorithm if the major controlling constructs are removed.

We can apply the principle of prototyping by writing Pascal code for the main program along with dummy procedures, referred to as *stubs*. As our understanding of the problem progresses, we will add more Pascal code to test our concepts and if necessary redefine the requirements and alter the design. We begin by testing the main module of `Tutor_System` to see if it is functional. The following provides an example of Pascal code for testing these stubs.

```
program Tutor_System(input, output);
{ Purpose:  Provide the student with the options of studying or }
{           practicing with a multiplication table. }
   type
      Option = ( Tutor, Practice, Quit);
   var
      Choice : Option;
      Screen : Rect;
{ ------------------ Forward Directives ----------------------- }
   procedure Initialize_Drawing_Window( var Drawing_Box: Rect );
      forward;
   procedure Practice_With_Table( Drawing_Box : Rect );
      forward;
   procedure Present_Menu_To_Student( Drawing_Box: Rect;
                                       var Choice: Option );
      forward;
   procedure Tutor_The_Student( Drawing_Box: Rect );
      forward;
{ --------------------- Module Definitions ------------------- }
   procedure Initialize_Drawing_Window; { var Drawing_Box:Rect }
{ Purpose:      This procedure sets the boundaries for both the }
{               Drawing window and the Text window. The Drawing }
{               window will be located directly below the title }
{               bar; the remainder of the screen shows only the }
{               Drawing window. }
   begin
      Writeln(' Procedure Initialize_Drawing_Table has been
               executed.');
   end; {Initialize_Drawing_Window}
{ ------------------------------------------------------------ }
   procedure Practice_With_Table; { Drawing_Box : Rect }
{ Purpose:      This procedure lets the student practice with }
{               his or her own multiplication table. }
   begin
      writeln(' Procedure Practice_With_Table has been
                executed.');
   end; {Practice_With_Table}
{ ------------------------------------------------------------ }
   procedure Present_Menu_To_Student; { Drawing_Box: Rect;
                           var Choice: Option }
      begin
{ Purpose:      This procedure returns a choice for an option.}
         writeln(' Procedure Present_Menu_To_Student has been
                  executed.');
         Choice := Quit;
      end;{ Present_Menu_To_Student }
{ ------------------------------------------------------------ }
   procedure Tutor_The_Student; { Drawing_Box: Rect }
{ Purpose:      This procedure presents a multiplication table }
{               for the student to review. }
      begin
         Writeln(' Proced ~e Tutor_The_Student has been
                  executed.');
      end; {Tutor_The_Student}
{ ================= Body of the main program. ================= }
begin
{ Establish the Drawing window for prompts and responses. }
   Initialize_Drawing_Window( Screen );
{ Use writeln statements in procedure stubs to display messages }
{ that the the procedures have been executed. }
   repeat
      Present_Menu_To_Student( Screen, Choice );
         case Choice of
            Tutor:
                  Tutor_The_Student( Screen );
            Practice:
                 Practice_With_Table( Screen );
            Quit:
                 HideAll;
         end;
   until Choice = Quit;
end.
```

The forward directives as well as the module definitions have been listed alphabetically to make the names and the module definitions easier to locate when reading code. At this level of abstraction, calls to procedures are tested as well as side effects on actual parameters. This approach has the advantage of discovering linkage problems between actual and formal parameters before dealing with the detailed steps for the complete algorithm. For example, failing to place a **var** before a variable parameter, failing to match the data types of actual and formal parameters, or a mismatch in the number of actual and formal parameters is easier to notice when the detailed steps of a module have not been defined.

Once this initial testing is completed, we can complete the definitions of the supporting modules at the second level of our structure chart. The first module, `Initialize_Drawing_Window`, requires three major steps:

1. Hide all windows.
2. Establish boundaries for both Text and Drawing windows.
3. Set and show the Text and Drawing windows.

Why do the boundaries of Text window need to be set? THINK Pascal activates the Text window when executing the command `read` or `readln` from standard input. While the command HideAll conceals all windows that are on the screen, it does not prevent the Text window from being opened on execution of any `read` command. Establishing a boundary for the Text window that is off the screen keeps it from appearing in the active viewing region of the screen. This problem does not occur with Macintosh Pascal.

```
Procedure Initialize_Drawing_Window;
{  Purpose: Set the boundaries for both the Drawing window and the
            Text window. The Drawing window is located
            directly below the title bar; the remainder of the
            screen shows only the Drawing window.  }

begin
{ Hide all windows. }
   HideAll;
{ From experimentation, the following values are used to set the
  boundaries for the Text and Drawing windows. }
   Text_Box.Top <-- 0;
   Text_Box.Left <-- 0;
   Text_Box.Bottom <-- -10;
   Text_Box.Right <-- -10;
   SetTextRect(Text_Box);
   Drawing_Box.Top <-- 0;
   Drawing_Box.Left <-- 0;
   Drawing_Box.Bottom <-- 342;
   Drawing_Box.Right <-- 512;
```

```
{ Set the area for the Drawing window and display the Drawing
   window. }
   SetDrawingRect( Drawing_Box );
   ShowDrawing;
end; { Initialize_Drawing_Window }
```

This module employs predefined window-manipulation procedures for both the Text and Drawing windows. `SetTextRect` and `SetDrawingRect` establish the boundaries of the Text and Drawing windows, while `ShowDrawing` displays the Drawing window to the screen. Now we can convert the definition of this module into Pascal code, replacing the present procedure code of `Initialize_Drawing_Window` and retesting `Tutor_System`.

The next module, called `Present_Menu_To_Student`, displays three choices: study a multiplication table, practice multiplication, or quit. As is shown in the top window of Figure 7.18, each option is preceded by a small box. The student chooses an option by clicking the mouse button when the pointer is in one of the three boxes. No input from the keyboard is required. The major steps of this module follow:

1. Fill background with a light-gray pattern, and then draw a rectangle in the center of the Drawing window..
2. Display three options to the student.
3. Continue to check coordinates of the cursor and mouse button until the proper choice has been made.

```
Procedure Present_Menu_To_Student;
{ Purpose:  This module results one of three choices: Tutor,
            Practice, or Quit.  It requires three local
            variables; X and Y representing the coordinates of the
            mouse, and Continue_To_Check. The latter local
            variable is true while the mouse button has not been
            clicked within one of the three small rectangles. }
begin
{ Fill the background of the Drawing window with light-gray
   pattern and then display a rectangle in the center of the
   Drawing window. }
   FillRect ( Box, Ltgray );
   Center_Options_Area;
{ Display three options within this new rectangle. }
   PenSize (2,2);
   MoveTo(85,100);
   WriteDraw('  CHOOSE ONE OF THE FOLLOWING OPTIONS');
   MoveTo (95, 115);
   WriteDraw('  BY CLICKING ON ONE OF THE BOXES.');
   FrameRect(138, 120, 153,135);
   MoveTo(140,150);
   WriteDraw(' Study a multiplication table');
   FrameRect(178,120,193,135);
   MoveTo(140,190);
   WriteDraw(' Practice Multiplication');
   FrameRect(218, 120, 233, 135);
   MoveTo(140,230);
   WriteDraw(' Quit');
{ Now check to see if the mouse button has been clicked. }
  Continue_To_Check <-- true;
      while Continue_To_Check do
         begin {while-do loop}
         { Locate the coordinates of the mouse. }
            GetMouse ( X, Y );
         { Check if mouse button has been pressed in the square
           for the option 'Study a multiplication table.' }
            if (( X >= 120) and (Y >= 138)) and
                  (( X <= 135 ) and ( Y <= 153 )) and Button then
               begin
                  Choice <-- Tutor;
                  Continue_To_Check <-- false
               end
            else
            { Check if the mouse button has been pressed in the
              square for the option 'Practice multiplication.' }
               if (( X >=120 ) and ( Y >= 178 )) and
                  (( X <= 135) and ( Y <= 193)) and Button  then
                  begin
                     Choice <-- Practice;
                     Continue_To_Check <-- false
                  end
               else
               { Check if the mouse button has been pressed in the
                 square for the option 'Quit'. }
                  if (( X >= 120) and ( Y >= 218 )) and
                     (( X<=135 ) and ( Y<=223 )) and Button then
                     begin
                        Choice <-- Quit;
                        Continue_To_Check <-- false
                     end;
         end ;
end; { Present_Menu_To_Student }
```

This module calls on the supporting module Center_Options_Area. This supporting module displays the outline of a center rectangle where messages can later be exhibited. When initially testing this module, we must substitute the steps necessary for performing the equivalent action of Center_Options_Area if the body of this module has not been defined. It is important to include the steps for centering the options area in a separate module, because the actions of this module are called upon later by other superordinate modules.

The module Present_Menu_To_Student employs the procedure GetMouse from the Macintosh Operating System Event Manager for locating the position of the cursor on the screen, and the library function Button for testing if the mouse button is pressed. The module continues to execute the **while-do** loop until the the values of X and Y lie within a region of one of the three small squares. Testing can be done by converting the definition into Pascal code and integrating the code into a modified version of Tutor_System.

The next module, Tutor_The_Student, requires three major steps:

1. Choose a value for the multiplicand.
2. Draw the outline of the multiplication table.
3. Fill the table with values for multipliers, multiplicands, and products.

```
Procedure Tutor_The_Student;
{ Purpose:  This module presents a multiplication table for any
            student to  review. It requires only one value
            parameter, Drawing_Box, representing the screen
            boundary. The only local variable is Multiplicand. }
begin
{ Choose a value for Multiplicand. }
   Choose_Multiplicand ( Drawing_Box, Multiplicand );
{ Draw the outline of the multiplication table. }
   Draw_Mutiplication_Table( Drawing_Box );
{ Fill the multiplication table with multiplication rules. }
   Fill_Multiplication_Table( Multiplicand );
end; { Tutor_The_Student }
```

It is clear from the beginning that each of the major steps will require a significant amount of coding, so our approach is to express each step as a call to a subordinate module and define the detail later. In testing this module, the definitions of supporting modules as stubs are used to test the linkages between actual and formal parameters. Pascal does require that each of the procedures `Choose_Multiplicand`, `Draw_Multiplication_Table`, and `Fill_Multiplication_Table` be declared ahead of the procedure `Tutor_The_Student` in the program `Tutor_System`. We can avoid this by adding forward directives for each of these supporting modules to our initial list of directives in program `Tutor_System`.

The next module, `Practice_With_Table`, requires four major steps:

1. Choose a value for the multiplicand.
2. Fill the background of the screen with a white pattern, and display a center rectangle for viewing information.
3. Test the student's skill over the multipliers 1 through 12, returning the number of correct answers.
4. Report on the progress of the student's skill.

```
Procedure  Practice_With_Table;
{ Purpose:  This module lets the student practice with his or her
            own multiplication table. It has only one value
            parameter, Drawing_Box,  and two local variables:
            Multiplicand and Number_Correct_Answers. }
begin
{ Prompt student for a value of Multiplicand. }
   Choose_Multiplicand( Drawing_Box, Multiplicand );
{ Test student's multiplication skills. }
   Test_Students_Skills ( Drawing_Box, Multiplicand,
                          Number_Correct_Answers );
{ Report on the student's progress. }
   Report_On_Students_Progress ( Number_Correct_Answers );
end; { Practice_With_Table }
```

When writing the detailed steps for the module Practice_With_Table, we realize that choosing a value for the multiplicand is the same action required in module Tutor_The_Student. Rather than repeating this code, we invoke the module Choose_Multiplicand for both steps in modules Practice_With_Table and Tutor_The_Student, leaving Choose_Multiplicand as a supporting module for later definition. Figure 7.20 shows the structure chart with the added subordinate modules. Lines with arrowheads indicate the direction in which the values of parameters are being passed.

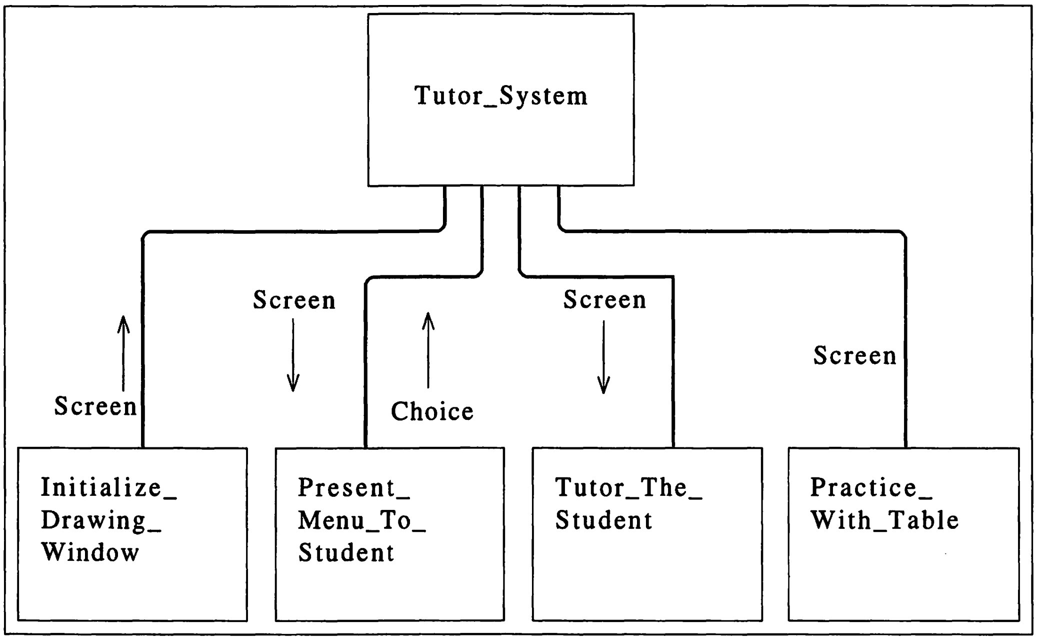


Figure 7.20 Structure chart for Tutor_System.

7.7.4 Abstraction 4

At the next level of abstraction, definitions of the supporting modules Center_Options_Area, Choose_Multiplicand, Draw_Multiplication_Table, Fill_Multiplication_Table, Test_ Students_Skills, and Report_On_Students_Progress are given. The first, Center_Options_Area, draws the center rectangle for displaying prompts:

```
Procedure Center_Options_Area;
{ Purpose:  Establish a rectangular area for viewing various
  options. }
begin
{ Draw a rectangle in the center of the Drawing window, composed
  of a white background. }
   FillRect( 75, 50, 250, 370, White );
{ Draw a black border about this new rectangle. }
```

```
   PenSize (5, 5);
   FrameRect(75, 50, 250, 370);
{ Set the font type and size for characters written to this new
   rectangle. }
   TextFont(3);
   TextSize(12);
end; { Center_Options_Area }
```

The next module, Choose_Multiplicand, fills the background with a light-gray pattern, draws a center rectangle, and then prompts the student for the value of the multiplicand. If the value for the multiplicand is out of range, this module causes the computer to beep (emit a short sound from the speaker), erase the prompt and the improper value from the screen, and then repeat the step of requesting a value for a multiplicand.

```
Procedure Choose_Multiplicand;
{ Purpose:  This module returns a value representing the
  multiplicand. It has an additional value parameter Drawing_Box.
  It will require a delay for viewing the screen. }
begin
{ Fill the background of the Drawing window with a gray pattern
  and then draw the center rectangle. }
   FillRect( Drawing_Box, Gray );
   Center_Options_Area;
{ Prompt the student for a multiplicand. }
   MoveTo(110, 160);
   WriteDraw(' Enter a multiplicand: ');
   readln( Multiplicand );
   WriteDraw( Multiplicand );
{ Provide a delay of one second. }
{ Check if the value of the multiplicand is within range. }
   while (Multiplicand < 1) or (Multiplicand >12) do
      begin
      { Provide short beep. }
         SysBeep(12);
      { Erase the region containing the prompt. }
         EraseRect(150, 110, 160, 350);
      { Prompt the student for another value. }
         WriteDraw(' Enter multiplicand: ');
         readln( Multiplicand );
         WriteDraw( Multiplicand );
      { Provide a delay of approximately one second. }
      end;
end; { Choose_Multiplicand }
```

For the beep, we use the procedure SysBeep. This procedure provides a multiple of 12 short (0.022-second) tones to the speaker. For establishing a delay of one second, one of two options is available. If you are using THINK Pascal, execution of the Operating System Utilities Library procedure Delay can result in multiples of sixtieths-of-a-second time delays. This procedure has the following header information:

```
procedure Delay( Number_Ticks:longint; var Final_Time:longint );
```

The parameter `Number_Ticks` represents the delay, and `Final_Time` is system-clock time from when the Macintosh system began execution until the end of the delay. For a one-second delay, this procedure is invoked by the command

```
Delay( 60, Time );
```

In Macintosh Pascal we must define our own procedure for delaying execution, because Macintosh Pascal has no access to the routines in the Operating System Utilities Library. The following procedure is suggested as a replacement for procedure `Delay`:

```
procedure Delay_Execution ( Time: integer );
{ Purpose:  This procedure delays execution of an application. }
   var
      I, Result : integer;
begin
   for I := 1 to Time do
      Result := BitShift( I, 1000 );
end; { Delay_Execution }
```

The **`for`**-loop and the `BitShift` operation are used to delay the execution of an application using this procedure. Depending upon the speed of the microprocessor, the execution of the command `Delay_Execution(240)` should provide a 3-4 second delay.

For module `Choose_Multiplicand`, why not use the `writeln` command instead of invoking the `WriteDraw` procedure? The `writeln` procedure writes only to the Text window, not the Drawing window. We need the `WriteDraw` procedure to display the value of expressions within the Drawing window. In addition, we can use the subordinate module `Center_Options_Area` to draw the center rectangle in the Drawing window. Values for the actual parameters of `FillRect` and `FrameRect` were picked by trial and error.

```
Procedure Draw_Mutiplication_Table;
{ Purpose:  This module draws a multiplication table under the
            option to tutor a student. It requires a single value
            parameter called Drawing_Box and uses only a single
            local variable, Y. The variable Y represents a
            vertical position for displaying horizontal lines to
            the Drawing window. }
begin
{ Fill the Drawing window with a white background, and display
  a title at the top of the table. }
   FillRect( Drawing_Box, White );
   PenSize(1, 1);
   MoveTo(180, 45);
   WriteDraw( 'MULTIPLICATION TABLE' );
{ Display the border of the table. }
   FrameRect(50, 50, 230, 450);
{ Draw five horizontal lines across the rectangle. }
```

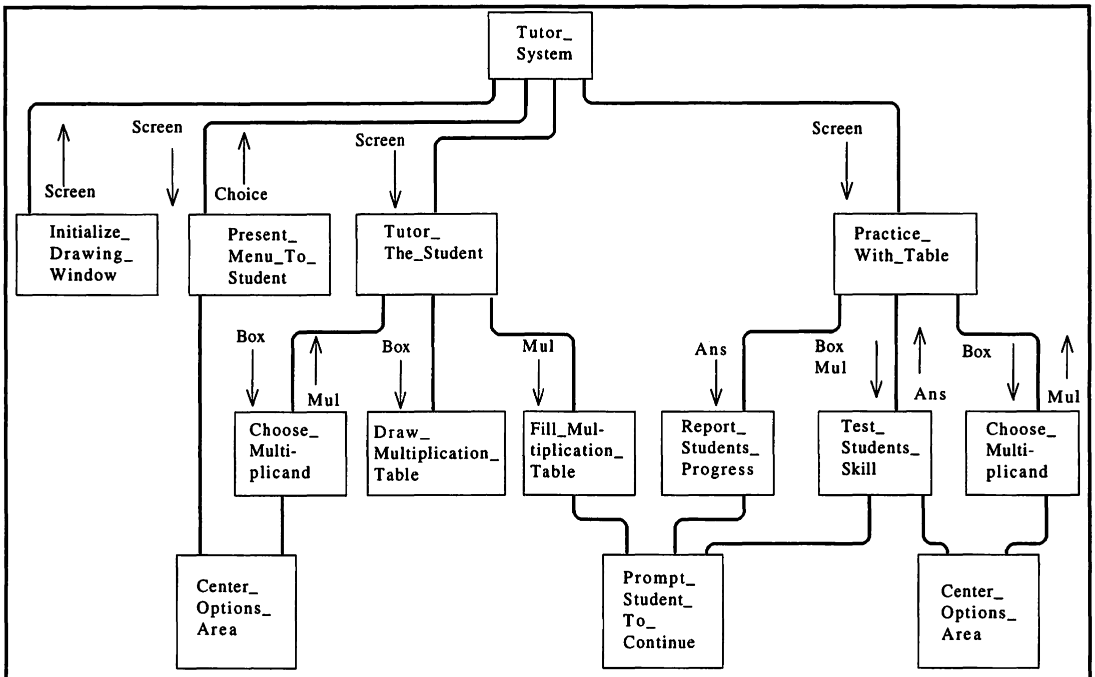
```
   Y <-- 80;
   repeat
      DrawLine (50, Y, 450, Y);
      Y <-- Y + 30;
   until Y = 230;
{ Draw a single vertical line through the middle of this
  rectangle. }
   DrawLine (250, 50, 250, 230)
end; { Draw_Mutiplication_Table }


Procedure Fill_Multiplication_Table;
{  Purpose: This module fills the contents of smaller rectangles
            in the multiplication table with strings composed of a
            multiplier, multiplicand, and product. }
begin
{ Initialize Multiplier and the X coordinate. }
   Multiplier <-- 1;
   X <-- 70;
{ Fill the subtable on the left and then on the subtable on the
  right with basic multiplication rules. }
   for Outer_Count <-- 1 to 2 do
      begin
      { Initialize the Y coordinate. }
         Y <-- 70;
         for Inner_Count <-- 1 to 6 do
            begin
            { Fill a subtable with six multipliers. }
               MoveTo( X, Y);
               WriteDraw( Multiplier:3, '  X' ,   Multiplicand,
                          '=', Multiplier  *  Multiplicand:3 );
            { Increment Multiplier and the Y coordinate. }
               Multiplier <-- Multiplier + 1;
               Y <-- Y + 30
            end;
      { Modify the X coordinate to fill the right subtable. }
         X <-- X + 200
      end;
{ Prompt the student to continue execution. }
   Prompt_Student_To_Continue;
end; { Fill_Multiplication_Table }
```

The coordinates for filling the small rectangles were chosen by trial and error, using a simple test program.

```
Procedure Prompt_Student_To_Continue;
{ Purpose:     This module prompts the student to continue
               execution by clicking the mouse button. }
begin
{ Display prompt for continuing execution. )
   MoveTo(110, 270);
   WriteDraw(' Press the mouse button to continue : ');
   while not( Button ) do  { wait for button to be pressed };
{ Provide a time delay for the mouse button to be released. }
end; { Prompt_Student_To_Continue }


Procedure Test_Students_Skills;
{ Purpose:  This module checks the multiplication skill of the
            student. It requires two value parameters; Drawing_Box
            and Multiplicand, and one variable parameter called
            Number_Correct_Answers. Local variables include
            Multiplier, Product, and Student_Answer. }
begin
{ Display a background color of white to the Drawing window, and
  then draw the center rectangle. }
   FillRect( Drawing_Box, White );
   Center_Options_Area;
{ Prompt the student with 12 questions. }
   for Multiplier <-- 1 to 12 do
      begin
      { Prompt the student with a question. }
         MoveTo( 90, 130);
         WriteDraw( Multiplier, '  X', Multiplicand, '   =   ?' );
         readln( Student_Answer);
      { Delay execution for approximately one second. }
      { Erase the prompt and replace it with the student's
         response. }
         EraseRect( 120, 60, 130, 350);
         MoveTo( 110, 130);
         WriteDraw(Multiplier. '  X' , Multiplicand. '   =' );
         WriteDraw( Student_Answer );
         Product <-- Multiplier * Multiplicand;
      { Check if the student's answer is correct. }
         MoveTo (170, 185);
         if Student_Answer = Product  then
            begin
               Number_Correct_Answers
               <-- Number_Correct_Answers + 1;
               WriteDraw( 'CORRECT' )
            end
         else
            begin
               SysBeep(12);
               WriteDraw( 'WRONG' )
            end;
         Prompt_Student_To_Continue;
      { Erase center rectangle. }
         Center_Options_Area
      end;
end; { Test_Students_Skills }
```

```
Procedure Report_On_Students_Progress;
{ Purpose:  This module provides a short report on the student's
            progress. It requires one value parameter,
            Number_Correct_Answers. No local variables are
            necessary. }
 begin
{ Display the number of correct answers.  }
   MoveTo(110,  140);
   WriteDraw(' Number of correct answers: ',
Number_Correct_Answers);
{ Determine and display the progress of the student. }
   MoveTo( 175,190);
   if Number_Correct_Answers > 10  then
      WriteDraw( 'EXCELLENT' )
   else
      if Number_Correct_Answers > 7  then
         WriteDraw( 'GOOD' )
      else
         WriteDraw( 'POOR' );
{ Delay execution while the student reads his or her score. }
{ Prompt the student to continue execution. }
   Prompt_Student_To_Continue;
end; { Report_On_Students_Progress }
```

Figure 7.21 shows the complete structure chart for Tutor_System. Some of the module names have been duplicated to make the chart easier to read. Lines with arrows are provided to show the direction in which the values of parameters are being passed. Keep in mind that developing, coding, and testing software is a process based on experience as well as trial and error, motivated by a desire to improve the design. In this example, the values of actual parameters for library procedures MoveTo, FillRect, FrameRect, were changed to determine the best effects on the screen. This at times required modification of steps in module definitions as well as in the main module. For example, the call to the module Prompt_Student_To_Continue was removed from Tutor_The_Student and Practice_With_Table and then embedded in the supporting modules Fill_Multiplication_Table and Report_On_Students_Progress to allow these modules to act as supporting utility modules. The need for accurate requirements helps us to understand the problem, and the implementation of a proper design directs us in writing the proper code. Before ending this discussion, let us consider a few additional issues related to writing modules. First, a module should be written to be functionally cohesive. In theory this is the most desirable form for a module. It implies that the parts of the module perform a single, elementary function. In practice, this may lead to applications with an unreasonable number of modules. Often, we may compromise on functional cohesion by practicing the principle of sequential cohesion, by which a module is designed to perform several functions of a supportive and related nature. For example, module Choose_Multiplicand is responsible for returning the value of a multiplicand, but it performs several actions that support its major purpose: prompting for a value for the multiplicand, reading a value from the keyboard, and checking that the value of the multiplicand is within the range 1 through 12.



Notes:
The parameter `Box` is an abbreviation for `Drawing_Box`
The parameter `Mul` is an abbreviation for `Multiplicand`
The parameter `Ans` is an abbreviation for `Number_Correct_Answers`

Figure 7.21 A detailed structure chart for the program `Tutor_System`.

Although each action could be written as a separate module, the combined steps in one module are easily understood. The size of the module depends more on the functions that it supports than on the number lines of code that are written. Some programmers define a module to be a body containing no more than 50 or 100 lines of executable code, but this definition does not take into account the principles of functional and sequential cohesion. If you feel the need to limit the size of a module, never exceed nine major steps when defining the algorithm for the module. This is in keeping with the concept that a person manages well when he or she is limited to supervising no more then seven to nine individuals.

A second major point in the development of modules is to practice the principle of loose coupling (also called *data coupling*). With loose coupling, only the information that is needed is passed between modules; no more and no less. For example, while module `Initialize_Drawing_Window` modifies the dimensions of Text window, these dimensions are not returned, because there is no need to share them with other modules. Returning the dimensions of the Drawing window is necessary, because these are shared with other modules. Common coupling can be practiced by assigning the dimensions of Text and Drawing windows to global variables, but this could limit our modular independence if we decide later to reassign dimensions of our windows during the execution of several different modules.

Implementation of the complete Tutor_System program is left as an exercise. Enhancements to this application can include several options for tutoring students on addition, subtraction, and division. We can select the next pixel location in the Drawing window where the value of an integer is to appear, so we can extend our application for tutoring students on the use of fractions.

7.8 WHITE-BOX VERSUS BLACK-BOX TESTING

Testing is an important aspect of software design and is not limited to the stage of coding the program. When the body of a module is being written, it is important to establish test cases for exploring the correctness of the solution. When defining the algorithmic steps of a module, we can select test cases for checking the executability of the module itself. By hand-tracing the algorithm, we better understand its function as well as what is expected for the values of identifiers at intermediate steps. In addition, we are examining the logical structure of the program for correctness. This method of testing is referred to as *white-box* (*glass-box*) testing, because we can see the actual code that we are testing through the module's executable paths. Grouped as a set, the test cases must be able to show that all paths are executable for the method to be successful. Although this method of testing cannot guarantee that the algorithm for the module is correct in relation to its function, it does guarantee that each path of the algorithm is executable.

As an example of white-box testing, consider the module Choose_ Multiplicand that appeared earlier in this chapter. The conditions for testing the while-loop within the body of the module come from entering values at the keyboard beyond the range 1 through 12. In particular, three types of tests are required: (1) where the value of the multiplicand is less than 1, (2) where the value of the multiplicand is greater than 12, and (3) where the value of the multiplicand is between 1 and 12. Together these tests result in all statements of the module being tested for executability. By using this set of tests, we are checking both valid as well as nonvalid values of the multiplicand.

A second approach for testing modules is referred to as *black-box* testing. Here a module is tested solely on the basis of its documentation, that is, the definition of its function(s) and a list of its parameters. It is the responsibility of the programmer to consider test cases for verifying the accuracy of the documentation by writing one or more test programs that can invoke the module and observe its effect. We refer to this as *black-box testing* because the internal code of the module is hidden from view; only the interface is available for interaction with the body of the module. One way to choose values for actual parameters is to try trivial values, typical but realistic values, extreme values, as well as illegal values. Trivial values test for simple cases, whereas typical values allow us to test for values within a range of acceptable values. For extreme values, we can view the actions of a module given excessive limits, whereas illegal values allow us to observe the actions of a module when handling improper data. Not every module that we are testing requires all four categories of test data to be defined. The choice depends upon the definition of the module and upon the integration of the module within the framework of a larger application.

For example, the procedure FillRect is specified by the following procedure header and definition:

```
procedure FillRect( R : Rect; Pat : Pattern );
{ Purpose:  This procedure fills the specified rectangle R with a
            pattern given by the value of Pat. The QuickDraw
```

```
        library has five predefined patterns limited to White,
        Black, Gray, LtGray, and DkGray. }
```

For testing `FillRect`, we can invoke this procedure from a loop iterating over the range of a variable representing values from `white` to `dkGray` for a fixed specified rectangle. Such a loop would offer us a better impression for a background pattern when viewing the Drawing window. For trivial data we might establish a rectangle with both the top-left and bottom-right points of `R` being (0,0). A realistic value for rectangle `R` might be the points (75,50) and (250,370), whereas an extreme value, depending upon the size of screen, would be the points (0,0) and (512,342). We might try choosing illegal values that are either less than –32767 or greater than 32767, but this would only result in a translation error, because the elements of type `Rect(Left, Top, Right, Bottom)` are all defined as `integer` types.

A third approach to testing is known as the *ticking-bomb* method. This method assumes that little or no testing is necessary once the application has been coded and appears to be operational. It assumes that the customer, by using the application, can discover all further errors. Unfortunately, this approach to testing is a time bomb waiting to detonate. It can leave the user with little or no confidence in the program.

7.9 STANDARD PASCAL VERSUS THINK PASCAL

For procedures and functions, how does THINK Pascal differ from ANS (American National Standards) Pascal? In ANS Pascal, the body of a function must contain at least one assignment statement that assigns a value to the name of the function. In THINK Pascal, this requirement is not enforced. In THINK Pascal a function can be written to act like a procedure; that is, the body of a function can be used to cause side effects upon actual parameters associated with formal parameters. The function still must be invoked from an expression and will still return a value (random) through the name of the function.

THINK Pascal does support extensions for procedures and functions not specified by ANS Pascal. First, THINK Pascal relaxes the restrictions on the ordering of declarations. Functions and procedures can be declared before variables, and variables before types and constants. The only requirement is that an identifier must be declared before being referenced in any further declaration.

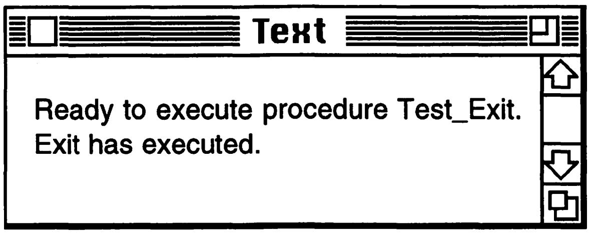
Second, THINK Pascal supports the definition of inline routines (procedures or functions) for directly embedding machine code within a Pascal program. The machine code that is listed following the header of an inline routine is placed directly within the translated code of the program in which it is invoked. Inline routines are common in the languages FORTRAN and C, as well as in assembly language. In some instances they are called *macro definitions*.

Third, THINK Pascal supports a predefined command called `exit`, which allows a program to exit from an enclosing procedure. Syntax for using this command is

```
Exit( Procedure_Name )
```

where `Procedure_Name` is the name of the enclosing procedure from which the program can exit. When executed, the program returns to the statement following the point where the procedure was invoked. An exit statement may also be placed within a nested procedure that is enclosed by `Procedure_Name`. Figure 7.22 shows a simple program that tests the `Exit` command.

```
program  Testing_Exit_Statement(input,  output);
{ Purpose:  This program shows a simple test of the }
{            command Exit. }
   var
      Test_Path: boolean;

procedure  Test_Exit(var  Check:  boolean);
   procedure  Nested_Procedure;
   begin
      Exit(Test_Exit);
   end; { Nested_Procedure }
begin
    Check := true;
     Nested_Procedure;
    Check := false;
end; {Test_Exit }

begin
     ShowText; writeln('Ready to execute procedure Test_Exit.');
     Test_Exit(Test_Path);
   if Test_Path then
       writeln(' Exit has executed. ')
   else
       writeln(' Exit has failed to execute. ');
end.
```



```
Ready to execute procedure Test_Exit.
Exit has executed.
```

Figure 7.22 A simple program for testing the command `Exit`.

The `Exit` command offers the programmer an equivalent to the concept of the `return` statement found in other computer languages like FORTRAN, C, and Ada. In short, when within the body of a procedure we may find ourselves ready to return (exit) from the procedure for a condition that is *true*. Having to assign a value to a sentinel and test for the condition of a sentinel may be awkward and confusing to a person reading the body of the procedure.

Executing a statement such as

```
if condition then
   Exit( Procedure_Name );
```

seems more natural to the solution of a problem and indicates where the procedure will be exited. It does violate structured programming, because there is more than one place where the procedure ends execution. Which is better depends on the physical length of the procedure body and on the person writing the code.

Fourth, THINK Pascal allows the use of the qualifier **univ** for disabling type-checking of a routine's parameter. When a formal parameter is qualified with the reserve word **univ**, the type for the actual parameter is not required to match that of its corresponding formal parameter. What is required is that both the qualified formal parameter and actual parameter be types with the same memory size. For example, the following program `Testing_Univ_Qualifier` shows the value of a 32-bit `longint` type parameter being used to initialize the value of a 32-bit `real` type. Although both the formal and actual parameter types are different, they do have the same memory size.

```
program Testing_Univ_Qualifier(input, output);
{ Purpose:  This programs shows a simple test for the univ }
{           qualifier.}
   var
      N: real;
   procedure Test_Univ  (var Number: univ longint);
   begin
      Number := 0;
   end; { Testing_Univ }
begin
   ShowText;
   N := 3.333;
   writeln('Value of N before execution of Test_Univ: ', N: 7: 5);
   Test_Univ(N);
   writeln('Value of N after execution of Test_Univ : ', N: 7: 5);
end.
```

It is important to avoid the use of extensions in THINK Pascal if your code will be used with other types of Pascal translators. Not all Pascal translators apply the same set of extensions.

SUMMARY

There are several important points to remember about procedures and functions. First, they can reduce the need to write repetitive code. For example, if your program requires several different types of tables, you can write one procedure containing nested **for** loops, with the maximum number of iterations representing the maximum number of rows and columns. These **for** loops need only be written once, because we can control the number of rows and columns being displayed through the parameter list.

Second, procedures and functions allow us to develop and test large programs in a modular fashion. Each program can be developed with basic building blocks, each block representing a major step in the solution of a problem and evolving into a refined module, and each module being tested and integrated into the whole program. This allows the main body of the Pascal program to act as a supervisor over subordinate procedures and functions.

Third, the actual parameter list and the formal parameter list provide the means for transferring information between the environment that has called on the subprogram and the environment of the subprogram itself. Formal parameters can use names that reflect the nature of the data objects described in our algorithms and act as variables local to the programming environment represented by the procedure or function. The environment for the subprogram becomes active when it is the called on from some other local programming environment (subprogram or main program) and becomes deactivated when the subprogram ends execution. Pascal procedures or functions can have constants, user-defined types, and variables that are local to the body of the subprogram. In turn, a procedure or function can also share global constants, types, and variables declared outside of its own environment.

Fourth, procedures and functions provide a mechanism for extending the definitions and capabilities of a programming language. With subprograms we can create our own library of utilities that goes beyond the given set of Pascal library procedures and functions.

What are the differences between a procedure and a function in Pascal? A procedure is used for its side effects. This can be the execution of a process: drawing to the screen, displaying of text, or the modification of the value of an actual parameter. Procedures can affect the value of an actual parameter only if its corresponding formal parameter is a variable type. In addition, an actual parameter can be represented by an expression if its corresponding formal parameter is a value type; otherwise, the actual parameter must be a variable. A function can be used to return a value through its own name, in addition to using it for its side effect. A procedure is called by executing a calling statement, whereas a function can only be executed when called from within an expression.

Pascal procedures and functions can be written recursively; that is, they can either be written to call directly on themselves or, to do so indirectly, by calls from other procedures or functions. When writing directly recursive procedures or functions, keep in mind the heuristic steps discussed earlier. Be sure that you understand the trivial and nontrivial cases, using the trivial cases to terminate the recursion. Use recursive steps to reduce values closer to the trivial cases, and employ several nontrivial cases to test the recursive definition.

Pascal allows the names of procedures and functions to be passed as arguments, allowing us to write general subprograms in terms of formal parameters. On execution, the names of previously defined procedures or functions can be substituted and called on.

REVIEW QUESTIONS

1. Define the terms *procedure* and *function.*
2. What is meant by the expressions *user-defined procedure* and*user-defined function*?
3. Why is it important for a higher level language to support the concept of procedures and functions?
4. What is meant by the term *subprogram*?
5. List and briefly define each of the library procedures that we have already used in Pascal.
6. List and briefly define each of the library functions that we have already used in Pascal.
7. Describe the structure of a Pascal program.

8. Describe the structure of a user-defined Pascal procedure.
9. Why is the structure of a Pascal procedure similar to the structure of a Pascal program?
10. What is the purpose of the formal parameter list?
11. Can a **uses** clause follow a procedure-header?
12. How are constants, user-defined types, and variables declared in a procedure?
13. What is meant by the term *compound statement*?
14. What is the difference between an actual parameter and a formal parameter?
15. Why is the complete structure of a procedure considered a declaration?
16. Why does the declaration of a procedure end with a semicolon, while a Pascal program ends with a period?
17. What are the rules for naming a procedure?
18. For the following short program, what represents the formal parameters and what represents the actual parameters?

```
program Sample;
   const
      A = 3;
      B = 4;
   var
      Total : integer;
   procedure X( N, M, T : integer );
   begin
      writeln('The sum of ', N:2, ' and ', M:2, ' is ', T:3 )
   end;
begin
   Total := A + B;
   X( A, B, Total )
end.
```

19. For Question 18, what is the purpose of the procedure X? Can you think of a better name than just plain X?
20. In some Pascal systems, a procedure cannot have the same name as the program. Is this true in Macintosh Pascal?
21. Modify the program `Diagonal_Lines_Revised`, so that a procedure called `Initialize` is executed, hiding all the windows, setting the Drawing window size, and then opening the Drawing window before the diagonal lines are drawn. The procedure `Initialize` must be called from within the body of the main program.
22. Will the following Pascal program execute? Why or why not?

```
program Sample;
   procedure Output( N, M, T : integer );
   begin
      writeln('The sum of ', N:2, ' and ', M:2, ' is ', T:3 )
   end;
   var
```

```
      Total : integer;
   const
      A = 3;
      B = 4;
begin
   Total := A + B;
   Output( A, B, Total);
end.
```

23. What is meant by the term *value parameter*?
24. In the following program, what effect does the procedure `Double` have on the three actual parameters `X`, `Y`, and `Z`?

```
program Example;
   var
      Z : integer;
      X, Y : real;
   procedure Double( C : integer;  A, B : real);
   begin
      A := 2 * A;
      B := 2 * B;
      C := trunc( A + B );
   end;
begin
   write(' Enter two numbers: ');
   readln( X, Y );
   Double( Z, X, Y );
   writeln( X, Y, Z )
end.
```

25. What is the effect on `X`, `Y`, `Z` when the following program is executed?

```
program Try_Again;
   var
      Z : integer;
      X, Y : real;
   procedure Double( C : integer;  A, B : real);
   begin
      C := trunc( A + B );
   end;

begin
   write(' Enter two numbers: ');
   readln( X, Y );
   Double( Z, 2 * X, 2 * Y );
   writeln( X, Y, Z )
end.
```

26. What is meant by the term *variable parameter* ?

27. Modify the short program Example in Question 24 so that all formal parameters are variable parameters.
28. Consider the following program, titled Try_Over_Again:

```
program Try_Over_Again;
   var
      Z : integer;
      X, Y : real;
   procedure Double( C : integer; var A, B : real);
   begin
      C := trunc( A + B );
   end;
begin
   write(' Enter two numbers: ');
   readln( X, Y );
   Double( Z, 2 * X, 2 * Y );
   writeln( X, Y, Z )
end.
```

What syntactic and semantic errors keep the variable Z from receiving a computed value?

29. How can the program in Question 28 be modified so that when the procedure Double is executed, the values of both X and Y will be doubled before Z <-- trunc(X + Y)?
30. Modify your program in Question 28 by defining an additional procedure called Input_Data. When executed, this procedure should display the message Enter two numbers: and read two real numbers typed from the keyboard.
31. Describe the structure of a user-defined Pascal function.
32. How is a function called during the execution of a Pascal program?
33. What are the major differences between a Pascal function and a Pascal procedure?
34. In the body of a function, how does the name of the function obtain a value?
35. The following program uses a function called Sum and was written for the purpose of computing the summation of a set of integer numbers from 1 to *n*. Correct the syntactic and semantic errors so that it will execute correctly.

```
program Sum:
   var
      I : integer;
   function Sum ( Limit : real ) : integer;
      var
         Counter : real;
   begin
      for Counter := 1 to Limit do;
         Total := Total + Counter;
      Sum := Total
   end;
```

```
begin
   Sum( I );
end.
```

36. The following is a procedure for computing the product of integers from 1 to *n*. Rewrite this procedure as a function.

```
procedure Product( N : integer; var Result : real );
begin
   Result := 1.0;
   while N > 0 do
      begin
         Result := Result * N;
         N := pred( N )
      end;
end;
```

37. Would the condition `N > 1` in the **while-do** loop of the procedure `Product` provide the same answer?
38. What is the value returned for `Result` if the value of `N` is negative?
39. What is meant by the terms *global* and *local variables*?
40. Can constants and user-defined types be global as well as local data objects? Explain your answer through some short examples.
41. What is the advantage of having a global data object? What can be a disadvantage of having a global object?
42. Trace the program `Bad_Habits` both by hand and by using the Observe window during execution to convince yourself that it will never end execution unless you select the option **Halt** from the menu **Pause**.
43. Correct the program `Bad_Habits` so that it will execute to completion.
44. Why are forward declarations necessary?
45. Use forward declarations to keep the following procedures in the order in which they are declared. Check your code by using the **Check** mode of the **Run** menu.

```
procedure Total( N ; integer : var Value : integer );
begin
   Value := 0;
   while N > 0 do
      begin
         Add_and_Decrement( Value, N );
      end;
end;

procedure Add_and_Decrement( var R, M : integer );
begin
   R := R + M;
   Decrement( M );
```

```
end;

procedure Decrement( var X : integer );
begin
   X := pred( X )
end;
```

46. Rewrite the procedures `Total` and `Decrement` in Question 45 as functions, while keeping the order in which they are declared.
47. What is the concept of modular programs?
48. What do structure charts represent?
49. How do abstractions help in defining a solution to a problem and in developing algorithms specifying a solution?
50. What is meant by the term *stub*?
51. How should testing be introduced as a large program is being developed? What advantages do you see in this approach for testing? What disadvantages do you see?
52. How can the names of procedures and functions be used as values of actual parameters in Pascal? How must the corresponding formal parameters be declared?
53. What are the advantages of being able to pass the names of routines to formal parameters?
54. What is meant by the term *recursion* ?
55. In the discussion of the concept of recursion, trace the steps necessary to compute the factorial of 10. Then trace the necessary steps to compute `Sum(10)`.
56. What is meant by *recursive function* or *recursive procedure*?
57. What is meant by the statement that a procedure or function is directly recursive?
58. What is meant by the statement that a procedure or function is indirectly recursive?
59. What is the purpose of the stack when executing a recursive routine in Pascal?
60. Using the function `Factorial`, draw the stack for computing `Factorial(10)`.
61. List the four useful heuristics for defining a recursive procedure or function.
62. Write a recursive function for `Sum`.
63. Can user-defined procedures and functions be parameterless? If you are not sure, think of a simple example to test your answer.
64. Can a user-defined function have **var** parameters? If you are not sure, think of a simple example to test your answer.

PROGRAMMING EXERCISES

Although not all programming exercises require you to write an algorithm, you may better understand the problem and what is required by first writing an algorithm and tracing it by hand with several examples before writing a Pascal program.

1. Write a procedure called `Minimum` that has three formal parameters. This procedure must compute the minimum for two of the formal parameters, and return the smallest value through the third formal argument.

2. Using only two formal parameters, write a function for the problem in Exercise 1.

3. Write a procedure called `Response` that has only one variable parameter, called `Answer`. `Answer` must also be declared as type `string`. When executed, this procedure will prompt the user with the following message: `Type "YES" to continue, "NO" to quit:` After the user has typed a response, this procedure will check `Answer` to see if the user has typed `YES`, `YEs`, `YeS`, `Yes`, `yES`, `yEs`, `yeS`, `yes`, `NO`, `No`, `nO`, or `no`. If none of these has been typed, the procedure must produce a short tone using the routine `SysBeep`, display the message `IMPROPER RESPONSE`, and then prompt the user to re-enter an answer.

4. Write a procedure that can determine the largest and smallest of a set of numbers entered from the keyboard. This procedure must prompt the user, read the next number, and report the largest and smallest values.

5. Some Pascal systems support a remainder operator that is different from the **mod** operator. Write a procedure called `Quotient_Remainder` for computing both the quotient and the remainder of two integers, `M` and `N`, where `M` is divided by `N`. Assume two value parameters `M` and `N` and two variable remainders `Q` and `R`. *Hint* : Remember that `N/M = Q + R/M` where `Q` represents a quotient and `R` the remainder.

6. Write a function called `Rem` that has two value parameters, `M` and `N`. This function returns the integer remainder of two integers `M` and `N`, where `M` is divided by `N`. How does this function compare with the **mod** operator?

7. Write a nonrecursive function for the function `Fibonacci` in the program `Computing_Nth_Fibonacci`. Compare the times required for computing the *n*th term of the Fibonacci series using both recursive and nonrecursive functions with *n* equal to 32.

8. Write a procedure for computing the mean or average of a set of numbers entered from the keyboard. This procedure prompts for the first and next numbers and continues prompting and reading numbers until the user wants to end input. Once the mean has been computed, this procedure displays the mean and quantity of numbers entered from the keyboard. The numbers entered from the keyboard can be either positive or negative.

9. Write a function titled `Floor`. This function will have only one formal parameter declared as `real`. It will return the largest integer that is less than or equal to the value of the actual argument. For example,

`Floor(4.89)` is 4 and `Floor(5)` is 5. In the case of negative numbers, `Floor(-4.89)` is -5 and `Floor(-4.005)` is also -5.

10. Using the Pascal library function **random**, write a function called `Roll_of_Dice` for returning a random number 1 through 6. This is to represent the roll of a single die. Add this function to a short program. By executing a loop 1000 times, count the number of times that sides 1, 2, 3, 4, 5, and 6 appear, and see for yourself if each side is equally likely to appear; that is, see if the total count for any side divided by 1000 is close to the ratio 1/6.

11. Write a procedure called `Sample_Data` for displaying to the Drawing window a bar chart for the six total die counts.

12. Write a function for computing the common logarithm given by the relationship

$\log_{10} m = \ln(m) / \ln(10)$ where $m > 0$.

If $m <= 0$, have your function return the value zero and report the message

```
Value undefined for zero or negative argument.
```

13. Write a function that computes X^Y using the expression $e^{Y \ln(X)}$. In executing this function, return a value for the following conditions:
    (a) X is zero and Y is nonzero.
    (b) X and Y are both zero.
    (c) X is negative and Y is less than 1.
    For cases (b) and (c), report the message `Improper arguments for computing a number raised to a power.`

14. Write a recursive function for `Exponentiation`. Remember that the value for `Exponentiation(X,0)` is 1 and `Exponentiation (X, N)` is `X * Exponentiation( X, N-1 )`, where N is greater than zero.

15. The greatest common divisor of two integers can be computed recursively from the following definition:

```
Great_Common_Div( M, 0 ) = M
Great_Common_Div( M, N ) = Great_Common_Div( M, M mod N ),
```

where N is greater than zero. Write a recursive function called `Great_Common_Div` for computing the greatest common divisor of two integers, M and N.

16. Write a program that will display a table of Fahrenheit versus Celsius temperatures to the Text window, given an initial temperature, final temperature, and incremential value. In this program the initial temperature can exceed the final temperature if the incremental value is

negative. Use procedures for setting and opening the Text window, entering initial data from the keyboard, and checking that this data is correct; use a function for computing a temperature, a procedure for displaying the header for the beginning of a table, and a procedure for displaying the table of temperatures.

17. Write an algorithm for counting the number of vowels in a sentence entered from the keyboard. After testing this algorithm by hand, convert it into a procedure and retest it. This procedure is required to prompt for a line of text. As each character is being typed and read, it is checked to see if it is a vowel. This continues until a period is read. After all characters have been read and checked, a report is given on the number of vowels read from input. This procedure must be able to recognize both uppercase and lowercase letters.

18. Consider the table shown in the Figure 7.23:

Quantity Ordered	**Price Per Piece**
1–99	1.01
100–199	0.98
200–299	0.95
300–399	0.90
400–499	0.85
500 and above	0.75

Figure 7.23

Write a program that executes each of the following steps by calling a separate procedure:

(a) Hide all the windows, and then set and open the Text window in the middle of the screen.
(b) Prompt the user for quantity ordered. If the quantity is less than 1, provide a short tone, clear the Text window executing the routine `Page`, and again prompt the user for input.
(c) Compute the total cost by performing a simulated table lookup.
(d) Compute the sales tax and total cost.
(e) Hide all windows, and then set and open the Drawing window.
(f) Report the following to the Drawing window:

```
Quantity ordered:                   dddd
Cost for quantity ordered:  $ ddddddddd.dd
Sales Tax:                    ddddddddd.dd
                            ______________
Total costs:                $ ddddddddd.dd
```

Each small *d* represents a digit position.

19. Write a program that draws in the Drawing window either a cosine function or a sine function, given the following information:
(a) An initial angle greater than or equal to 180 degrees.
(b) A final angle less than or equal to 180 degrees.
(c) The option to draw either a cosine or sine function.
(d) Periodicity factor `M` for drawing `sin( M * Angle)` or `cos(M * Angle)`.
This program requires procedures for each of the following steps:
(a) Hide all windows; then set and open the Text window.
(b) Enter initial and final angles. Test these values to be sure they are within range.
(c) Option to choose a sine or cosine function and the periodicity factor.
(d) Hide all windows; then set and open the Drawing window.
(e) Draw the proper function with x and y axis in the center of the Drawing window.

20. A program is needed for computing the balance and cumulative interest earned for a simple Individual Retirement Account for which the interest rate and yearly deposit are fixed. When executed, this program must enter from prompts in the Text window the amount of the deposit, the interest, and the age of the person holding the account. Assume that deposits to the IRA cannot be made beyond the age of 65, nor can a deposit be greater than $2000. If this information is correct, the program will then display a table in the Drawing window containing a column for the present year of the IRA, the total balance in the IRA, and the cumulative interest earned, ten entries at a time. If additional entries can be placed in the table, the user is prompted either to continue or to exit by clicking the mouse for the appropriate message. Once all entries have been displayed, the user is prompted to begin a new table or to quit the program by again clicking the mouse button for the appropriate message. First design the system, assuming modules for setting and opening windows, prompting for information, drawing the table outline, and displaying table entries.

21. Complete the program called `Tutor_System`. When adding a procedure, test the system to see that it is executing properly according to the given specifications.

22. Redesign Tutor_System so that a student can practice with addition, subtraction, multiplication, and integer division, providing both a

quotient and remainder. Consider using an additional dialog window for selecting one of four options.

23. Write a program that prompts the user to choose the display of one of the following functions to the Drawing window in the range −180 degrees to 180 degrees :

```
sin(X)        tan(X)        cot(X)
cos(X)        sec(X)        csc(X)
```

where `sec(X) = 1/cos(X)`, `csc(X) = 1/sin(X)`, and `cot(X) = 1/tan(X)`. Keep in mind that some of these functions may have undefined values at angles such as −180, −90, 0, 90, or 180 degrees. Figure 7.24 shows an example of such output.

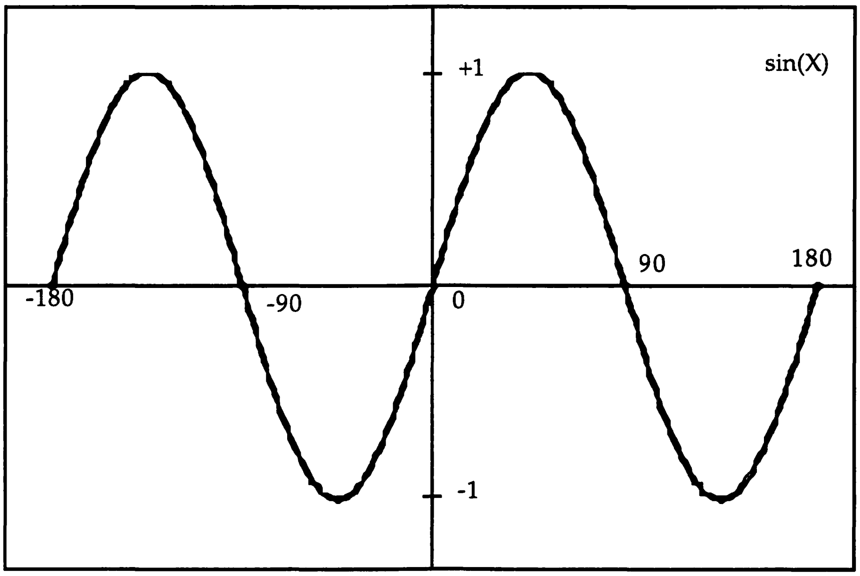


Figure 7.24

Chapter 8

Modularity: Building Programmer-Defined Libraries

OBJECTIVES

After completing Chapter 8, you will know the following:

1. How to apply the **uses** clause within a program unit.
2. How to build programmer-defined units.
3. How to build programmer-defined libraries.
4. What predefined libraries are available in THINK Pascal and how to use them.
5. How to allocate a project as an application.
6. How to use the profiler to study execution characteristics.
7. The basic steps involved in applying the LightsBug debugger.

8.1 UNITS AND LIBRARIES IN THINK PASCAL: THE USES CLAUSE

THINK Pascal allows us to write modular programs by applying units and libraries. When used within a project, a unit is a program component that is a part of a larger application. In the context of a THINK Pascal Project, a unit, as well as a library, is referred to as a *file*. By itself, a unit can be a collection of constants, data types, variables, procedures, and functions. When integrated into a project, a unit can share the definitions of constants, data types, procedures, and functions with other units as well as with a main Pascal program through the **uses** clause. By employing a **uses** clause and having the main program interface with a unit, all the declarations that are in the *public part* of the unit become accessible to the program. Although a unit's definitions are hidden from view, they can be included within the program itself. By itself a unit allows the programmer to hide information as a project is built, and later save it as part of a library.

In THINK Pascal a unit is composed of a unit-header, followed by an interface-part, followed by an implementation-part, and ending with an **end** statement followed by a period. Figure 8.1 shows the format for a unit.

```
unit Unit_Name;

interface

   uses
   { list of units interfacing with Unit_Name }
   { list of constants for public sharing }
   { list of user-defined types for public sharing }
   { list of variables for public sharing }
   { list of procedure and function headings for public }
   {     sharing }
   { in-line body }

implementation

   { list of constant declarations local to this unit }
   { list of user-defined types local to this unit }
   { list of variable declarations local to this unit }
   { list of procedure and function declarations local }
   {     to this unit }
   { list of procedure and function blocks defined in the }
   {     interface-part }

end.
```

Figure 8.1 Format for a THINK Pascal Unit.

The unit-header contains the logical name of the unit. This name must be unique and can never be used as the name of any other identifier in the unit's definition, nor can other files in the Project window define units with the same name. Although it may be convenient for the physical file storing the source code of a unit to have the same name as the unit, THINK Pascal does not require the physical file name to match the unit name. Rules for naming a unit are the same as for any other THINK Pascal identifier.

The interface-part of a unit is composed of the reserved word **interface**, followed by a **uses** clause, a constant-declaration-part, a type-declaration-part, a variable-declaration-part, and a procedure-and-function-header-part. The procedure-and-function-header-part can contain an inline-body declaration for defining routines local to the unit. The **uses** clause allows a unit to borrow from any previously defined unit or units. All of the bodies of procedures and functions are defined within the implementation-part of a unit. As you will see, the interface-part represents the *public part* of a unit's interface because it defines constants, types, variables, procedures, and functions that can be borrowed from the unit.

Following the interface-part of a unit is its implementation-part. The implementation-part is preceded by the reserved word **implementation**, optionally followed by a **uses** clause that can allow only this part of the unit to borrow from previously defined units. Next follows a constant-declaration-part, a type-declaration-part, a variable-declaration-part, and a procedure-and-function-declaration-part. The implementation-part contains procedure and function blocks for the headers of routines defined in the interface-part of a unit. It can also include the full declaration of routines that are local to the unit being defined. The redeclaration of formal parameters given in the headers for public procedures and functions can be omitted in the implementation-part of a unit. The implementation-part is referred to as the *private part* of a unit, because it contains constants, types, variables, procedures, and functions that are local to the unit and cannot be shared with any other programming units. Directives such as forward and external can only appear within the implementation-part, provided that public procedures and functions defined there are not being referenced. Keep in mind that the scope of the declarations given in the interface-part of a unit includes the interface-part as well as the implementation-part. Constants, types, variables, procedures, and functions defined in the interface-part act as global identifiers to the implementation-part of a unit. Remember that the reserved words **interface** and **implementation** are required for separating the interface and implementation parts. Omitting one or both of these reserved words raises the bugs dialog window when the unit is compiled. Like the body of the main Pascal program, a unit ends its definition with the reserved word **end**, *followed by a period, not a semicolon.*

How can a unit be borrowed from a Pascal program or from another unit? Figure 8.2 shows the syntax for borrowing the public definitions defined in the interface-part of a unit. It simply requires the **uses** clause, followed by a list of one or more unit names. These unit names are the logical unit names and not the names of files listed in the Project window or in within folders.

```
program  Program_Name (Input, Output);

   uses
      Unit_Name;

   { list of label declarations }
   { list of constant declarations }
   { list of user-defined types }
   { list of variable declarations }
   { list of procedure and function declarations }

   begin
   { executable body of the main Pascal program }
   end.
```

Figure 8.2 Pascal program using a predefined unit.

It is important to understand that identifiers defined within the public parts of units, and borrowed by either another unit or a Pascal program, cannot be redeclared. Identifiers defined in the interface-part are global to the unit itself and act as external identifiers to any unit or program that borrows from a unit. Any attempt to redeclare identifiers defined in the interface of a unit raises the bugs dialog window, indicating that the declaration of an identifier is being repeated. Identifiers declared in the private parts (implementation-parts) of units are local to the units in which they appear and can be redeclared by other units as well as by the main Pascal program. Declarations for these identifiers are never seen by the Pascal program or unit that is borrowing the unit. Remember that only program units can borrow other units. In THINK Pascal, no syntax rules exist for applying the **uses** clause within the declarations of procedures and functions.

By choosing the option **Build Library...** under the **Project** menu, a programmer can store a unit, or possibly several units, as a collection of precompiled functions and procedures. In short, this option allows the programmer to build a library. In THINK Pascal, a library is defined as a collection of one or more compiled units stored as a file. A library has a definite advantage over a single unit in that copies of its compiled code can be shared among several different projects. The option **Build** from the **Run** menu does not cause routines within a library to be recompiled (since they have already been compiled).

8.2 BUILDING A THINK PASCAL PROJECT CONTAINING A PROGRAMMER-DEFINED UNIT

As an example of a programmer-defined unit, consider the `Text_File_Program` listed in Chapter 7. This program requires four procedures. Three of these are interfaced with the the main Pascal program, and the fourth is a subordinate procedure called `Prompt_For_Data`, which interfaces only with the superordinate procedure `Write_To_Data_File`. This last procedure is treated as a routine local to the program unit in which it is defined.

In modularizing `Text_File_Program`, we define a unit called `Interface_File_Routines`, which has the following public definitions: a variable called `File_Name` and three procedures called `Prompt_For_File_Name`, `Set_And_Show_Text_Window`, and `Write_To_Data_File`. The private part of our unit defines a fourth procedure called `Prompt_For_Data`. This last procedure is only needed as a supporting procedure and is not invoked by the body of the main Pascal program. The following shows the THINK Pascal code for this new unit:

```
unit Interface_File_Routines;
{Purpose :  This unit defines several interface routines that }
{           request a file name from standard input (keyboard); }
{           prompt for the name, tax number, and income of an }
{           individual; and output all of the collected }
{           information to a text file. }

interface
   var
      File_Name: string;
   procedure Prompt_For_File_Name (var File_Name: string);
   procedure Set_And_Show_Text_Window;
   procedure Write_To_Data_File (File_Name: string);

implementation
{ ------------------------------------------------------------ }
```

```
  procedure Prompt_For_Data (var Name, Number: string; var
                                   Income: real);
  { Purpose:  This procedure prompts for and allows the entry }
  {           of a person's full name, tax number, and gross }
  {           income. This procedure is local only to this unit }
  {           and is not shared through the interface }
  begin
  { Prompt for person's name, tax number, and gross income. }
     writeln(' ---------------------------------------------- ');
     write(' Enter the last name of the person: ');
     readln(Name);
     write(' Enter the person`s tax number: ');
     readln(Number);
     write(' Enter the person` s gross income : $ ');
     readln(Income);
  end; { Prompt_For_Data }
{ ------------------------------------------------------------- }
  procedure Prompt_For_File_Name;
  { Purpose:  This procedure prompts for a file name and }
  {           returns that name through the formal parameter }
  {           Name. }
  begin
  { Prompt user for a file name. }
     write(' Enter a file name for storing the data: ');
     readln(File_Name);
  end; { Prompt_For_File_Name }
 { ------------------------------------------------------------ }
  procedure Set_And_Show_Text_Window;
  { Purpose:  This procedure sets the boundary of and opens the }
  {           Text window for viewing. }
     var
        Border: Rect;
  begin
  { Hide all windows before establishing and showing the Text }
  { window. }
     HideAll;
  { Establish the boundaries for displaying the Text window. }
     SetRect(Border, 10, 40, 400, 250);
     SetTextRect(Border);
  { Show the Text window for viewing output.}
     ShowText;
  end; { Set_And_Show_Text_Window }
{ ------------------------------------------------------------- }
  procedure Write_To_Data_File;
  { Purpose:  This procedure opens a file for writing data, }
  {           and then writes data to an output file }
  {           represented by the variable parameter Output_File.}
     var
        Output_File: Text;
        Name, Tax_Number: string;
        Tab, Response: char;
```

```
         Gross_Income: real;
   begin
   { Initialize the value for a horizontal tab. }
      Tab := chr(9);
   { Open the output file for writing data. }
      rewrite(Output_File, File_Name);
   { Write column headers to the output file.}
      writeln(Output_File, 'Name', Tab, 'Tax Number', Tab, 'Gross
               Income');
   { Continue to prompt for data until the user is ready to quit.}
      repeat
      { Prompt for and enter name, tax number and gross income. }
         Prompt_For_Data(Name, Tax_Number, Gross_Income);
      { Write these three entries to the data file. }
         write(Output_File, Name, Tab, Tax_Number, Tab);
         writeln(Output_File, Gross_Income : 6 : 2);
      { Prompt the user to either continue or quit. }
         write(' Press `Q` to quit, `C` to continue: ');
         readln(Response);
      until ((Response = 'Q') or (Response = 'q'));
   { Close the output file before exiting this program. }
      close(Output_File);
   end; { Write_To_Data_File }
end.
```

The modified main Pascal program now requires only a **uses** clause and its own executable body. No other declarations are required, because all variables and procedures are borrowed through the interface with the unit Interface_File_Routines. Following is the code for the main Pascal program, Text_File_Program:

```
program Text_File_Program (Input, Output);
{Purpose :  This program creates a text file with three columns }
{           and several rows of information. The first column }
{           lists names, the second, tax numbers, and the third, }
{           gross income. Once the file is closed, it can }
{           be opened by a spreadsheet application such as Excel.}
   uses
      Interface_File_Routines;
{ ================ Body of the main program. ================ }
begin
{ Hide all windows and then open the Text window for viewing }
{ prompts and responses. }
   Set_And_Show_Text_Window;
{ Prompt user for a file name. }
   Prompt_For_File_Name(File_Name);
{ Now write data to the output file represented by File_Name.}
   Write_To_Data_File(File_Name);
end.
```

The steps for building a project having one or more programmer-defined units and a main Pascal program are outlined below.

1. Assuming that THINK Pascal is presently the active desktop application, follow the steps for creating a new project or for opening an existing project. If the existing project has units and libraries that are not needed, highlight the name of the unneeded file in the Project window and choose the command option **Remove** from the **Project** menu. You may have to repeat this step if several such files require removal from the current Project window. For now, however, assume that we are building a new project.

   When building a new project, choosing the command option **New Project...** from the **Project** menu allows you to specify a new project name. Clicking **New Project...** opens the dialog window shown in Figure 8.3. Notice that you have a choice of two methods for the creation of a project.

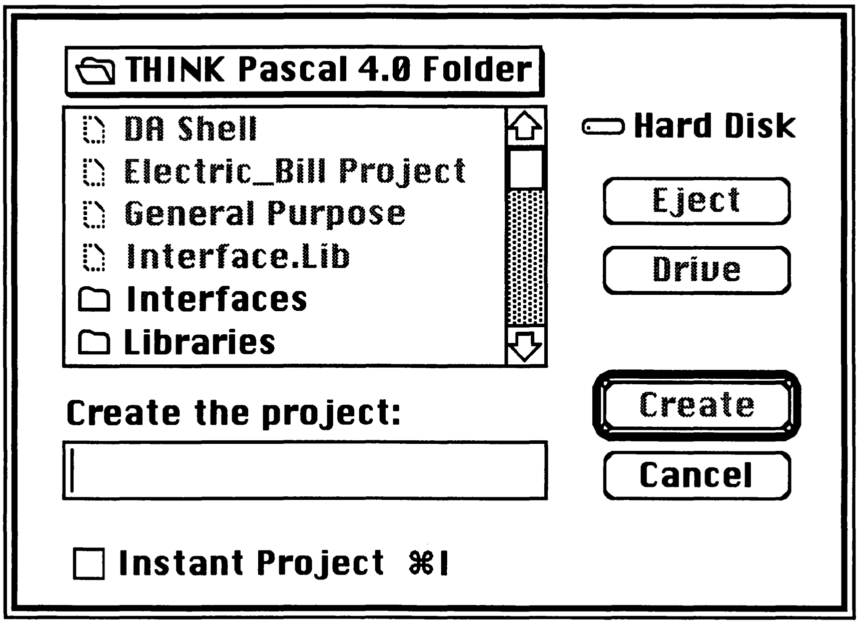
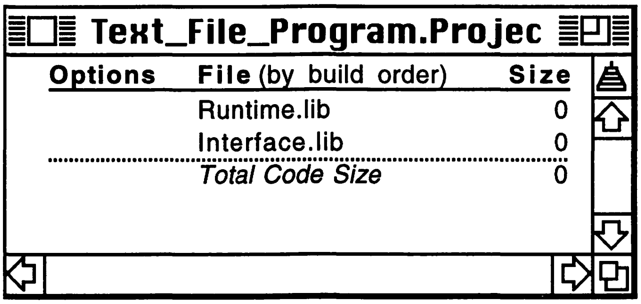


Figure 8.3 The dialog box for creating a new project.

   The first method is to enter a project name in the field provided and then click the **Create** button. This sequence results in the creation of a new project with the name you entered. Then the dialog window is closed, and a Project window is displayed. Figure 8.4 illustrates this outcome with the Project window for the present example. The second method is to click the **Instant Project** button, enter the name of the project as requested, and then click the **Create** button. Then an Edit window is displayed that has a program header and a basic shell representing a Pascal program. This shell is similar to the outline displayed by Macintosh Pascal when the option to create a new program is selected. The **Instant Project** approach creates three new files: a source file having the extension .p, a project file having the extension .π, and a folder holding both the source and project files.


Text_File_Program.Projec

Options	File (by build order)	Size
	Runtime.lib	0
	Interface.lib	0
	Total Code Size	0

Figure 8.4 The Project window for the file `Text_File_Program.Project`.

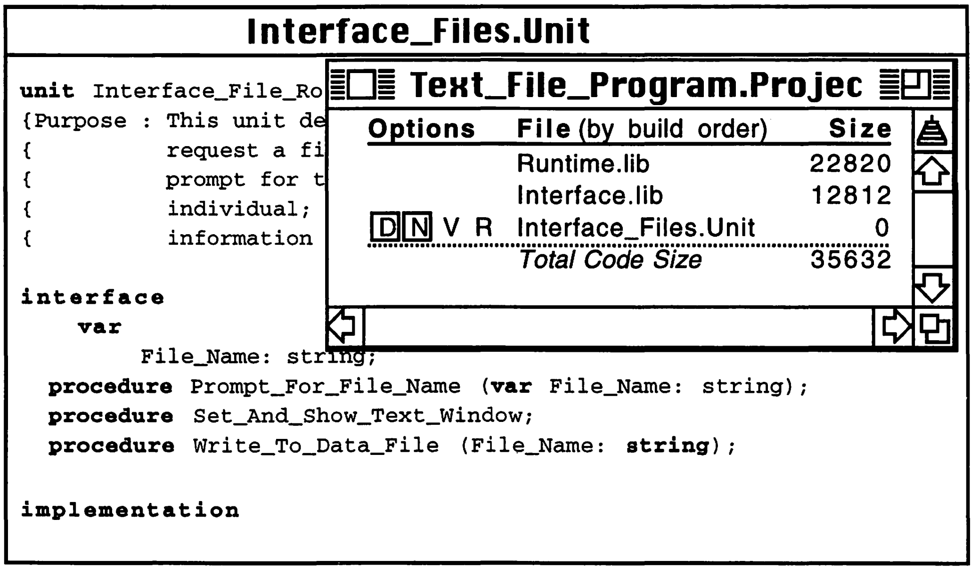
Notice that in the present example the project is named `Text_File_Program` with the extension `.Project` appended to distinguish this file as a project file. Although the *THINK Pascal User Manual* suggests the use of the extension `.p` for Pascal source files and the extension .π for project files, the choice of extensions for files is left to the programmer. THINK Pascal understands various types of files from the properties that each file has when it is created.

When a new project is created, the Project window includes two special libraries, `Runtime.lib` and `Interface.lib`. Both of these have an initial size of zero. The file `Runtime.lib` contains standard predefined routines for input and output, and the file `Interface.lib` contains code for borrowing many of the units in the Macintosh Toolbox. These two libraries can be compiled at this time by selecting the command option **Build** from the **Run** menu.

2. You are now ready to insert a new program unit into an empty Edit window. To begin, select the command option **New** from the **File** menu to open a new Edit window. When convenient, you can choose the option **Check Syntax**, located under the the **Run** menu, to check the syntax of the Pascal code in the newly defined unit. At this stage you must remove all syntax errors before the unit can be either built or compiled. Note that pressing the shift key when selecting from the **Run** menu gives you the option to compile what is in an Edit window rather than simply checking for syntax errors.

   When you have corrected all of the syntax errors in the source code of a unit, use the command option **Save As...** (or **Save**) from the **File** menu to save the program unit to disk as a Pascal source file. In our example a copy of the source file for our newly defined unit `Interface_File_Routines` is saved as `Interface_Files.Unit`. The extension "`.Unit`" is used to distinguish this Pascal source file from an ordinary Pascal program file or from a project file.

3. Once all syntax errors have been removed and the source file has been saved, you are free to add the desktop file of the current Edit window to the Project window. This is done by choosing the option **Add "Filename"** from the **Project** menu. The actual name you assigned the source file for the unit will appear in the menu in place of **"Filename"**. Figure 8.5 shows the Project window with the new unit `Interface_File_Routines`, stored as the file `Interface_Files.Unit`, added to the Program window.



```
unit Interface_File_Ro
{Purpose : This unit de
{          request a fi
{          prompt for t
{          individual;
{          information

interface
   var
      File_Name: string;
  procedure Prompt_For_File_Name (var File_Name: string);
  procedure Set_And_Show_Text_Window;
  procedure Write_To_Data_File (File_Name: string);

implementation
```

```
Options     File (by build order)     Size
            Runtime.lib               22820
            Interface.lib             12812
D N V R     Interface_Files.Unit          0
            Total Code Size           35632
```

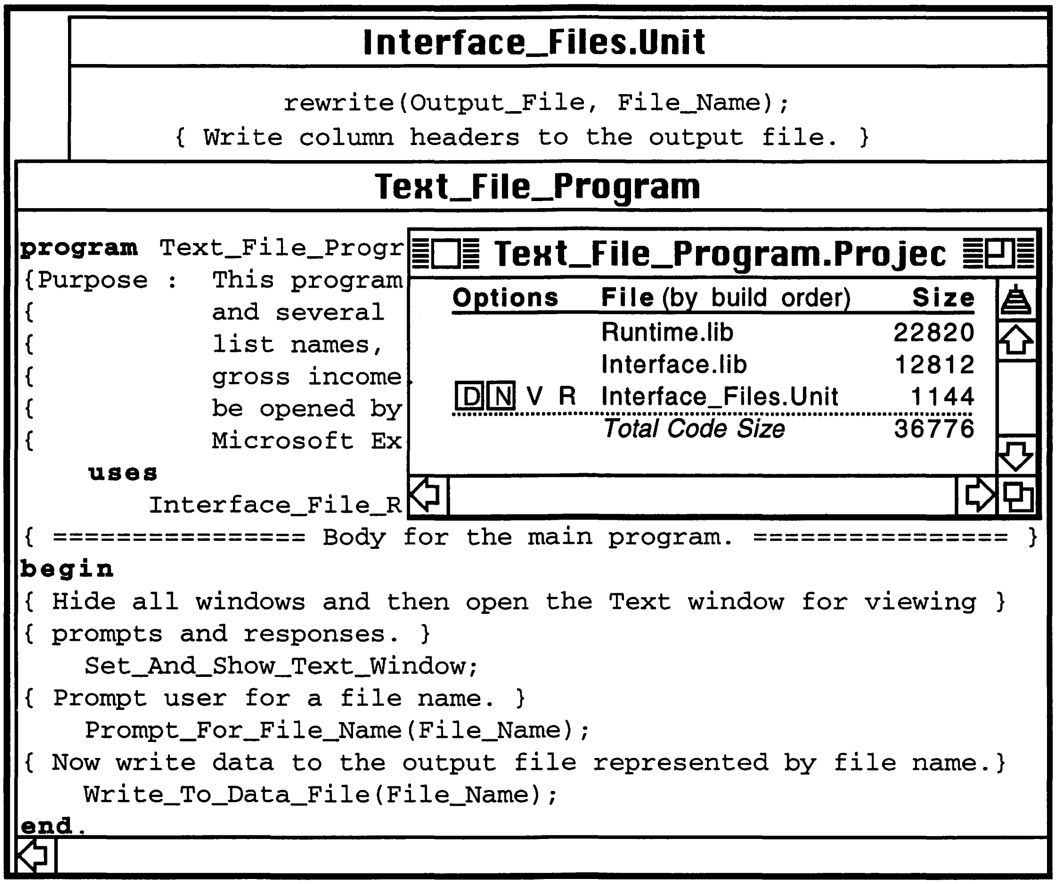
Figure 8.5 The Project window, showing the insertion of file `Interface_Files.Unit`.

4. Notice that the program unit has zero size when it is first added to the Project window. The program unit can now be compiled. Select the option **Build** (or **Compile**) from the **Run** menu, and the source code for this new unit will be built (compiled) and stored within the file for the project. Figure 8.6 shows the Project window after the program unit and the libraries have been built.

   An alternative step in adding a new unit to the Project window is to use the option **Add File...** from the **Project** menu. This command option displays a dialog window where a file name is selected prior to clicking the **Add** button. A copy of the source code of the selected file is then added to the Project window. The dialog window can only be closed by clicking the **Done** button.

5. Select the command option **New** from the **File** menu. It is now time to insert the main Pascal program into an empty Edit window. Remember to periodically select the option **Check Syntax** from the **Run** menu to check the syntax of the Pascal code in the newly defined Pascal program. Once you have entered the main program and corrected all syntax errors, use the option **Save As...** from the **File** menu to save

a copy of the Pascal source program. Figure 8.6 shows a portion of the Edit window with another inactive Edit window in the background. The active window shown is the Project window. As you can see from the list of files in the Project window, the program source file `Text_File_Program` has not yet been added to the project.



Interface_Files.Unit

```
          rewrite(Output_File, File_Name);
    { Write column headers to the output file. }
```

Text_File_Program

```
program Text_File_Progr
{Purpose :  This program
{           and several
{           list names,
{           gross income
{           be opened by
{           Microsoft Ex
    uses
        Interface_File_R
{ =============== Body for the main program. =============== }
begin
{ Hide all windows and then open the Text window for viewing }
{ prompts and responses. }
    Set_And_Show_Text_Window;
{ Prompt user for a file name. }
    Prompt_For_File_Name(File_Name);
{ Now write data to the output file represented by file name.}
    Write_To_Data_File(File_Name);
end.
```

Text_File_Program.Projec

```
Options     File (by build order)      Size
            Runtime.lib               22820
            Interface.lib             12812
[D][N] V R  Interface_Files.Unit       1144
            Total Code Size           36776
```

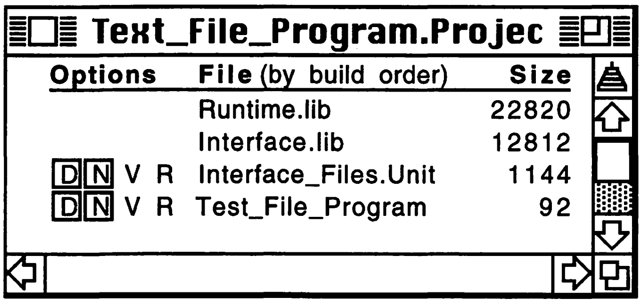
Figure 8.6 The Project window wiui two inactive Edit windows in the background. The file `Interface_Files.Unit` has now been built.

Remember that when the syntax of either a Pascal program or a program unit is checked, any program units being borrowed must have already been built (or compiled). If this is not the case, a bugs window will appear, indicating that the unit referenced by the program unit being checked either has no file in the Project window or the file in the Project window has not yet been compiled. All units upon which a Pascal program or program unit depends must be built before further units are defined.

It is also important that units in the Project window be listed in order of dependency. For example, if unit Z is a primary unit having no dependency upon other programmer-defined units, it must be built first.

If unit X uses only unit Z, it can be built second. A Pascal program using unit X can then be defined and, since the previous dependent units have already been built, it can be compiled or built. In the Project window, the names of files for the two program units appear first, followed by the file for the Pascal program. Toward the top of the list is the file for unit Z, followed by the file for unit X, and at the bottom is the file for the Pascal program. Note that you can change the position of a file name in the Project window by highlighting the file that is to be moved and using the mouse cursor, represented by a small hand, to move the file to another position. A file can be moved upward or downward.

6. You are now ready to add the main Pascal program to the Project window. Make the Edit window containing the Pascal program active (click somewhere within the Edit window), and then choose the option **Add "Filename"** from the **Project** menu. This adds the file name for the Pascal program to the Project window, but the compile size shows as zero. Select the command option **Build** (or **Compile**) from the **Run** menu to compile the program. The compiled code for the Pascal program will reside within the project file. Figure 8.7 shows the Project window after all of the files have been built.


Text_File_Program.Projec

Options	File (by build order)	Size
	Runtime.lib	22820
	Interface.lib	12812
D N V R	Interface_Files.Unit	1144
D N V R	Test_File_Program	92

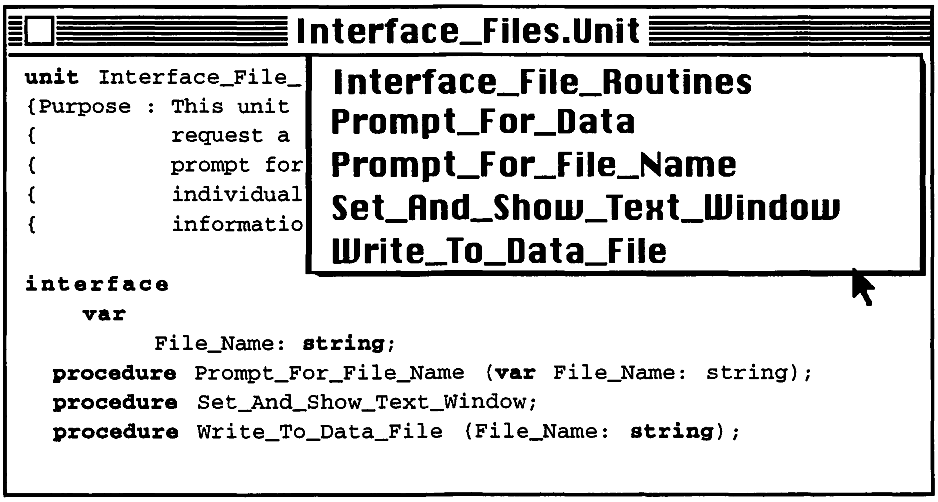
Figure 8.7 The Project window after all of the required files have been built.

Note that THINK Pascal has the ability to "remake" files that depend on units where syntax changes are made. For example, if some lines of code in unit Z are changed, selecting option **Build** from menu **Run** for unit Z causes both unit Z and unit X to be rebuilt, because unit X depends upon unit Z. This effect does not occur if the option **Compile** is selected.

7. At this point you can execute the Pascal program by choosing the option **Go** from the **Run** menu.

Choosing the option **Views Options...** in the **Project** menu displays additional choices for observing information on files listed in the Project window. This can include the file name storing the source; special run-time options for debugging, as well as integer arithmetic and range-checking; the size of the unit or program; the unit name as it relates to the identifier given in the program or unit header, but only after a file has been compiled; the volume name of the folder where the source file is stored; and the date and time when the source file within project was last saved.

When working with the Edit window of a program unit containing several routines, you can locate the beginning of the source code of any routine by holding down the Apple (command) key, clicking the title bar of the program unit, and after the pop-up window appears, selecting the name of the routine to be viewed. Figure 8.8 shows what occurs for the Edit window containing the file `Interface_Files.Unit`. The beginning of procedure `Prompt_For_File_Name` is displayed if you highlight the procedure name and release the mouse button.



```
unit Interface_File_
{Purpose : This unit
{          request a
{          prompt for
{          individual
{          informatio

interface
   var
         File_Name: string;
  procedure Prompt_For_File_Name (var File_Name: string);
  procedure Set_And_Show_Text_Window;
  procedure Write_To_Data_File (File_Name: string);
```

Figure 8.8 Viewing a list of routines within a program unit after pressing the Apple Key and clicking the title bar.

There is one additional point to consider before ending this section. If you need to examine the source file of any program unit listed in the Project window, just highlight the name of the file containing the unit while in the Project window, and double-click the mouse button. This will open the Edit window if the file is closed, or make the Edit window of the source file the active window if the file is open. An alternative for opening the source file of any unit i to choose the option **Open...** from the **File** menu. Use whichever approach you feel is convenient.

8.3 BUILDING THINK PASCAL LIBRARIES

A library can be either a single compiled unit or a collection of compiled units stored as a special file and built by choosing the option **Build Library...** from the **Project** menu. A library can be added to a project by choosing the command option **Add File...** from the **Project** menu before any of the units in the Project window are built. In building a library it is important that you also define an interface file that is separate from the library file. An interface file allows the specification of constants, types, variables, as well as function- and procedure-headers that can be borrowed by other units. It is important to understand that while the interface unit allows another unit to see what can be borrowed, the library contains the compiled code for borrowed functions and procedures.

As an example, we will build a library for performing four basic complex arithmetic operations: addition, subtraction, multiplication, and division. As a brief introduction, we can represent a complex number R in rectangular form by the definition

```
R = A + j B
```

where A represents the real part of the complex number R, B the imaginary part of the complex number R, and j the square root of the value (-1). For two complex numbers defined as

```
R = A + j B

S = C + j D
```

the following is a list of rules for performing four complex arithmetic operations:

```
R + S =  ( A + C ) + j ( B + D )
R - S =  ( A - C ) + j ( B - D )
R * S =  [ ( A * C ) - ( B * D ) ] + j [ ( B * C ) + ( A * D ) ]
R / S =  [ ( A + j B ) * ( C - j D ) ]  / ( C² + D² )
```

When storing complex values, we will define a Pascal record structure for representing a complex number. Pascal record types are discussed in detail in Chapter 9. The structure representing a complex number as a type is composed of two parts: a field labeled Real_Part representing the real part of a complex number, and a field labeled Imag_Part representing the imaginary part of the number. The following defines our complex number as a record type called Complex_Rect:

```
type
   Complex_Rect = record
                    Real_Part : real;
                    Imag_Part : real;
                  end;
```

The following unit is an interface unit specifying our record type Complex_Rect as well as the definition of four functions for performing basic complex arithmetic:

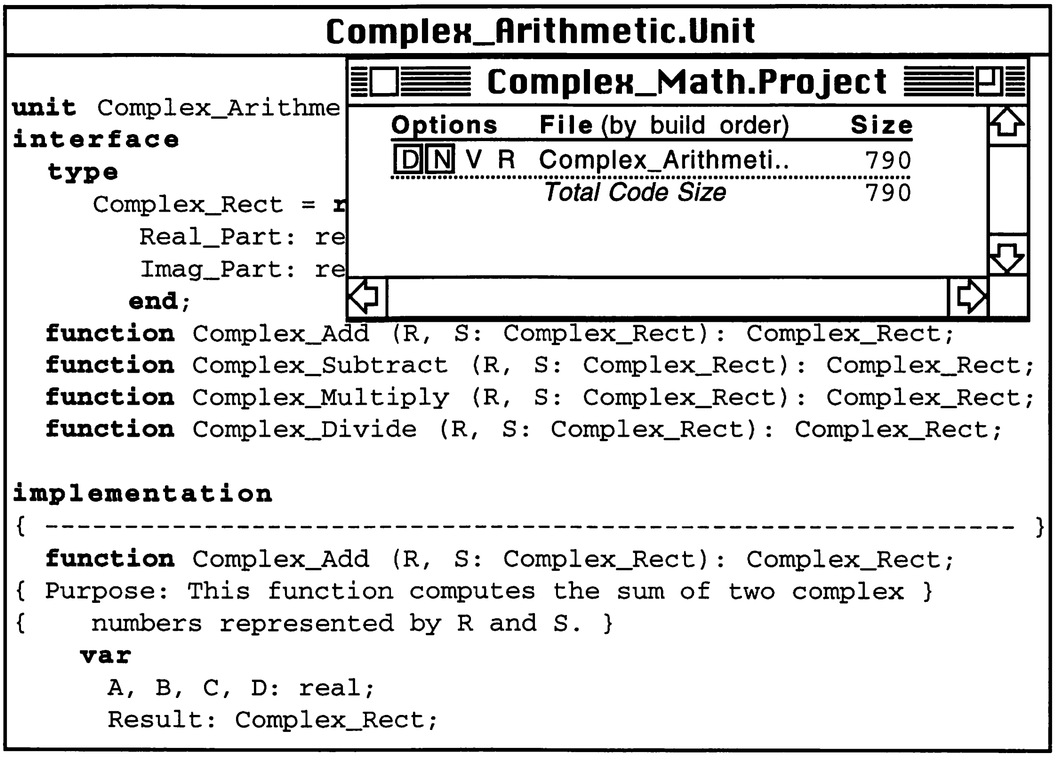
```
unit Complex_Data_Interface;
{ Purpose:  This unit defines the structure of a complex number }
{           stored in rectangular form and lists four }
{           functions for performing complex arithmetic. }

interface
type
   Complex_Rect = record
                    Real_Part: real;
                    Imag_Part: real;
                  end;
function Complex_Add( R, S : Complex_Rect ) : Complex_Rect;
function Complex_Subtract( R, S : Complex_Rect ) : Complex_Rect;
function Complex_Multiply( R, S : Complex_Rect ) : Complex_Rect;
function Complex_Divide( R, S : Complex_Rect ) : Complex_Rect;

implementation
{ The declarations that follow are optional. They are given to }
{ show that the body for each function is externally defined in }
{ another unit. }
function Complex_Add(R, S : Complex_Rect) : Complex_Rect;
   external;
function Complex_Subtract(R, S : Complex_Rect) : Complex_Rect;
   external;
function Complex_Multiply(R, S : Complex_Rect) : Complex_Rect;
   external;
function Complex_Divide(R, S : Complex_Rect) : Complex_Rect;
   external;
end.
```

This interface unit is important because it discloses the type that represents a complex number and the functions that can be borrowed from this unit. Notice that all four functions are declared as being external. This means that the executable bodies for the four functions (Complex_Add, Complex_Subtract, Complex_Multiply, and Complex_Divide) are stored in a separate file created as a library. The interface file is created first. Open a new Edit window, and enter the text of the interface unit. Check the syntax of the unit, and eliminate any syntax errors. Then you can save the source file using the **Save** or **Save As...** option from the **File** menu. At this point it is not necessary to add this file to the Project window, because we are concerned with building a library unit. *You must remember that the interface file is never a part of the library.* The purpose of the interface unit is to specify identifiers that can be borrowed by other units using the library. For now, the source file for our interface unit, called Complex_Data_Interface, will be referred to as Complex_Interface.Unit.

The next step is to define the unit (or units) representing the source code for the library. This file repeats the declarations of constants, types, variables, functions, and procedures defined in the interface file but with the bodies of all routines defined in the implementation-part of the unit. For our example, the unit called Complex_Arithmetic defines each of the four functions. Notice that the declaration for type Complex_Rect is repeated. This is necessary because the type Complex_Rect is being used in this unit as well as in the interface unit.

```
unit Complex_Arithmetic;
{ Purpose:  This unit represents the source code for the library }
{           Complex_Math.lib. }

interface
type
   Complex_Rect = record
                     Real_Part: real;
                     Imag_Part: real;
                  end;

function Complex_Add (R, S: Complex_Rect): Complex_Rect;
function Complex_Subtract (R, S: Complex_Rect): Complex_Rect;
function Complex_Multiply (R, S: Complex_Rect): Complex_Rect;
function Complex_Divide (R, S: Complex_Rect): Complex_Rect;

implementation
{ ---------------------------------------------------------------- }

function Complex_Add (R, S: Complex_Rect): Complex_Rect;
{ Purpose:  This function computes the sum of two complex }
{           numbers represented by R and S. }
   var
      A, B, C, D: real;
      Result: Complex_Rect;
   begin
   { Initialize the values of variables A, B, C, and D. }
      A := R.Real_Part;
      B := R.Imag_Part;
      C := S.Real_Part;
      D := S.Imag_Part;
   { Compute the complex sum of R and S. }
      Result.Real_Part := (A + C);
      Result.Imag_Part := (B + D);
   { Return the value of Result through the function name. }
      Complex_Add := Result;
   end; { Complex_Add }
{ ---------------------------------------------------------------- }

function Complex_Subtract (R, S: Complex_Rect): Complex_Rect;
{ Purpose:  This function computes the difference of two complex }
{           numbers represented by R and S. }
   var
      A, B, C, D: real;
      Result: Complex_Rect;
   begin
   { Initialize the values of variables A, B, C, and D. }
      A := R.Real_Part;
      B := R.Imag_Part;
      C := S.Real_Part;
      D := S.Imag_Part;
   { Compute the complex difference of R and S. }
      Result.Real_Part := (A - C);
      Result.Imag_Part := (B - D);
   { Return the value of Result through the function name. }
      Complex_Subtract := Result;
   end; { Complex_Subtract }
{ ---------------------------------------------------------------- }

function Complex_Multiply (R, S: Complex_Rect): Complex_Rect;
{ Purpose:  This function computes the product of two complex }
{           numbers  represented by R and S. }
   var
      A, B, C, D: real;
      Result: Complex_Rect;
   begin
   { Initialize the values of variables A, B, C, and D. }
      A := R.Real_Part;
      B := R.Imag_Part;
      C := S.Real_Part;
      D := S.Imag_Part;
   { Compute the complex product of R and S. }
      Result.Real_Part := (A * C) - (B * D);
      Result.Imag_Part := (A * D) + (C * B);
   { Return the value of Result through the function name. }
      Complex_Multiply := Result;
   end; { Complex_Multiply }
{ ------------------------------------------------------------ }

function Complex_Divide (R, S: Complex_Rect): Complex_Rect;
{ Purpose:  This function computes the quotient of two complex }
{           numbers represented by R and S. }
   var
      A, B, C, D, Temp: real;
      Result: Complex_Rect;
   begin
   { Initialize the values of variables A, B, C, and D. }
      A := R.Real_Part;
      B := R.Imag_Part;
      C := S.Real_Part;
      D := S.Imag_Part;
   { Compute the complex quotient of R divided by S. }
      Temp := sqr(C) + sqr(D);
      Result.Real_Part := ((A * C) + (B * D)) / Temp;
      Result.Imag_Part := ((C * B) - (A * D)) / Temp;
   { Return the value of Result through the function name. }
      Complex_Divide := Result;
   end; { Complex_Divide }
end.
```

At this point you should insert the above code into an Edit window and, after checking the syntax to remove all syntax errors, add the file to the Project window. Be sure that a copy of the source is kept, because you may later need to edit the library if functions need to be added or changed. In our example, the source for our library is stored in a file called `Complex_Arithmetic.Unit`. After adding this file to the Project window, apply the command option **Build** from the **Run** menu in order to build the compiled code. Figure 8.9 shows the Project window, from which the files `Runtime.lib` and `Interface.lib` have been removed. These files will be added to the Project window when the main Pascal program is built.


Complex_Arithmetic.Unit
Complex_Math.Project
Options File (by build order) Size
D N V R Complex_Arithmeti.. 790
Total Code Size 790

```
unit Complex_Arithme
interface
  type
     Complex_Rect = r
        Real_Part: re
        Imag_Part: re
       end;
  function Complex_Add (R, S: Complex_Rect): Complex_Rect;
  function Complex_Subtract (R, S: Complex_Rect): Complex_Rect;
  function Complex_Multiply (R, S: Complex_Rect): Complex_Rect;
  function Complex_Divide (R, S: Complex_Rect): Complex_Rect;

implementation
{ -------------------------------------------------------------- }
  function Complex_Add (R, S: Complex_Rect): Complex_Rect;
{ Purpose: This function computes the sum of two complex }
{    numbers represented by R and S. }
    var
      A, B, C, D: real;
      Result: Complex_Rect;
```

Figure 8.9 The Project window `Complex_Math.Project` without the libraries `Runtime.lib` and `Interface.lib`.

In the next step we will build the library unit, using the command option **Build Library...** from the **Project** menu. Figure 8.10 shows the dialog window that appears when you choose this option. Note that we have used the name `Complex_ Math.lib` for storing the compiled machine code of this unit.[1] Click the **Save** button in the dialog window to create this library file.

In the next step we add our library to a new project. Figure 8.11 shows a Project window where the source file titled `Complex_Interface.Unit` and the library file titled `Complex_Math.lib` have been added to a project by using the option **Add Files...** from the **Project** menu. This is done by highlighting the file `Complex_ Interfaces.Unit` and clicking the **Add** button, and then repeating this step for the file `Complex_Math.lib`. Once all the files are included in the Project window, it must be built (or rebuilt), using the command option **Build** from the **Run** menu.

[1] The extension .lib is used to distinguish a library from a project or a unit. This extension, like others, is optional, because THINK Pascal automatically stores information on the type of file it has created.

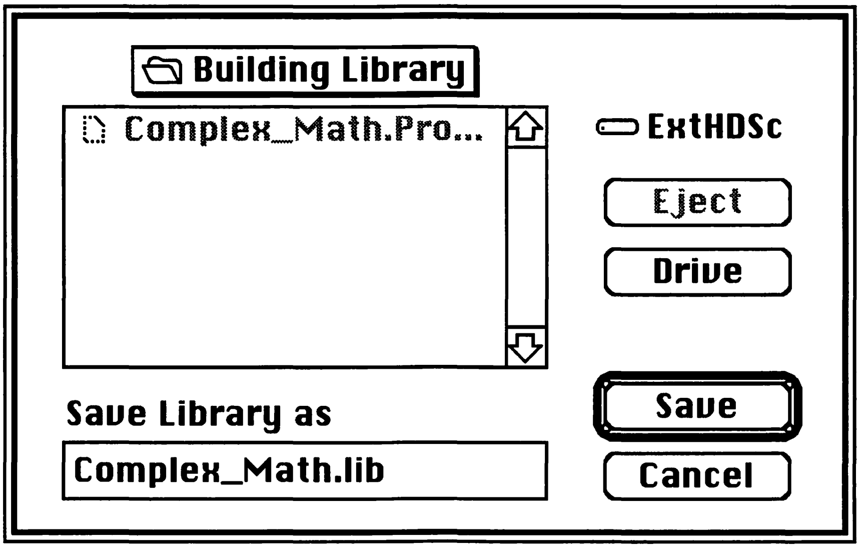
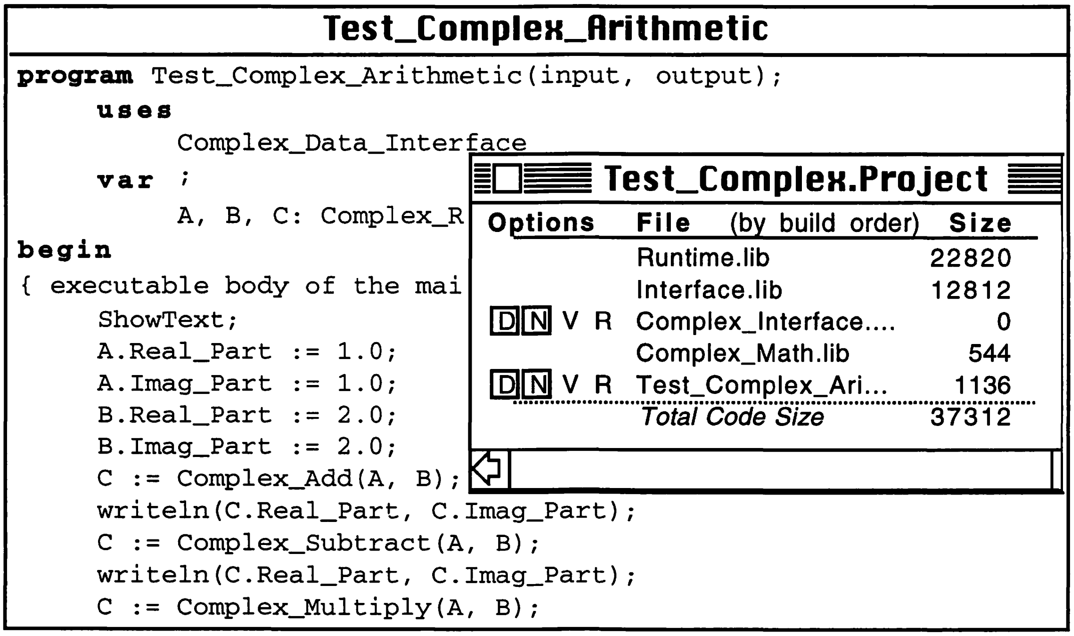


Figure 8.10 The dialog window for building or replacing a library.

Figure 8.11 shows a portion of the main Pascal program for testing the routines in



```
program Test_Complex_Arithmetic(input, output);
    uses
        Complex_Data_Interface
    var ;
        A, B, C: Complex_R
begin
{ executable body of the mai
    ShowText;
    A.Real_Part := 1.0;
    A.Imag_Part := 1.0;
    B.Real_Part := 2.0;
    B.Imag_Part := 2.0;
    C := Complex_Add(A, B);
    writeln(C.Real_Part, C.Imag_Part);
    C := Complex_Subtract(A, B);
    writeln(C.Real_Part, C.Imag_Part);
    C := Complex_Multiply(A, B);
```

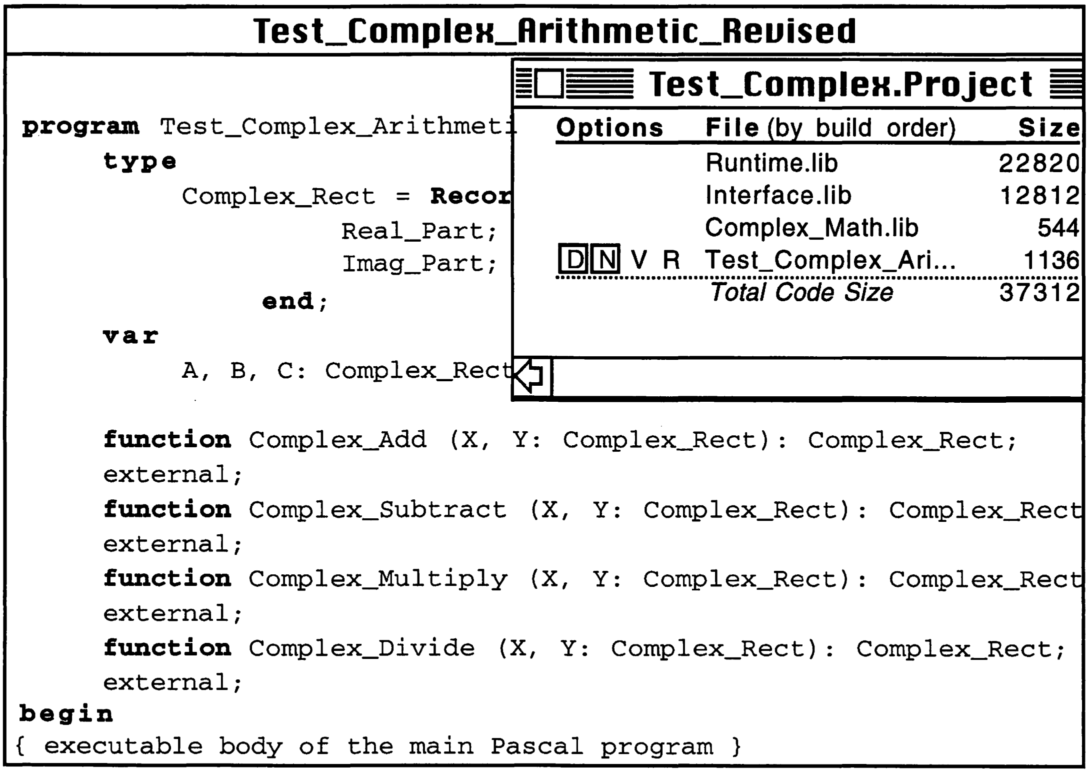
Options	File (by build order)	Size
	Runtime.lib	22820
	Interface.lib	12812
D N V R	Complex_Interface....	0
	Complex_Math.lib	544
D N V R	Test_Complex_Ari...	1136
	Total Code Size	37312

Figure 8.11 The Project window with the complex mathematical library `Complex_Math.lib` added to the project `Test_Complex.Project`.

the complex math library. Here the **uses** clause has the argument `Complex_Data_Interface`. Through this statement the Pascal program understands the type

Complex_Rect and external functions that it needs to borrow from Complex_ Data_Interface. The routines in the library Complex_Math.lib contain the actual machine code for execution.

Is it necessary to use an interface unit such as Complex_Data_Interface in our main Pascal program? Actually no, because we can explicitly declare the functions required by our test program by using the directive external. Figure 8.12 shows the declaration of function-headers for our complex mathematics library with the interface unit excluded. Notice that we must include the declaration for the type Complex_Rect, because this information is not kept by the compiled file Complex_Math.lib when the file is linked with our Pascal program. In our previous example we borrowed this type from unit Complex_Data_Interface by applying the **uses** clause. The external directive informs the compiler that the executable body of a routine is not within the program unit that is being compiled. Rather, the executable code will be found when a special program called the *linker* attempts to bind the call of a routine (where each complex function is invoked in our program) with the location in a library where the routine begins execution.


Test_Complex_Arithmetic_Revised

```
program Test_Complex_Arithmeti
   type
      Complex_Rect = Recor
                  Real_Part;
                  Imag_Part;
               end;
   var
      A, B, C: Complex_Rec

   function Complex_Add (X, Y: Complex_Rect): Complex_Rect;
   external;
   function Complex_Subtract (X, Y: Complex_Rect): Complex_Rect
   external;
   function Complex_Multiply (X, Y: Complex_Rect): Complex_Rect
   external;
   function Complex_Divide (X, Y: Complex_Rect): Complex_Rect;
   external;
begin
{ executable body of the main Pascal program }
```

Test_Complex.Project

Options	File (by build order)	Size
	Runtime.lib	22820
	Interface.lib	12812
	Complex_Math.lib	544
D N V R	Test_Complex_Ari...	1136
	Total Code Size	37312

Figure 8.12 Without the interface file included in the Project window, the external directive is required in declaring the four complex functions to be external routines.

Another issue arises with multiple declarations. It is important to avoid having the compiler generate multiple storage space when writing interface files that declare constants and variables within the interface part of a unit. This requires using an external variable directive of the form {$J+} to turn on the option not to allocate storage for constants and variables, and the directive {$J-} to turn off the option when storage is to be allocated. Understand that explicit programmer-defined types do not take up storage during the execution of a Pascal program. Programmer-defined types only allow the compiler to understand the properties of identifiers that represent a data type. Following is an example that uses this compile directive.

```
unit  Sample_Interface;
interface
const
      {$J+}
      Local_Tax_Rate = 0.02;
      State_Tax_Rate = 0.04;
var
      Local_Tax : real;
      State_Tax : real;
      Total_Tax : real;
      {$J-}

implementation
. . .
```

8.4 USING THE **uses** CLAUSE WITHIN THE IMPLEMENTATION SECTION OF A UNIT

As the previous sections have shown, the **uses** clause defines the dependencies that a program or unit can have on other files that compose a part of the entire project. A **uses** clause can appear within the interface portion as well as within the implementation portion of a unit. If the **uses** clause is within the interface portion of a particular unit, definitions in other units can be propagated to another unit or program that borrows from this particular unit. Everything that a particular unit sees can be seen by any program or unit that borrows from the unit. In turn, if the **uses** clause is within the implementation section of a unit, then the interface-part cannot see the units that are listed, and those units cannot be propagated.

If you choose the command option **Compiler Options...** in the **Project** menu and click on the **USES Extensions** box in the resulting dialog window, THINK Pascal allows you to apply the **uses** clause in two different ways. First, it allows units listed in the **uses** clause of the interface-part of a unit to be propagated. For example, consider the followed unit, titled `Propagate_Test`:

```
unit Propagate_Test;
interface
uses
   Unit_A, Unit_B, Unit_C, Unit_D;
implementation
end.
```

Propagate_Test depends upon four units, given as Unit_A, Unit_B, Unit_C, and Unit_D. A type called Test_Type is defined within Unit_D's interface part and is used to define a new type in the unit that follows:

```
unit Check_Propagation;
interface
uses
   Propagate_Test;
type
   New_Data_Type = Test_Type;
implementation
end.
```

Although the **uses** clause of Check_Propagation does not list Unit_A, Unit_B, Unit_C, and Unit_D, the public parts defined in these units are propagated to this unit through Propagate_Test. Any unit that Propagate_Test can borrow can be borrowed by Check_Propagation. Any program or unit that uses Check_Propagation also borrows all the information defined in the public part of Check_Propagation as well information defined in the public parts of Propagate_Test, Unit_A, Unit_B, Unit_C, and Unit_D.

Second, setting the **USES Extensions** option allows the **uses** clause to appear in the implementation portion of a unit. This allows a unit to borrow only the information that it needs and to hide information that it has borrowed from the interface portion of the unit. It prevents a unit from propagating information that it borrows as well as new information that it creates from definitions and declarations within the implementation part of a unit. The following shows the modification of the content of Propagate_Test and Check_Propagation. Notice that the information from Unit_D is only used in the implementation part of Check_Propagation and is not promulgated by Propagate_Test:

```
unit Propagate_Test;
interface
uses
   Unit_A, Unit_B, Unit_C;
implementation
end.
```

```
unit Check_Propagation;
interface
uses
   Propagate_Test;
implementation
uses
   Unit_D;
type
   New_Data_Type = Test_Type;
end.
```

By including a **uses** clause in the implementation portion of Check_ Propagation, information such as New_Data_Type is hidden from any unit that uses Check_Propagation. Remember that the propagation of units and the application of a **uses** clause in the implementation portion of a unit are possible only when the **USES Extension** is turned on.

Without **USES Extension** turned on, the following Pascal program will fail to compile:

```
program Main ( Input, Output );
   uses
      Unit_A;
begin
   writeln( X, Z );
end.
```

```
unit Unit_A;
interface
   uses
      Unit_B;
   const
      X = Z ;
implementation
end.
```

```
unit Unit_B;
interface
   const
      Z = 345;
implementation
end.
```

The program Main fails to compile even though Unit_A and Unit_B are built because constant X in Unit_A is equated with constant Z in Unit_B. Unit_A sees constant Z because Unit_A borrows from Unit_B, but the program Main borrows only from Unit_A and understands only the declaration for constant X. It does not understand that X is equated with Z, because it cannot see the declaration for Z. How can this problem be rectified? First, the main program must include a reference to Unit_B in the **uses** clause. This reference must precede Unit_A, because constant X is being replaced with constant Z. Otherwise the same compile error appears, because the translator needs to know where identifier Z is located. The translator is looking at the symbol table in Unit_A and at this point would not see a proper reference to Z as it builds a symbol table for program Main. Even if constants X and Z are not accessed by the body of program Main, this problem still exists if program Main borrows from Unit_A. The **uses** clause in program Main requires Unit_B to precede Unit_A, so that the symbol table being created for program Main will understand the public part of Unit_B before seeing any reference to constant Z. This is not the case if **Uses Extension** is set. When this option is set, Unit_B is propagated to program Main through Unit_A, and only Unit_A needs to be declared in the **uses** clause of Main. Program Main will properly see all the symbol tables from all units that are propagated to it.

8.5 PREDEFINED LIBRARIES

THINK Pascal supports all of the standard Pascal procedures and functions as well as most of the routines from the common Macintosh Toolbox libraries. When a project window is created, standard predefined procedures and functions from the library file `Runtime.lib` are inserted. `Runtime.lib` includes standard IO routines such as `read`, `readln`, `write`, `writeln`, other routines required by the compiler when interacting with special data types, such as sets and 32-bit multiples. This library also contains routines that make Macintosh Pascal compatible with THINK Pascal. In fact, you could consider most of Macintosh Pascal as a subset of THINK Pascal, because many Macintosh Pascal programs can be compiled and executed in THINK Pascal.

THINK Pascal supports all Macintosh Toolbox libraries referenced in *Inside Macintosh, Volumes I-V*, both ROM- and RAM-based. ROM-based routines are stored in programmable ROM chips in the Macintosh machine. RAM-based routines are allocated to main memory by the operating system. Macintosh Toolbox routines are further classified as either stack-based or register-based routines. Stack-based routines follow standard run-time conventions by pushing the values and addresses of actual parameters onto a system runtime stack. This means that values and address references of parameters are referenced in main memory–increasing the execution time of a routine. Register-based routines access the values and addresses of their parameters through registers. A register is a hardware device for the temporary storage of information such as data or the address of a RAM location. For register-based routines that are functions, results are returned through the use of one or more registers. Register-based routines can access the values and addresses of their actual parameters faster than stack-based routines. The trade-off for using register-based routines is that fewer general-purpose registers are available for use during the execution of a routine. Its major advantage is in speed of execution, because access to registers is faster than access to the contents of addresses in random memory. On the other hand, stack-based routines can allow for a lengthy list of formal parameters, because only the run-time stack is needed for storage of parameter values. At the same time more registers are available during execution.

The THINK Pascal interface files hide the distinction between stack-based and register-based routines. These interface files provide information that is necessary for interaction with a Pascal program. The burden of properly interfacing and branching to the appropriate machine-code routine is left to the compiler and the linking program. The THINK Pascal compiler generates the appropriate inline calls to all register-based routines that are trapped by the system during execution. For most of the register-based Toolbox routines that are both RAM- and ROM-based, THINK Pascal uses the file `Interface.lib` for "gluing" the appropriate code where a Pascal program invokes a Toolbox routine. Following is a list of interface files that are incorporated in the file `Interface.lib`. The names of these files should not be included in any **uses** clause in a THINK Pascal program. (Recall the number of times you have been told to remove the

```
uses
   QuickDraw1;
```

clause from a Macintosh Pascal program in order to make a program execute under THINK Pascal.)

Controls.p	Desk.p	Devices.p	Dialogs.p
DiskInit.p	Errors.p	Events.p	Files.p

Fonts.p	GestaltEqu.p	Lists.p	MacPrint.p
Memory.p	MemTypes.p	Menus.p	OSEvents.p
OSIntf.p	OSUtils.p	Packages.p	PackIntf.p
PaletteMgr.p	PickerIntf.p	QDOffscreen.p	QuickDraw.p
Resources.p	Scrap.p	SCSIIntf.p	SegLoad.p
Sound.p	StandardFile.p	TextEdit.p	ToolIntf.p
ToolUtils.p	Types.p	VideoIntf.p	Windows.p

For each of these built-in interfaces, THINK Pascal provides a dummy interface file containing no definitions. These files are provided to ease the difficulty of porting Pascal programs from other development systems into THINK Pascal. When one of the dummy interface files is added and built in a Project window, it is unnecessary to modify a unit that includes one of these interfaces in its **uses** clause.

Not all available interface files are incorporated in `Interface.lib`. Following is a list of interface files excluded from `Interface.lib`:

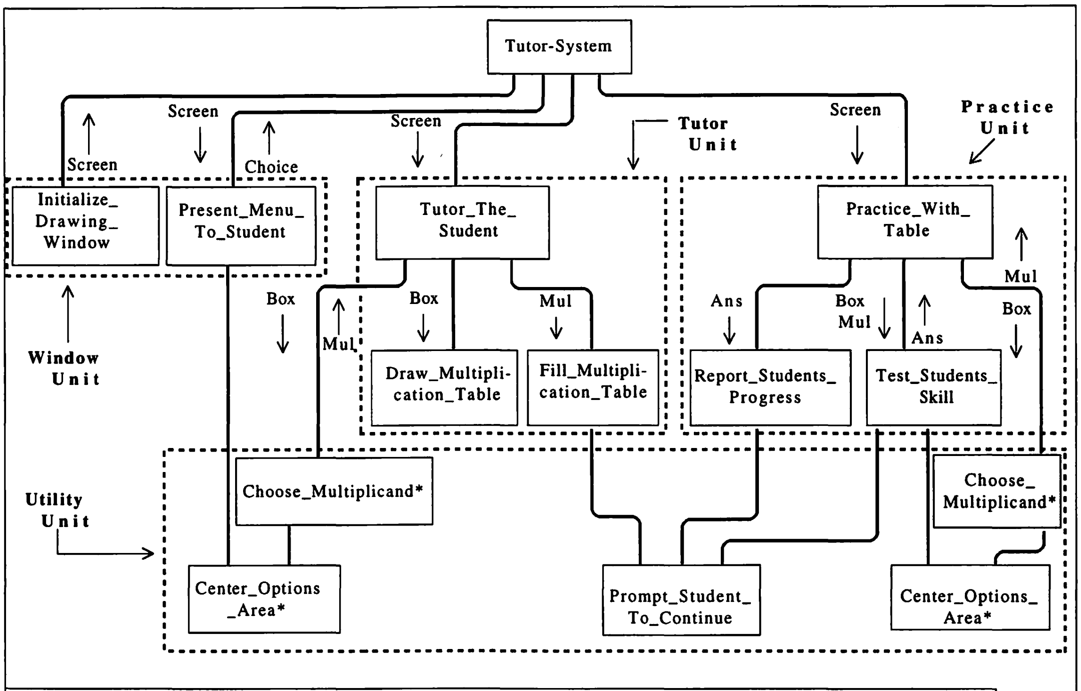
ADSP.p	AIFF.p	Aliases.p
AppleEvents.p	AppleTalk.p	Balloons.p
CommResources.p	Connections.p	ConnectionTools.p
CRMSerialDevices.p	CTBUtilities.p	DatabaseAccess.p
DeskBus.p	Disks.p	Editions.p
ENET.p	EPPC.p	FileTransfers.p
FileTransferTools.p	Finder.p	FixMath.p
Folders.p	Graf3D.p	HyperXCmd.p
Icons.p	Language.p	MIDI.p
Notification.p	ObjIntf.p	Palettes.p
PasLibIntf.p	Picker.p	PictUtil.p
Power.p	PPCToolBox.p	Printing.p
PrintTraps.p	Processes.p	Retrace.p
ROMDefs.p	SANE.p	Script.p
SCSI.p	Serial.p	ShutDown.p
Slots.p	Sound.p	SoundInput.p
Start.p	Strings.p	SysEqu.p
Terminals.p	TerminalTools.p	Timer.p
Traps.p	Video.p	

If these interface files are used, they must be explicitly included in a Project window as well as in a unit's **uses** clause. In some cases, both the interface file and and its associated library file must be included if the program and/or unit is to be successfully compiled or built. The following list includes those interface files that require an affiliated library file along with an interface file. In some circumstances, we must choose between two possible library files:

AppleTalk.p	ABPackage.Lib, nAppleTalk.Lib
FixMath.p	FixMath.lib
Graf3D.p	Graf3D.Lib
HyperXCmd.p	HyperXLib.Lib
Printing.p	PrintCalls.Lib
SANE.p	SANELib.lib, SANELib881.lib

8.6 APPLYING ADDED MODULARIZATION TO THE TUTOR SYSTEM

While it is common in software engineering to refer to a procedure or function as a *module*, the idea of modularization is furthered by packaging the definitions of constants, types, variables, and routines within units. Consider the hierarchical diagram of `Tutor_System` given in Figure 8.13.



Notes:
The parameter "Box" is an abbreviation for `Drawing_Box`
The parameter "Mul" is an abbreviation for `Multiplicand`.
The parameter "Ans" is an abbreviation for `Number_Of_Correct_Answers`.
The * indicates that a module appears more than once.

Figure 8.13 The hierarchical diagram of the tutor system given in Chapter 7, with modules divided among four units.

Four zones appear in this figure, representing distinct program units. Each unit is chosen to provide a basic area of relevance to the hierarchical structure of the tutoring system. For example, at the second level of the hierarchical diagram, there are three unique units. `Window Unit` defines routines for initializing the Drawing window as well as presenting a menu to the student. `Tutor Unit` provides the routines that drill the student, and `Practice Unit` serves to test the student's skills. These three units borrow routines from `Utility Unit`, and the main executive program borrows routines from `Window Unit`, `Tutor Unit`, and `Practice Unit`. A fifth unit, called `Data Unit` (not seen in Figure 8.13), is used to store a single data type called `Options`. Within `Tutor Unit` are three modules called `Tutor_The_Student`, `Draw_Multiplication_Table`, and `Fill_Multiplication_Table`. The latter two modules are subordinate to Tutor_The_Student. Each subordinate module is defined as a local routine within the implementation portion of Tutor Unit. This same argument applies to the routines Report_Students_Progress, and Test_ Students_Skill in Practice Unit. The module Choose_Multiplicand is moved into the Utility Unit, because it serves both Tutor Unit and Practice Unit.

Below are the listings of the units. Code for the bodies of the routines is not included. The units are built in the order in which they are listed.

```
unit Data_Unit;
interface
   type
      Option = (Tutor, Practice, Quit);
implementation
end.
```

```
unit Utility_Unit;
interface
   procedure Center_Options_Area;
   procedure Choose_Multiplicand (Drawing_Box: Rect;
                                   var Multiplicand: integer);
   procedure Prompt_Student_To_Continue;
implementation
{ ---------------- List of module definitions. ---------------- }
   procedure Center_Options_Area;
   { Purpose:  This procedure establishes the rectangle area in }
   {  the Drawing window for viewing various options. }
   begin
   ...
   end;  { Center_Options_Area }
{ ----------------------------------------------------------- }
   procedure Choose_Multiplicand;
   { ( Drawing_Box:Rect; var Multiplicand: integer ) }
   { Purpose:  This module returns a value representing the }
   {           multiplicand.}
      var
         Time: longint;
   begin
   ...
   end; { Choose_Multiplicand }
{ ----------------------------------------------------------- }
   procedure Prompt_Student_To_Continue;
   { Purpose:  This procedure prompts the student to continue }
   {           execution by having the student click the mouse }
   {           button. }
      var
         Time: longint;
   begin
   ...
   end; { Prompt_Student_To_Continue }
```

```
{ ------------------------------------------------------------------- }
end.
```

```
unit Window_Unit;
interface
   uses
      Utility_Unit, Data_Unit;
procedure Initialize_Drawing_Window (var Drawing_Box: Rect);
procedure Present_Menu_To_Student (Drawing_Box: Rect; var
                                          Choice: Option);
implementation
{ --------------- List of module definitions. ---------------- }
   procedure Initialize_Drawing_Window; { var Drawing_Box: }
   {                                            Rect)}
   { Purpose:  This procedure sets the boundaries for both the }
   {           Drawing window and the Text window. The Drawing }
   {           window will be located directly below the title }
   {           bar; the remainder of the screen shows only the }
   {           Drawing window. }
      var
         Text_Box: Rect;
   begin
   ...
   end; { Initialize_Drawing_Window }
{ ------------------------------------------------------------------- }
   procedure Present_Menu_To_Student;
   { (Drawing_Box:Rect; var Choice: Option ) }
   { Purpose:  This procedure returns a value as a choice for an }
   {           option. }
      var
         X, Y: integer;
         Screen_Point: Point;
   begin
   ...
   end; { Present_Menu_To_Student }
{ ------------------------------------------------------------------- }
end.
```

```
unit Tutor_Unit;
interface
   uses
      Utility_Unit;
   procedure Tutor_The_Student (Drawing_Box: Rect);
implementation
{ ---------------- List of forward directives. --------------- }
   procedure Draw_Mutiplication_Table (Drawing_Box: Rect);
   forward;
   procedure Fill_Mutiplication_Table (Multiplicand: integer);
   forward;
```

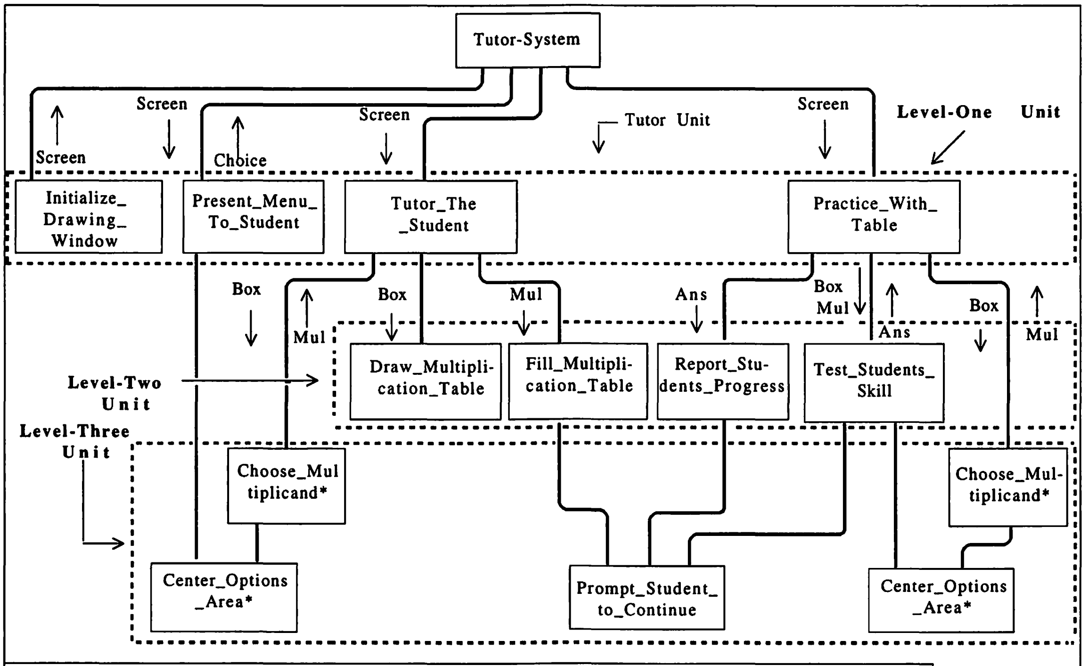
```
{ --------------- List of module definitions. ---------------- }
   procedure Draw_Mutiplication_Table; { (Drawing_Box: Rect) }
   { Purpose:  This procedure draws a multiplication table under }
   {           the option  to tutor the student. }
   var
      Y: integer;
   begin
   ...
   end; { Draw_Multiplication_Table }
{ ------------------------------------------------------------ }
   procedure Fill_Mutiplication_Table; {(Multiplicand: integer)}
   { Purpose:  This module fills the contents of smaller }
   {           rectangles in the multiplication table with }
   {           strings composed of a multiplier, multiplicand, }
   {           and product. }
      var
         Multiplier, Inner_Count, Outer_Count, X, Y: integer;
   begin
   ...
   end; { Fill_Mutiplication_Table }
{ ------------------------------------------------------------ }
   procedure Tutor_The_Student; { ( Drawing_Box: Rect) }
   { Purpose:  This procedure presents a multiplication table }
   {           for the student to review. }
      var
         Multiplicand: integer;
   begin
   ...
   end; { Tutor_The_Student }
{ ------------------------------------------------------------ }
end.
```

```
unit Practice_Unit;
interface
   uses
      Utility_Unit;
   procedure Practice_With_Table (Drawing_Box: Rect);
implementation
{ ---------------- List of forward directives. --------------- }
   procedure Report_On_Students_Progress
   (Number_Correct_Answers: integer);
   forward;
   procedure Test_Students_Skill (Drawing_Box: Rect;
                                       Multiplicand: integer;
      var Number_Correct_Answers: integer);
         forward;
{ --------------- List of module definitions. ---------------- }
   procedure Report_On_Students_Progress;
   { (Number_Correct_Answers: integer) }
   { Purpose:  This procedure provides a short report of the }
   {  student's progress. }
      var
         Time: longint;
   begin
   ...
   end; { Report_On_Students_Progress }
{ -------------------------------------------------------------- }
   procedure Test_Students_Skill;
   { ( Drawing_Box : Rect; Multiplicand: integer;
   {   var Number_Correct_Answers: integer ) }
   { Purpose:  This procedure checks the multiplication skill of }
   {  the student. }
      var
         Multiplier, Product, Student_Answer: integer;
         Time: longint;
   begin
   ...
   end; { Test_Students_Skill }
{ -------------------------------------------------------------- }
   procedure Practice_With_Table; { (Drawing_Box: Rect) }
   { Purpose:  This procedure lets the student practice with his }
   {           or her own multiplication table. }
      var
         Multiplicand, Number_Correct_Answers: integer;
   begin
   ...
   end; { Practice_With_Table }
{ -------------------------------------------------------------- }
end.
```

```
program Tutor_System (Input, Output);
{ Purpose:  This program provides a student with the option to }
{           study and practice with his or her own }
{           multiplication tables. }
   uses
      Data_Unit, Window_Unit, Tutor_Unit, Practice_Unit;
   var
      Choice: Option;
      Screen: Rect;
{ =============== Body of the main program. ================== }
begin
   Initialize_Drawing_Window(Screen);
   repeat
   {Prompt the student for a choice. }
      Present_Menu_To_Student(Screen, Choice);
      case Choice of
            Tutor:
               Tutor_The_Student(Screen);
            Practice:
               Practice_With_Table(Screen);
```

```
        Quit:
          HideAll;
    end;
  until Choice = Quit;
end.
```

Figure 8.14 shows a second approach for enclosing the modules of Tutor_System in units. Here the units are based upon the levels of the hierarchical chart rather than on their functional capabilities. Notice that only three units are necessary; a fourth unit stores the data type called `Options`. Which approach is better? There is no clear answer, because this can depend upon the software maintenance aspect of the present application. The units in Figure 8.13 are based upon specific functionality, whereas Figure 8.14 shows no functional relationships between units. In Figure 8.13 if `Practice Unit` required changes to the code of the module `Choose_Multiplicand`, which is not needed by `Tutor_Unit`, then a copy with the changes could be placed into `Practice Unit`, and the older copy of `Choose_Multiplicand` placed into `Tutor_Unit`. Each module would be placed in the private portion of the proper unit, with each module acting as a procedure local to the unit.



Notes:
The parameter "Box" is an abbreviation for Drawing_Box
The parameter "Mul" is an abbreviation for Multiplicand.
The parameter "Ans" is an abbreviation for Number_Of_Correct_Answers.
The * indicates that a module appears more than once.

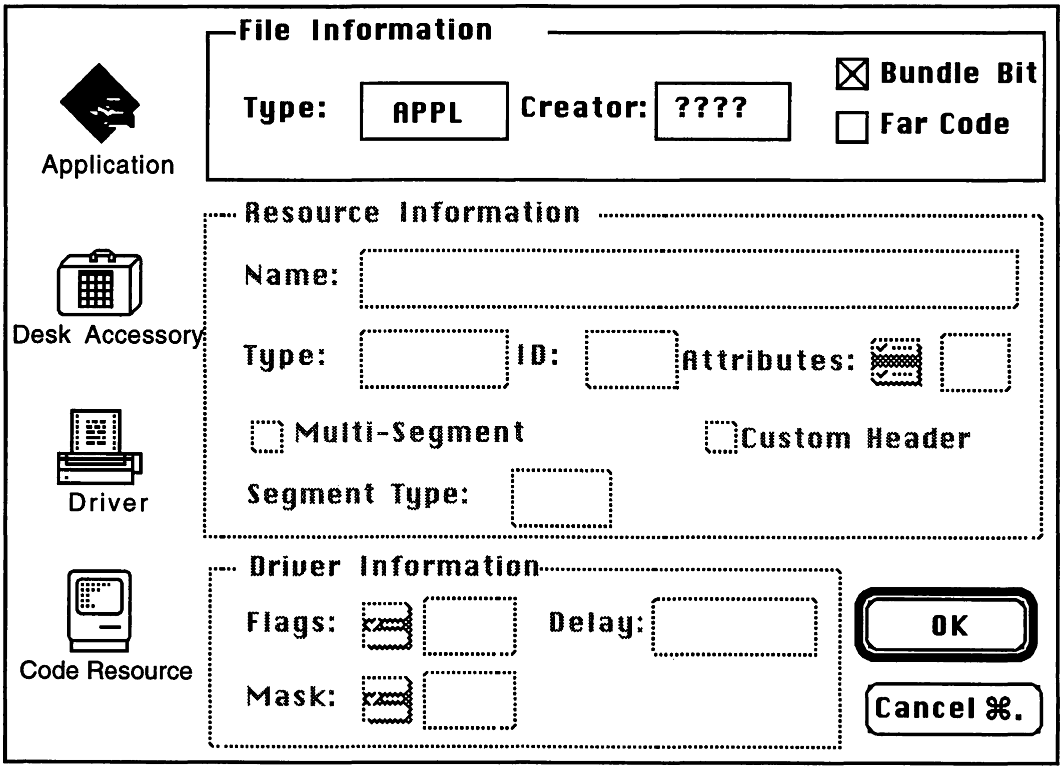
Figure 8.14 The hierarchical diagram of the tutor system, with units based on levels.

With the plan shown in Figure 8.14, modifying the module `Choose_Multiplicand` would be more difficult, because it is shared with modules in Level-One Unit. Here two distinct `Choose_Multiplicand` modules would be required and would be kept in Level-Two Unit.

When developing complex applications, planning for proper software maintenance is important. Good software applications will grow as enhancements are suggested. Planning for this growth during the early phases of development and implementation provides benefits in later stages of maintenance. Effective use of program units can help in establishing libraries that can be shared among several applications, as well as in building applications that are error-free.

8.7 ALLOCATING A PROJECT AS AN APPLICATION

In the previous sections we examined the steps for building projects and establishing programmer-defined libraries. THINK Pascal allows the programmer to establish a project as an application by applying the options **Select Project Type...** and **Build Application...** from the **Project** menu. When working on a project, a programmer can set the project type by selecting the command option **Select Project Type...** from the Project menu. Figure 8.15 shows the dialog window for setting the type of project.



File Information

Type: APPL Creator: ????

[✓ Bundle Bit] Far Code

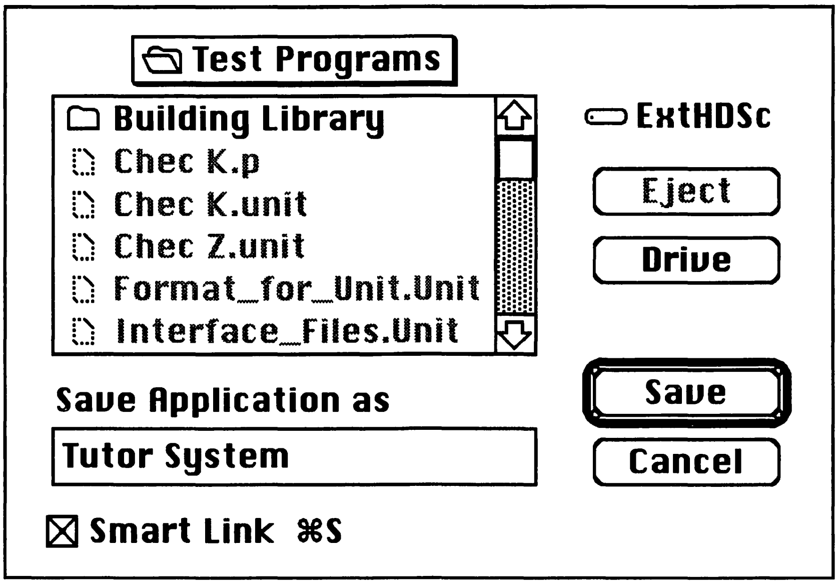
Figure 8.15 The dialog window for establishing an application from a project.

Notice that it offers four project types; application, desk accessory, driver, or code resource. This section deals only with establishing a project as an application. References on establishing desk accessories, drivers, and code resource can be found in the *THINK Pascal User Manual*, Chapter 12, “Building Projects” (Symantec Corporation) and in various chapters of *Inside Macintosh Volume I*, *Inside Macintosh Volume II*, *Inside Macintosh Volume IV*, and *Inside Macintosh Volume V*.

By default, you can choose project type APPL (short for application). The creator field, a unique four-character signature, identifies the application that has created the file. Having the **Bundle Bit** set lets the Finder know that the application being created has a resource type BNDL with its own icon. No other information is required for setting the project as an application type. On clicking the **OK** button, the Finder will learn the kind of file that is being created as well as how to display it on the desktop.

With the **Far Code** option set, THINK Pascal allows large applications to be written that include jump tables as large as 256K bytes. A jump table contains the entry of any routine that a program must invoke across boundary segments of RAM memory. When the option is *off*, a jump table can only support jumps with a maximum of 32K bytes.

The dialog window in Figure 8.15 sets the project type, andthe option **Build Application...** allows you to select a file name for an application. As shown in the dialog window of Figure 8.16, the name `Tutor System` for the program called `Tutor_System.THINK` is entered as the application name. When the **Smart Link** option is set, only referenced object code of sources and libraries by units and the main Pascal program are used to create the resulting application file. Functions and procedures that are not referenced are not included. Choosing the **Smart Link** option increases the time for building an application, but the resulting object file for the application is in general smaller than if the option was not set.



Save Application as

Tutor System

☒ Smart Link ⌘S

Figure 8.16 The dialog window for naming and saving an application.

We recommend that you leave the **Smart Link** unset if you repeat the building of an application for frequent testing. On leaving this last dialog window and then leaving the project, your application is ready for execution. Double-clicking on the name or on the application icon opens the application for execution. It is important to understand that the Drawing and Text windows will be closed once the application ends execution. You must provide a prompt for the user if the windows are to remain open for examination before execution of the application terminates. Once execution ends, all information shown in the Text and Drawing windows will be lost, because applications are always executed from outside the THINK Pascal environment. In addition, applications cannot be executed by entering THINK Pascal, opening a project, and building a Project window.

8.8 USING THE PROFILER TO COLLECT PROGRAM STATISTICS

THINK Pascal offers a profiler that collects statistics on any program executed from within a project. THINK Pascal's code profiler provides a summary on execution time for each routine. The actions of the profiler are set by turning on the **Profile** option in the dialog window of **Compile Options...** in the **Project** menu. The option **Long Names** should be set if the routines being examined have names longer than eight characters. The **Debug** and **Names** options in the Project window must be turned on for any unit containing routines that are being measured (profiled). No statistics are collected on routines in a program unit if the unit's **Debug** option is turned off. If the **Debug** option is on but the **Names** option is turned off, the profiler will not list the name of any routine that is being measured; instead it lists routine names as `????????`. When a program ends or halts execution, the profiler writes a report to the Text window. By choosing options such as echoing to the printer or to a file, you can direct what is displayed in the Text window to an output file.

As a tool for examining the execution characteristics of a program, the profiler records information on every procedure or function that is invoked. The code profiler examines only those routines that are defined within a program unit but does not report on the routines from the Macintosh Toolbox. Figure 8.17 shows a report on routines from `Tutor_System`. Notice that all time is measured in milliseconds (10^{-3} seconds).

This profile report has six columns: the name of each routine that has been executed; the minimum, maximum, and average time spent when executing each routine; the total time spent in execution of each routine; the percentage of the program's time spent on execution of each routine; and the number times that each routine was invoked. Routines are listed in the order of their invocation as the program is being executed. For `Tutor System` most of the execution time is spent in `Prompt_Student_To_Continue` and in `Test_Students_Skill`. As the table shows, the routines toward the top of the hierarchical structure like `Tutor_System`, `Present_Menu_To_Student`, `Tutor_The_Student`, and `Practice_With_Table` take little time in execution, because their primary function is to make decisions. For each of these superordinate modules, most of the work is performed by the subordinate modules. Adding the interface file `Profile.p` to your Project window and adding the unit name `Profiler` to the **uses** clause of any unit that directly calls upon profiler routines, gives you additional control over capturing profiled information from within the program.

```
THINK Pascal Procedure Profile

All time is measured in milliseconds.
rounded down to the nearest millisecond.

Elapsed Time  =    77329
Measured Time =    77327
Total Calls   =       45


Routine                              Min     Max     Avg    Total   %       Times
Name                                 Time    Time    Time   Time    Time    Called
________                             ____    ____    ____   ____    ____    ______
 CENTER_OPTIONS_AREA                   25      27      25     462   0.60        18
 CHOOSE_MULTIPLICAND                 3365    3504    3434    6869   8.88         2
 DRAW_MULTIPLICATION_TABLE             64      64      64      64   0.08         1
 FILL_MULTIPLICATION_TABLE             77      77      77      77   0.10         1
 INITIALIZE_DRAWING_WINDOW            272     272     272     272   0.35         1
 PRACTICE_WITH_TABLE                    4       4       4       4   0.01         1
 PRESENT_MENU_TO_STUDENT             1435    2871    1989    5967   7.72         3
 PROMPT_STUDENT_TO_CONTINUE          1084    3301    1668   23360  30.21        14
 REPORT_ON_STUDENTS_PROGRESS         3751    3751    3751    3751   4.85         1
 TEST_STUDENTS_SKILL                36348   36348   36348   36348  47.01         1
 TUTOR_SYSTEM                         151     151     151     151   0.20         1
 TUTOR_THE_STUDENT                      2       2       2       2   0.00         1
```

Figure 8.17 THINK Pascal profile report on execution of `Tutor_System`.

The following briefly reviews the procedures, function, and the profile variable that allow control of profiler:

```
var %_PTrace : Boolean ;
```

When set to the Boolean value *false*, the profiler turns off the collection of statistics. Reestablishing the collection of information is accomplished by resetting its value to *true*.

```
procedure DumpProfile;
```

This procedure will dump (display) current statistics on the program in execution to the Text window.

```
procedure DumpProfileToFile( File_Name : Str255 );
```

This procedure dumps (writes) the current statistics of the program in execution to a physical file in the default volume represented by the formal parameter `File_Name`. If the file exists, it is initially treated as an empty file and then written into, with all previous data being lost. If the file does not exist, it is created and written into as an empty file. The default volume is assumed to be the folder where the project is currently located.

```
procedure ResetProfile;
```

This routine reinitializes the profiler and its statistics. It allows the collection of multiple sets of statistics on portions of a program that repeat execution of an application.

```
procedure TerminateProfile;
```

This routine halts the collection of statistics by the profiler.

```
function InitProfiler( Number_Procedures, Number_Activations
    : integer ) : Boolean;
```

The formal parameter `Number_Procedures` represents the maximum number of routines that can be tracked by profiler. The formal parameter `Number_Activations` represents the maximum number of activations for any recursive routine. With the **Profile** option set, the profiler is automatically initialized and reserves memory space for tracking a maximum of 200 routines. The maximum number of routines to be tracked can be changed by invoking this function as the first executable statement in a program and passing to it values for `Number_Procedures` and `Number_Activations`. The maximum value for the formal parameter `Number_Procedures` is 32,768. The value passed to `Number_Activations` is 1 unless one or several recursive routines are present requiring a maximum number of activations. This function returns the `Boolean` value *true* if profiler can be initialized; otherwise, it returns *false.*

Following is a listing of the main program for the tutor system that uses the profiler to collect statistics. Profiler functions are invoked to collect data only on the superordinate module `Practice_With_Table` and its corresponding subordinate modules. Notice that the unit name `Profiler` has been added to the **uses** clause along with the list of other units being borrowed. This is necessary, because the program is using the variable `%_PTrace` as well as several profiler routines. In the initial steps, the profiler is turned off by assigning the value *false* to `%_PTrace`. When the program executes the case option `Practice`, the profiler is made active, and before it begins collecting information, it executes the routine `ResetProfile` to reinitialize all data to zero. When the practice session ends, the routine `DumpProfile` writes the current collection of information to the Text window. The profiler is then disabled by assigning the value *false* to `%_PTrace`.

```
program Tutor_System (Input, Output);
{ Purpose:  This program provides a student with the option to }
{           study and practice with his or her own }
{           multiplication tables.  }
   uses
      Profiler, Data_Unit, Window_Unit, Tutor_Unit, Practice_Unit;
var
   Choice: Option;
   Screen: Rect;
{ =============== Body of the main program. =================== }
begin
{ Turn off the collection of statistics by the profiler. }
   %_PTrace := false;
{ Continue to execute the program with the collection of }
{ statistics being activated at a later point within the program.}
```

```
   Initialize_Drawing_Window(Screen);
   repeat
   {Prompt the student for a choice. }
      Present_Menu_To_Student(Screen, Choice);
      case Choice of
            Tutor:
               Tutor_The_Student(Screen);
            Practice:
               begin
            { Reestablish the collection of statistics by the }
            { profiler. }
                  %_PTrace := true;
            { Reinitialize the profiler and its statistical }
            { information. }
                  ResetProfile;
            { Provide a practice session with Practice Unit. }
                  Practice_With_Table(Screen);
            { Write the current statistics to the Text window }
            { and then turn off the collection of information }
            { by the profiler. }
                  DumpProfile;
                  %_PTrace := false;
               end;
            Quit:
               HideAll;
      end;
   until Choice = Quit;
end.
```

The following shows the contents of the Text window for one practice session of the tutor system:

```
THINK Pascal Procedure Profile

All time is measured in milliseconds.
rounded down to the nearest millisecond.

Elapsed Time  =    71616
Measured Time =    68501
Total Calls   =       31


Routine                          Min     Max     Avg    Total   %       Times
Name                             Time    Time    Time   Time    Time    Called
_______                          ____    ____    ____   ____    ____    ______
 PRACTICE_WITH_TABLE                3       3       3       3   0.00         1
 CHOOSE_MULTIPLICAND             4081    4081    4081    4081   5.96         1
 CENTER_OPTIONS_AREA               25      26      25     358   0.52        14
 TEST_STUDENTS_SKILL            37140   37140   37140   37140  54.22         1
 PROMPT_STUDENT_TO_CONTINUE       944    5759    1781   23164  33.82        13
 REPORT_ON_STUDENTS_PROGRESS     3755    3755    3755    3755   5.48         1
```

Be sure that you disable the option for profiling before leaving the project and before building a final application. If you forget, the profiler will be included and will add extra overhead in executing your application and will report useless information to the application user. Understand that the profile option only allows us to examine the execution time of routines. It does not trace the values of variables, nor the execution time taken by individual control statements.

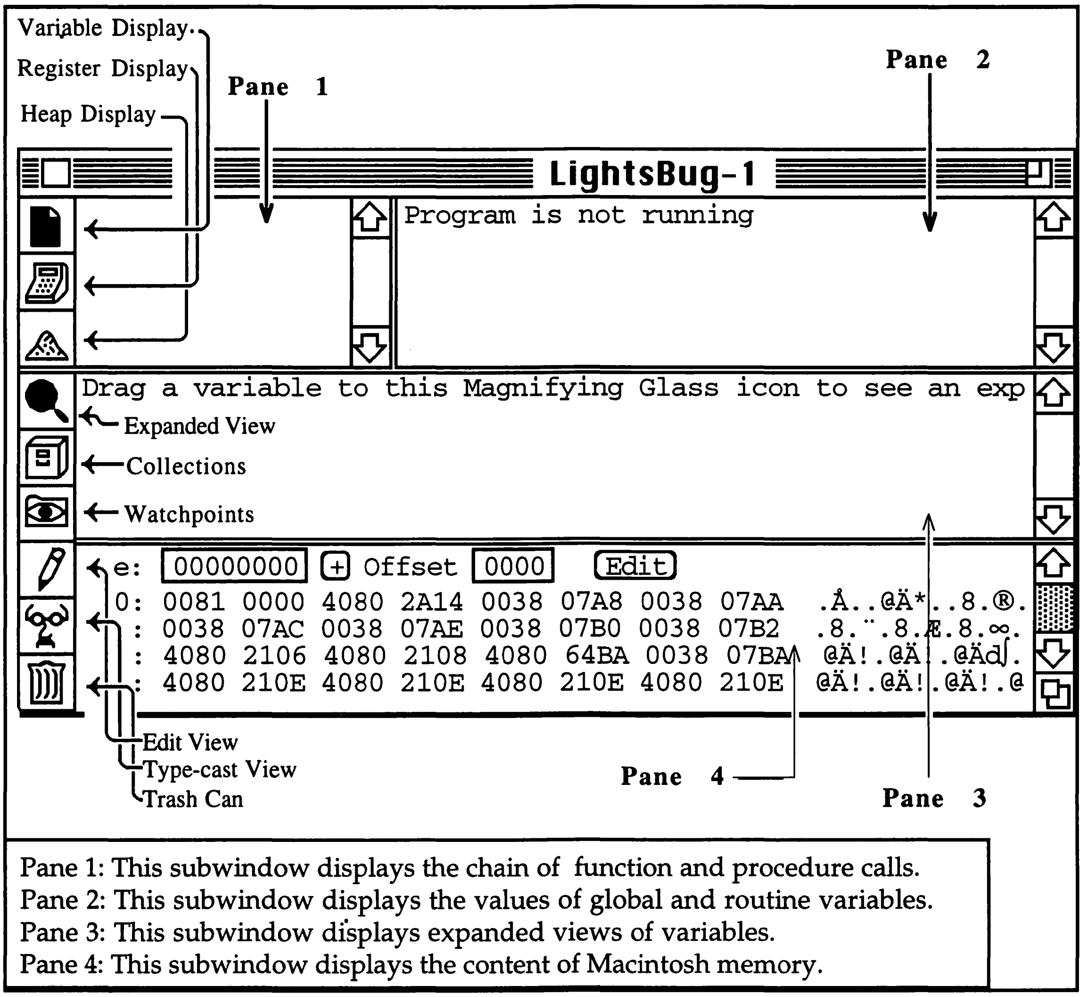
8.9 USING LIGHTSBUG FOR VIEWING THE EXECUTION OF A PROGRAM

LightsBug is a sophisticated programming tool for examining the execution of a program. It is different from a profiler, because it allows us to examine the values of variables and routine parameters as the program executes, and also allows us to change the content of memory. It does not provide statistical information on the execution of a program.

As a programming tool, a debugger is an application that can trace the execution of a program and allow us to interrupt its execution by inserting break points (modifying one or more source code statements). If the program fails to execute, the debugger allows us to jump around error-prone statements and continue execution by testing the remainder of a program. A debugger examines the code during execution of a program, but it does not examine the correctness of the program as an algorithm.

LightsBug allows us to view the order in which all procedures and functions are invoked. By inserting stops (break points) in a THINK Pascal program, we can observe the values of local and global variables before continuing execution. Programmers who are knowledgeable about the Macintosh at the machine level can choose to view the hexadecimal content of CPU registers and memory allocations within blocks and heaps of the software application and within the system heap directly on the screen.
The THINK Pascal debugger is made active by choosing the command option **LightsBug** from the **Debug** menu. For all units being examined, LightsBug works better when both the **Debug** and **Names** options are set within the Project window. Figure 8.18 shows the LightsBug window when LightsBug is first activated. Notice that the LightsBug window is divided into four subwindows called *panes*. Each pane provides a separate view of the program during execution, and each has its own vertical elevator bar. For example, the upper left pane allows us to view the chain of routines that are called as the program is being executed. This window is useful in observing routines that are either directly or indirectly recursive. The upper right pane displays the values of global, local, and parameter values as the program executes. It allows us to observe the content of various objects when the program in execution is interrupted. The center pane allows for expanded views of data. In particular, we can observe the expanded values of variables such as arrays and records. The bottom pane shows the contents of Macintosh memory as it relates to the execution of the program. Each pane can be made larger or smaller by dragging the double line that separates the panes. Panes can be adjusted to cover the complete screen or only a portion of the screen.

The four groups of icons to the left of the windows provide various options when viewing the execution of a program. The icon referred to as Variable Display allows the left and right panes at the top of LightsBug window to show all values of variables visible in one of the routines selected in the chain of calls. Selecting the content of one of the routines is done by highlighting one of the names listed in Pane 1; the values of all variables associated with the routine are displayed in Pane 2. The values of constants are



```
Program is not running
```

```
Drag a variable to this Magnifying Glass icon to see an exp
```

```
e: 00000000 + Offset 0000  Edit
0: 0081 0000 4080 2A14 0038 07A8 0038 07AA   .Å..@Ä*..8.®.
 : 0038 07AC 0038 07AE 0038 07B0 0038 07B2   .8.¨.8.Æ.8.∞.
 : 4080 2106 4080 2108 4080 64BA 0038 07BA   @Ä!.@Ä!.@Äd∫.
 : 4080 210E 4080 210E 4080 210E 4080 210E   @Ä!.@Ä!.@Ä!.@
```

Pane 1: This subwindow displays the chain of function and procedure calls.
Pane 2: This subwindow displays the values of global and routine variables.
Pane 3: This subwindow displays expanded views of variables.
Pane 4: This subwindow displays the content of Macintosh memory.

Figure 8.18 Format of the LightsBug window.

never displayed, because they never change. Although the contents of both Panes 1 and 2 can be highlighted, they cannot be cut or copied while the Project is active.

Figure 8.19 shows the contents of Panes 1, 2, and 3 for the program `Draw_Rings` and procedure `Draw_Concentric_Rings`. In the LightsBug window shown a stop is inserted at the beginning of procedure `Draw_Concentric_Rings`, using the option **Stops In** from the **Debug** menu.

This is necessary to stop the program before the body of the procedure is executed, so that we can examine the values of parameters and variables. All stops from the current Program window can be removed by selecting the **Pull Stops** option from the same menu. Pressing the option key makes the option **Pull All Stops** available for pulling all stops from all files in the project. When the program stops for the first time, the LightsBug window is opened by selecting the **LightsBug** option from **Debug** menu.

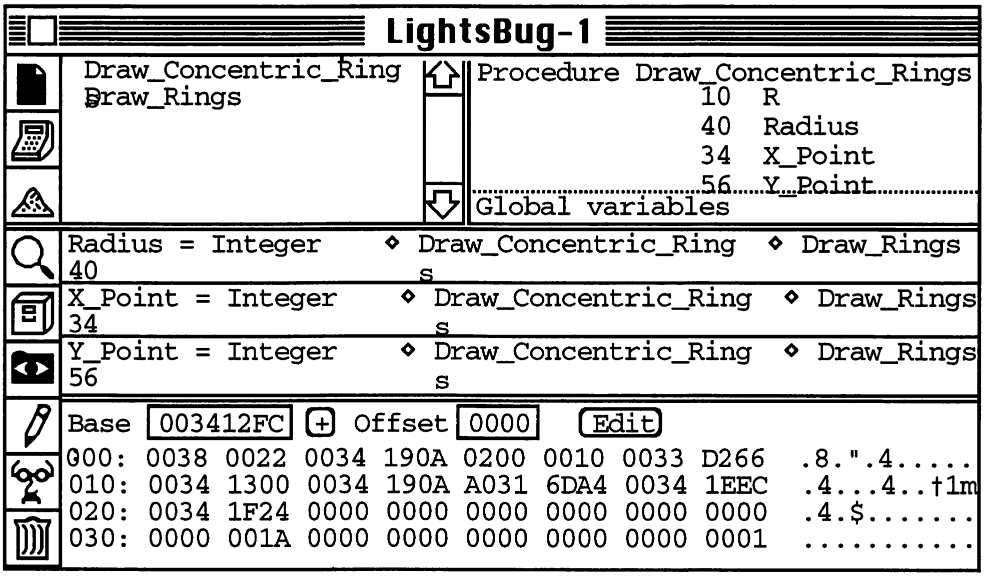

```
LightsBug-1
Draw_Concentric_Ring      Procedure Draw_Concentric_Rings
Draw_Rings                            10  R
                                      40  Radius
                                      34  X_Point
                                      56  Y_Point
                          Global variables
Radius = Integer       ◆ Draw_Concentric_Rings   ◆ Draw_Rings
40
X_Point = Integer      ◆ Draw_Concentric_Rings   ◆ Draw_Rings
34
Y_Point = Integer      ◆ Draw_Concentric_Rings   ◆ Draw_Rings
56
Base 003412FC  +  Offset 0000   Edit
000: 0038 0022 0034 190A 0200 0010 0033 D266  .8.".4.....
010: 0034 1300 0034 190A A031 6DA4 0034 1EEC  .4...4..†1m
020: 0034 1F24 0000 0000 0000 0000 0000 0000  .4.$.......
030: 0000 001A 0000 0000 0000 0000 0000 0001  ...........
```

Figure 8.19 The LightsBug window, where the values of variables for procedure `Draw_Concentric_Rings` are being tracked as their values change during execution of the program. In this example the Watchpoints icon is active.

In Figure 8.19 each of the variables, `Radius`, `X_Point`, and `Y_Point` has been highlighted and dragged to the Watchpoints icon for further observation of its values. Each time the value of one of these variables changes, the program is interrupted, and the LightsBug window is opened with the values of all of the variables displayed in Panes 2 and 3. Execution can be continued by using the option **Go** from the **Run** menu. In the context of debuggers, inserting a stop in a THINK Pascal Program window is the same as setting a break point for a debugger. A break point stops execution while a debugger is executing a program. The option **Stops In** and the use of the LightsBug window work as partners in helping the programmer observe the actions of break points as a THINK Pascal program executes.

By first making the Expanded View icon active and then dragging the name of a variable from Pane 2 into this icon, Lightsbug allows the value of a simple variable or the values of all the elements of a structured variable such as an array, record, or set to be displayed. The Expanded View icon can hold only a single value at any one time. The Collections icon can display the values of several variables when the LightsBug window is active. When using either the Expanded View or Collections icons, we must insert the proper stops for breaking execution of the program; otherwise we will fail to observe the changes in variables being analyzed as the program executes.

The Watchpoints icon allows the values of several variables to be held, and whenever the value of one of these variables changes, the THINK Pascal program stops execution and displays the window for this icon. When using the Watchpoints icon, we do not need to apply a stop to a line of code that follows the statement that changes the value of a variable.

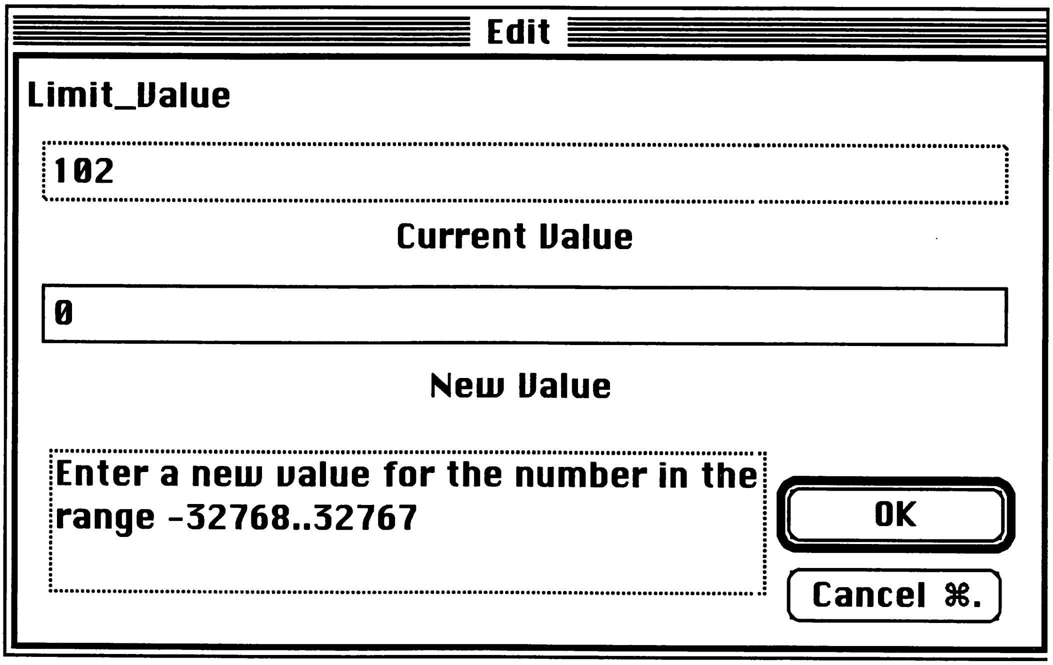
The icon register display allows us the option of viewing the contents of CPU registers as hexadecimal numbers. If the 68881/68882 option is set, Pane 2 also displays the contents of all eight floating-point registers. The Heap Display icon displays the application's heap zone or system heap zone.

Dragging a variable into the Edit Value icon displays a dialog box that allows us to change the value of a variable. By using this option, we can observe the program's characteristics when a different value is applied to a variable. The Type-cast view icon displays a dialog window where the type for a value can be recast (changed) without adding or changing the source code of the program. The dialog windows for both Edit and Type-cast icons is shown in Figure 8.20. The Trash Can icon allows variables from Pane 3 to be dragged and thrown away. This trash can is not the same as the Finder's trash can, in that you cannot drag things out of the LightsBug trash once they have been thrown away.

Additional options from the **Run** menu can be used to control the flow of execution. The command option **Step Over** allows us to execute a program one statement at a time. When this option is selected, one statement of the program is executed, and the execution finger points to the next statement ready for execution. By again choosing this option,we can execute the next statement in the body of the program unit. If this next statement is a procedure or function call, the command executes the routine without moving the execution finger into the routine. If we want to trace the individual statements of a routine, we must choose the option **Step Into** to move into the routine, and then use the option **Step Over** to execute each statement of the routine. While using these options, we are free to open the LightsBug window to observe changes to variables. If we decide to complete execution of a routine and halt any further tracing by the execution finger, we can choose the option **Step Out**.
The option **Break at A-Traps** in the **Debug** menu allows us to stop the program before calling a Macintosh Toolbox routine. By stopping the program at this point, we can examine the values of actual parameters by using the LightsBug window. The program can be made to continue execution by using either **GO** or **Step Over** from the **Run** menu. While the **Break at A-Traps** is set, you may have to apply the options **GO** or **Step Over** several times to complete execution of a toolbox routine, because the routine may itself call on several other toolbox routines. The following steps describe the setting and use of this option:

1. First, establish a break point (stop) to interrupt the program before the parameters of a toolbox routine are to be examined.
2. Be sure to establish a stop at the end of the main program, so that you can remove the A-traps setting before the program ends execution. *Failure to remove t' ^ setting will cause the Macintosh system to crash, and all desktop files will be lost.*
3. After reaching the first stop, set the **Break at A-Traps** in the **Debug** menu. Setting the **Auto-Show Finger** option may also be useful, showing where a toolbox routine is being invoked. You must click on the routine name in Pane 1 to observe the values of actual parameters of any toolbox routine.
4. When you decide to stop examining the memory content in the LightsBug window, be sure to remove the **Break at A-Traps** setting.

Recall that the Instant window will only execute a command if a program has been interrupted. For example, you may insert a stop to interrupt execution when the program reaches the statement with the stop. This interruption allows us to open the Instant



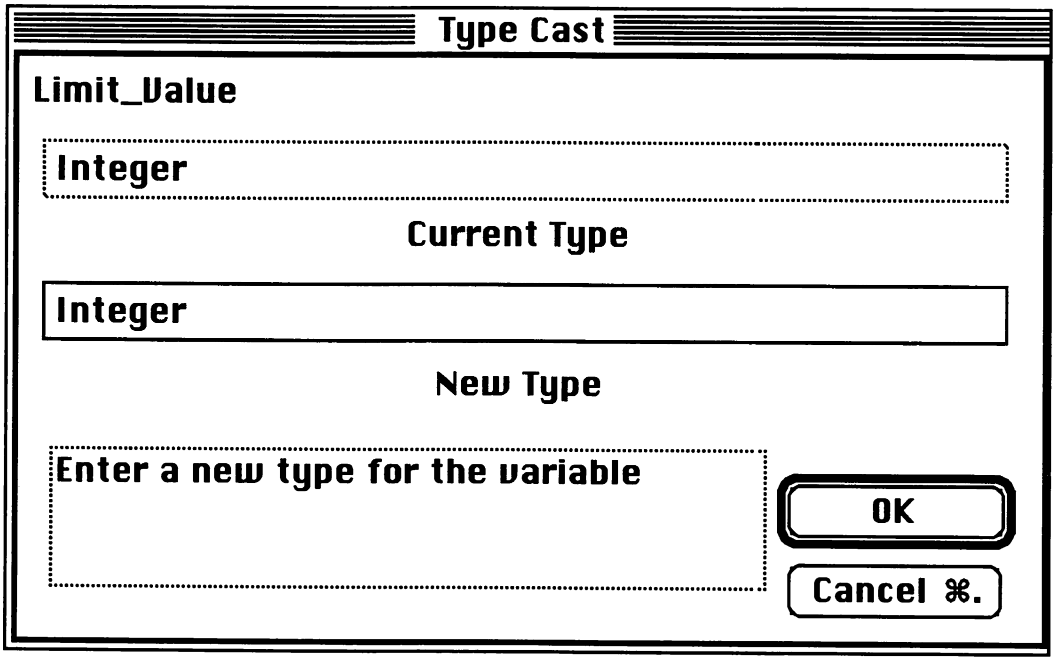


Figure 8.20. The dialog windows produced by choosing either the Edit or Type-cast View icons when the LightsBug window is active.

window from the **Debug** menu. Once it is opened, we can enter one or more commands and execute them by clicking the **Do It** button. Under normal execution using the **GO** option, the Instant window is not available for executing commands.

8.10 STANDARD PASCAL VERSUS THINK PASCAL

ANSI/IEEE standard Pascal neither recognizes nor supports the concept of units. By use of the reserved words **uses**, **interface**, and **implementation**, a programmer can extend the original definition of the Pascal language beyond the basic concept of internal and nested routines. Units allow programmers to define constants, types, variables, and routines that are external to a Pascal program. This concept goes beyond the basic scoping rule of global versus local identifiers, because the environment of routines lies beyond the contour of the main Pascal program. In addition, it allows information to be hidden and libraries, from which compiled routines are executed, to be built.

The concept of a unit and the application of the **uses** clause is not new to Pascal. This concept was first introduced in UCSD (University of California San Diego) Pascal for smaller machines such as Apple II and Radio Shack. Since that time it has become popular with most Pascal compilers implemented on personal computers and is also defined in Turbo Pascal. There is one basic difference between the THINK Pascal unit and the UCSD Pascal unit. The THINK Pascal unit does not support a **begin-end** statement at the end of its definition. Thus THINK Pascal cannot initialize variables in the interface-part of a unit through the execution of code in its implementation-part. Nor does THINK Pascal automatically execute code in its supporting units before the main Pascal program begins execution. While this may seem to be a limitation, initialization routines can always be written and called from the main program to initialize public variables that are being shared with other program units.

Standard Pascal does not define tools for analyzing the execution of any Pascal program. As a development system, THINK Pascal supports two basic tools: one that can analyze the run-time performance of an application and another for viewing step-by-step execution. The concept of a profiler is not new to computing; other larger systems such as UNIX and VAX VMS provide tools to examine run-time execution. Debuggers are common among computer systems that allow the development of software applications. Often called *symbolic debuggers*, they allow break points to be set, jumps to be defined, and often have the capability of examining and changing the values of variables.

SUMMARY

THINK Pascal allows programmers to build their own units and libraries. In turn, units can be borrowed by other units as well as by a Pascal program. This allows constants, types, variables, and routines to be hidden from view when reading other units or a Pascal program. It also allows using the resources that have been established by other programmers. Thus, programs can act as true executives, where subordinate routines are defined within units, and the main program acts only to make decisions. Compiling units and building libraries allows units to be shared among several other units and Pascal programs, as well as among several different projects.

The option to apply a **uses** clause is left to the programmer. If a **uses** clause is not declared, the programmer is required to state explicitly the constants, types, and variables hidden within a library or unit. This can force the application of special compile directives {$J+} and {$J–} to avoid duplication of memory storage. In addition, if the main Pascal program is borrowing routines from a library and a **uses** clause is not employed, the programmer must declare proper routine headers along with stating the external directive.

THINK Pascal supports two programming tools for examining the execution of a program. The profiler provides a report in the form of statistical information on any program executed from within a project. The information reported includes the minimum, average, and maximum times spent in executing each routine; the total time spent in execution of each routine; the percentage of the program's time spent on execution of each routine; and the number times that each routine has been invoked. Thus, a programmer can determine which routine is more heavily processor-bound. These statistics do not tell what the execution time is for individual control constructs such as conditional and iteration statements. However, the profiler can tell the programmer if the routines at the bottom of the hierarchical diagram perform more of the work, and the routines at the top, acting as supervisors, perform less. By borrowing from the unit `Profile.p`, the programmer can apply several routines and use the profile variable `%_PTrace` for capturing profiled information from within portions of a program.

The THINK Pascal debugger called LightsBug allows a programmer to examine the detailed execution of a program, statement by statement, through inserting stops and examining the LightsBug window at break points. Within this window the programmer can examine the value of a single variable, the values of a structured variable, or the values of several variables by making one of several icons active and by dragging the name of one or more variables into the active icon. The Instant window can be used to execute actions typed within it only when a THINK Pascal program has been interrupted. One of the subwindows (referred to as a *pane*) displays the chain of function and procedure calls up to the point where the program has been stopped. This data is useful for viewing the execution chain of direct and indirect recursive routines.

Both of these tools help in examining the execution of a program. However, while the programmer can observe the execution characteristics by measuring execution time and can examine the intermediate values of variables, neither of these tools can define the semantic errors of an algorithm. Their purpose is to analyze and debug a program in execution. They cannot help in detecting a logical error in an algorithm implemented as a program.

REVIEW QUESTIONS

1. How does THINK Pascal allow programs to be modularized?
2. In THINK Pascal, what is a unit?
3. How can other program units in THINK Pascal borrow from a unit?
4. What parts of a unit are accessible by either a Pascal program or another unit?
5. There are two basic sections to a unit: the interface-part and the implementation-part. Explain the purpose of each of these parts.
6. What is meant by the terms *public-part* and *private-part* as they apply to a unit?
7. Define the basic structure of a THINK Pascal unit. How is it similar to and different from a Pascal program?
8. Explain the steps for building a programmer-defined unit in THINK Pascal.
9. Which menu option in THINK Pascal is important to creating a unit?

10. How is the option **Build** different from the option **Check Syntax** or **Compile** in the **Run** menu of THINK Pascal?
11. What is the difference between the options **Add Window** and **Add File...** in the **Project** menu of THINK Pascal?
12. How can a unit be removed from the Project window?
13. What are the differences between a THINK Pascal library and a unit?
14. Explain the steps for building a programmer-defined library in THINK Pascal.
15. What purpose is served in having an interface file when building a programmer-defined library?
16. How can an external variable directive be used to avoid multiple declarations?
17. How can the **uses** clause be applied within the implementation section of a THINK Pascal unit?
18. What value is there in having the implementation section of a unit allow the **uses** clause?
19. How can **uses** clauses be propagated through several units? What advantages exist in allowing units to be propagated through the interface-part of a unit? What disadvantages exist in such a concept?
20. What is meant by the term *information hiding*? How do THINK Pascal units support this concept?
21. What is meant by the term *predefined libraries*?
22. What is the purpose of the library file `Runtime.lib`?
23. What is the purpose of the library file `Interface.lib`?
24. List the names of files that are included in the file `Interface.lib`.
25. If `SANE.p` is not one of the toolbox files incorporated in `Interface.lib`, how can this unit be borrowed from the Macintosh Toolbox when routines from the SANE Toolbox are utilized?
26. In the Macintosh system what is the difference between stack-based and register-based routines?
27. Describe how a THINK Pascal program can be established as an application.
28. How does THINK Pascal allow large applications, in which jump tables as large as 256K bytes are present?
29. What purpose is served by setting the **Smart Link** option in the dialog window for building an application?
30. What is the advantage in converting a THINK Pascal program into an application?
31. What purpose is served by having a profiler?
32. In THINK Pascal, how is the profiler implemented?
33. What purpose is served in declaring the unit `Profiler` using a **uses** clause?
34. Explain the purpose of the profiler variable `%_PTrace`.
35. Define the purpose of the following profiler procedures and function: `DumpProfile`, `DumpProfileToFile`, `ResetProfile`, `TerminateProfile`, and `InitProfile`?
36. What purpose is served by having a debugger?
37. How is the LightsBug window established in THINK Pascal?
38. Explain the purpose of the four window panes and the nine icons in the LightsBug window.

39. How can the **Stops In** option from the **Debug** menu be used to examine the execution of a program unit?
40. What purpose is served by the **Step Over** and **Step Into** options in the **Run** menu?
41. How can a THINK Pascal program be halted so that a programmer can examine the values of actual parameters for any Macintosh Toolbox routine being invoked?
42. How can the routines from unit `Profiler` be used to measure the time for invoking, but not executing, the body of a programmer-defined routine?
43. How can the routines from unit `Profiler` be used to measure the execution time of Macintosh Toolbox routines?
44. Is the following unit syntactically as well as semantically correct? Explain your answer.

```
unit Test_Ideas;
interface
var
   Exponent : integer;
implementation
   function Power( X, Y : integer ) : integer;
   begin
      if ( X = 0 ) and ( Y = 0 ) then
         Power := -1
      else
         if ( Y < 0 ) then
            Power := -2
         else
            begin
               Exponent := 1 ;
               while ( Y > 0 ) do
                  begin
                     Exponent := Exponent * X ;
                     Y := succ(Y);
                  end;
               Power := 0;
   end; { Power }
end.
```

45. How would the following unit be used to initialize the variables defined within its interface-part?

```
unit Major_Components;
interface
var
   Major_Capacity : real;
   Flow_Level : real;
   Temperature_Level : real ;
implementation
   procedure Initialize_Major_Components;
   begin
      Major_Capacity := 2000.00;
      Flow_Level := 150.00;
      Temperature_Level := 199.99;
   end; { Initialize_Major_Components }
end.
```

46. With the **USES Extension** off, what happens when the following program and units are built within the same project? How can the program and/or unit(s) be corrected to display the values of X and Y?

```
program Main( Input, Output );
   uses
      Unit_A;
begin
   writeln( X, Z );
end.
unit Unit_A;

interface
   uses
      Unit_B;
implementation
   const
      X = Z ;
end.

unit Unit_B;
interface
   const
      Z = 345;
implementation
end.
```

PROGRAMMING EXERCISES

1. Build and test a programmer-defined unit that will compute the following logarithmic and exponentiation functions:

function `Log_10( M : real; var Log_Value : real ): integer;`
This function computes the common logarithm for real number M greater than zero and returns this value through Log_Value. If the value of M is less than or equal to zero, this function returns –1 through the function name; otherwise it returns a value of zero.

function `Integer_Power( X, Y integer; var Power : integer ) : integer;`
This function will compute X raised to power Y for two integer numbers and will return the exponentiated value through Power under

the condition that Y is zero or positive. For the following conditions, the values that are indicated will be returned through the name of the function `Integer_Power`:

(a) If X and Y are both zero, return a value of −1 for the function.
(b) If Y is negative, return a value of −2 for the function. Otherwise, return a value of zero.

function `Real_Power( X,Y real;` **var** `Power : real ): integer;`
This function will compute X raised to power Y for two real numbers and return a real exponentiated value through `Power`. For the following conditions, the values indicated will be returned through the name of the function `Real_Power`:

(a) If X and Y are both zero, return a value of −1 for the function.
(b) If X is zero and Y is nonzero, return a value of −2 for the function.
(c) If X is negative and Y is less than 1, return a value of −3 for the function. Otherwise, return a value of zero.

2. Build and test a programmer-defined unit that computes the following trigonometric functions:

   `tan(X), cot(X), csc(X), sec(X),`
   where `cot(X) = 1/tan(X), sec(X) = 1/cos(X),` and `csc(X) = 1/sin(X).`

3. Build and test a programmer-defined unit for performing the following complex operations:

   (a) **procedure** `Add_Complex(` **var** `X : Complex_Rect; Y, Z : Complex_Rect );`
   This procedure will add the complex number Y to the complex number Z and return the the result through X.
   (b) **procedure** `Sub_Complex(` **var** `X : Complex_Rect; Y, Z : Complex_Rect);`
   This procedure will subtract the complex number Z from the complex number Y and return the the result through X.
   (c) **procedure** `Mult_Complex(` **var** `X : Complex_Rect; Y, Z : Complex_Rect);`
   This procedure will multiply the complex number Y by the complex number Z and return the the result through X.
   (d) **procedure** `Div_Complex(` **var** `X : Complex_Rect; Y, Z : Complex_Rect);`
   This procedure will divide the complex number Y by the complex number Z and return the the result through X.

(e) **procedure** `Conjugate( var X : Complex_Rect, Y : Complex_Rect );`
This procedure will take the conjugate of the complex number `Y` and return the the result through `X`.

(f) **procedure** `Modulus( var R : real; Y : Complex_Rect );`
This procedure will take the modulus of the complex number `Y` and return the the result through `R`.

(g) **procedure** `Write_Complex( X : Complex_Rect );`
This procedure will display to the Text window a complex number, given by `X`, in the form
$a + j\,b$.

(h) **procedure** `Read_Complex( var X : Complex _Rect );`
This procedure will read a complex number typed at the keyboard in the form $a + j\,b$ and assign this to `X`. The conjugate of

$$a + jb = a - jb,$$

while the modulus

$$a + jb = (a^2 + b^2)^{1/2}$$

4. Finish building the units given in either Figure 8.13 or Figure 8.14 by first using stubs and then replacing each stub one routine at a time. Each time a stub is replaced, test it to see that it works properly and that the program is satisfying its requirements.

5. Convert the unit in Exercise 3 into a library, and use your test program from Exercise 3 to test your library. In defining a library, you will be required to define an interface file.

6. Convert the unit in Exercise 2 into a library, and use your test program from Exercise 2 to test your library.

7. Convert Exercise 18 of Chapter 7 into an application.

8. Convert Exercise 20 of Chapter 7 into an application.

9. For analyzing `Tutor_System` in Section 8.8, implement the test program to verify the results of the profiler.

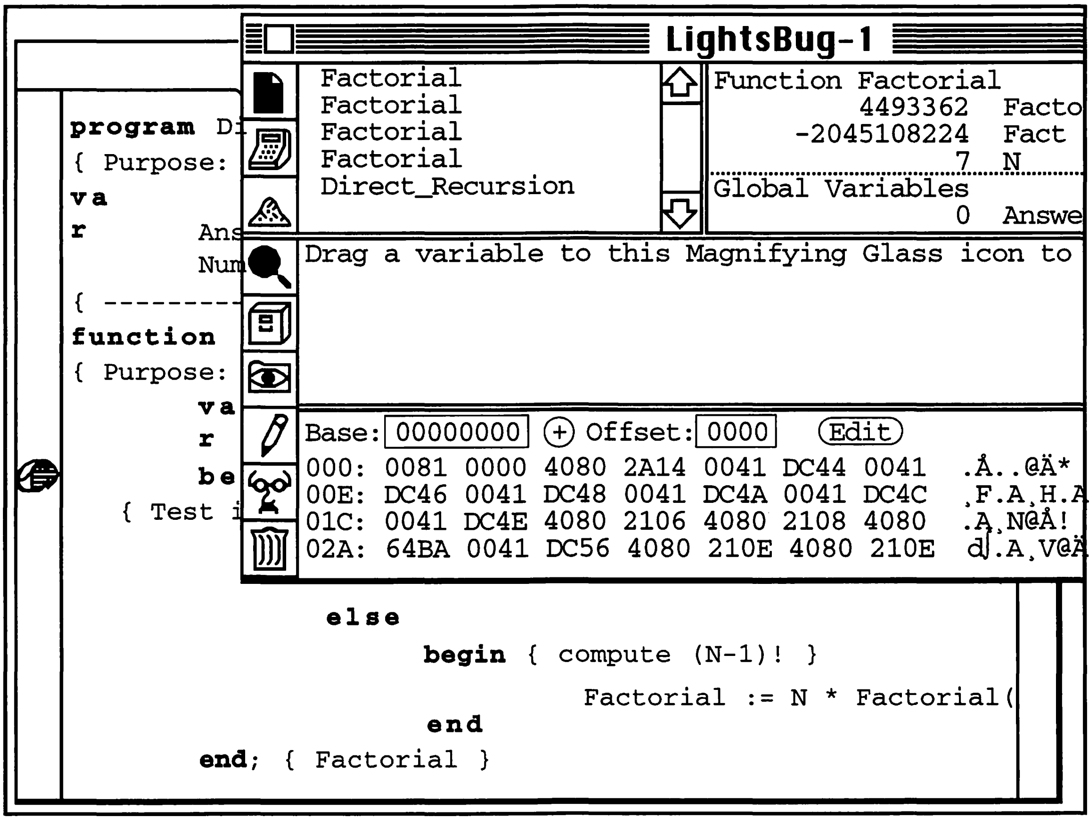
10. Extend the example in Section 8.8 by showing how the routines `InitProfiler`, `DumpProfileToFile`, and `Terminate-Profile` can be used.

11. Combine the routines from the following program, `Direct_Recursion`, and `Indirect_Recursion` in Chapter 7. Add a new

function for computing the factorial of an integer number iteritively. You may find it convenient to name the three factorial functions Factorial_1, Factorial_2, and Factorial_3, respectively. Now apply the variable %_PTrace, and select the appropriate routines from unit Profiler for reporting on the total execution time of each factorial function. Compare the execution times for these three factorial functions by selecting various integer numbers entered from the keyboard. You might try modifying your program and collecting data for plotting the total execution times versus the value of the number for integers in the range 1 through 20. From what you observe, which function is most efficient in terms of execution time? Can you interpret what your test results are telling you?

```
program Direct_Recursion (Input, Output);
{ Purpose:  This program is an example of direct recursion. }
   var
      Answer: longint;
      Number: integer;
{ ----------------------------------------------------------------- }
 function Factorial (N: integer): longint;
{ Purpose: Computes the factorial of a positive integer N. }
   var
      Fact: longint;
   begin
   { Test if a trivial case exists. }
      if N = 0 then
         Factorial := 1
      else
         begin { compute (N-1)! }
            Factorial := N * Factorial(N - 1);
         end
   end; { Factorial }
{ ================= Body of the main program. ================= }
begin { Body of the main program. }
   ShowText;
{ Prompt user for a nonnegative integer number. }
   write('Enter a positive whole number: ');
   readln(Number);
   if Number >= 0 then
      begin
         Answer := Factorial(Number);
         writeln(Answer : 11)
      end;
end.
```

12. Using LightsBug, duplicate the snapshot in Figure 8.21 by inserting the appropriate stops in sample program Direct_Recursion, given in Exercise 11. Use the value 10 for Number.

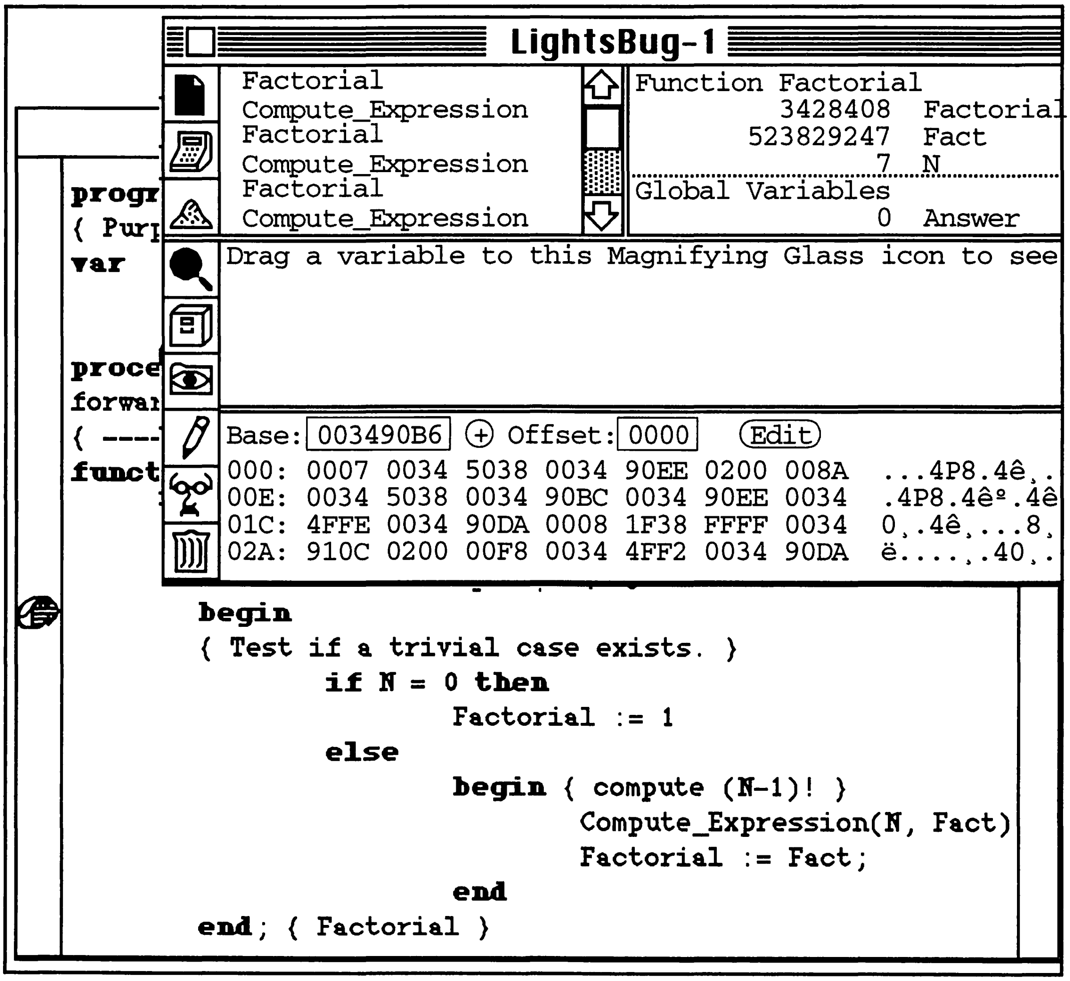


```
Factorial
Factorial
Factorial
Factorial
Direct_Recursion

Function Factorial
          4493362  Facto
      -2045108224  Fact
                7  N
Global Variables
                0  Answe

000: 0081 0000 4080 2A14 0041 DC44 0041  .Å..@Ä*
00E: DC46 0041 DC48 0041 DC4A 0041 DC4C  ¸F.A¸H.A
01C: 0041 DC4E 4080 2106 4080 2108 4080  .A¸N@Å!
02A: 64BA 0041 DC56 4080 210E 4080 210E  dJ.A¸V@Ä
```

```
program Di
{ Purpose:
va
r        Ans
         Num
{ --------
function
{ Purpose:
        va
         r
        be
   { Test i
           else
              begin { compute (N-1)! }
                        Factorial := N * Factorial(
              end
        end; { Factorial }
```

Figure 8.21
LightsBug window for examining execution of sample program Direct_Recursion.

13. Using LightsBug, duplicate the snapshot in Figure 8.22 by inserting the appropriate stops in sample program `Indirect_Recursion` given in Chapter 7. Use the value 10 for `Number`.

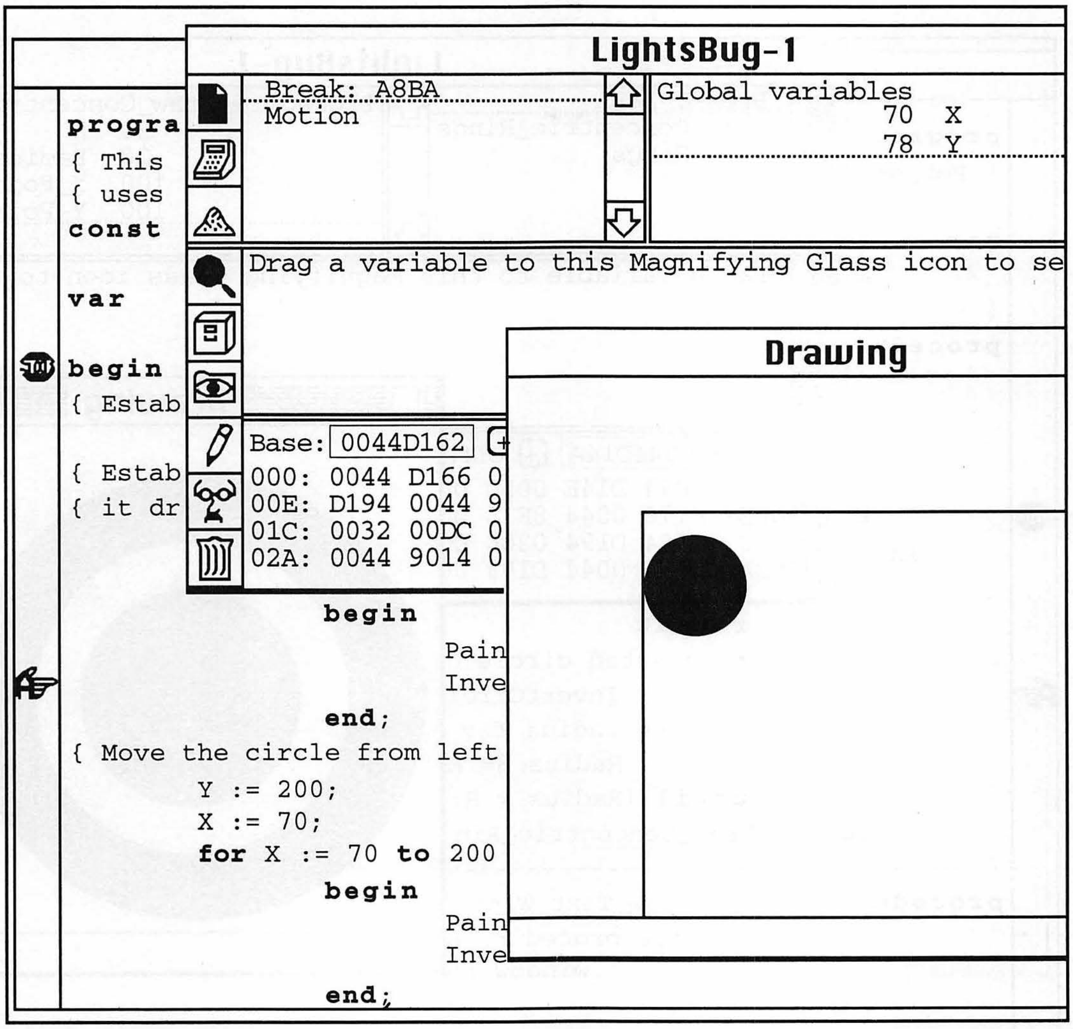


```
LightsBug-1

Factorial                      Function Factorial
Compute_Expression                      3428408  Factorial
Factorial                             523829247  Fact
Compute_Expression                            7  N
Factorial                      Global Variables
Compute_Expression                            0  Answer

Drag a variable to this Magnifying Glass icon to see

Base: 003490B6  ⊕ Offset: 0000   (Edit)
000: 0007 0034 5038 0034 90EE 0200 008A  ...4P8.4ê¸.
00E: 0034 5038 0034 90BC 0034 90EE 0034  .4P8.4êº.4ê
01C: 4FFE 0034 90DA 0008 1F38 FFFF 0034  0¸.4ê¸...8¸
02A: 910C 0200 00F8 0034 4FF2 0034 90DA  ë....¸.40¸.
```

```
progr
{ Pur
var

proce
forwa
{ ----
funct

begin
{ Test if a trivial case exists. }
        if N = 0 then
                Factorial := 1
        else
                begin { compute (N-1)! }
                        Compute_Expression(N, Fact)
                        Factorial := Fact;
                end
end; { Factorial }
```

Figure 8.22 LightsBug window for examining the execution of sample program `Indirect_Recursion`.

14. Using LightsBug, duplicate the snapshot in Figure 8.23 by inserting the appropriate stop(s) as well as setting the option **Break at A-Traps** from the **Debug** menu for the sample program `Motion` given in Chapter 6.



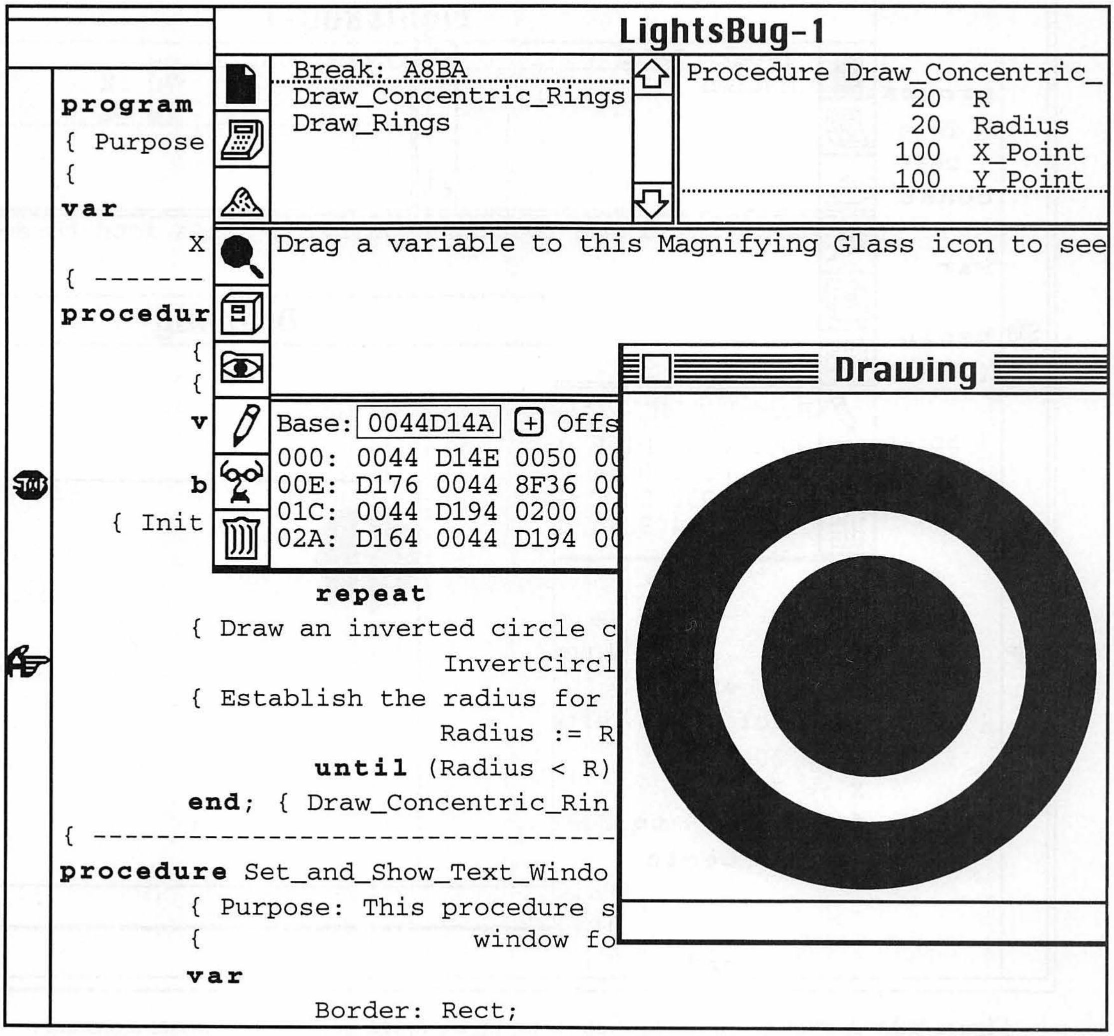
```
70  X
78  Y
```

```
000: 0044 D166 0
00E: D194 0044 9
01C: 0032 00DC 0
02A: 0044 9014 0
```

```
progra
{ This
{ uses
const

var

begin
{ Estab

{ Estab
{ it dr

                begin
                        Pain
                        Inve
                end;
{ Move the circle from left
        Y := 200;
        X := 70;
        for X := 70 to 200
                begin
                        Pain
                        Inve
                end;
```

Figure 8.23 A snapshot of the Macintosh screen for program `Motion`, where stops are inserted and `Break at A-Traps` is set.

15. Using LightsBug, duplicate the snapshot in Figure 8.24 by inserting the appropriate stop(s) as well as setting the option **Break at A-Traps** from the **Debug** menu for the sample program `Draw_Rings` given in Chapter 7.



```
Break: A8BA
Draw_Concentric_Rings
Draw_Rings

Procedure Draw_Concentric_
      20  R
      20  Radius
     100  X_Point
     100  Y_Point

Drag a variable to this Magnifying Glass icon to see

Base: 0044D14A  Offs
000: 0044 D14E 0050 00
00E: D176 0044 8F36 00
01C: 0044 D194 0200 00
02A: D164 0044 D194 00
```

```
program
{ Purpose
{
var
          X
{ -------
procedur
        {
        {
        v

        b
    { Init

            repeat
        { Draw an inverted circle c
                        InvertCircl
        { Establish the radius for
                        Radius := R
            until (Radius < R)
        end; { Draw_Concentric_Rin
{ -------------------------------
procedure Set_and_Show_Text_Windo
        { Purpose: This procedure s
        {                window fo
        var
                Border: Rect;
```

Figure 8.24 A snapshot of the Macintosh screen for program `Draw_Rings`, where stops are inserted and the `Break at A-Traps` is set.

Chapter 9

Structured Data Types

OBJECTIVES

After completing Chapter 9, you will know the following:

1. The characteristics and use of the array as a data structure. This includes one-dimensional arrays as well as multidimensional arrays.
2. The applications of arrays, particularly in sorting and searching routines. Sorting algorithms discussed include the bubble sort, insertion sort, Shellsort, and quicksort. Search algorithms include linear and binary search.
3. The record as a nonhomogeneous data structure, including the creation and access of records and their fields. This study is extended to consider variant records having both fixed and variant fields.
4. The creation of and use of the Pascal set. The set operations and set comparisons are discussed.
5. The use of the data entry technique called *lazy input*.
6. The use of the packed array of characters. This data structure is compared to the `string` type.

9.1 CONCEPT OF AN ARRAY AS A HOMOGENEOUS STRUCTURE

When solving problems, we often need to go beyond the use of simple variables and consider ways to structure `ordinal` data types. When developing a program, it is a common practice to use a variable to represent a block of information. Figure 9.1 shows a one-dimensional array called `Table`, consisting of a single column with several rows of data, each row represented by a unique index number. It can also be viewed as a single row consisting of several columns of data. In Pascal declaring an identifier as a one-dimensional array requires the following syntax:

```
var
   Variable_Name : array [ Low_Index..High_Index ] of data_type;
```

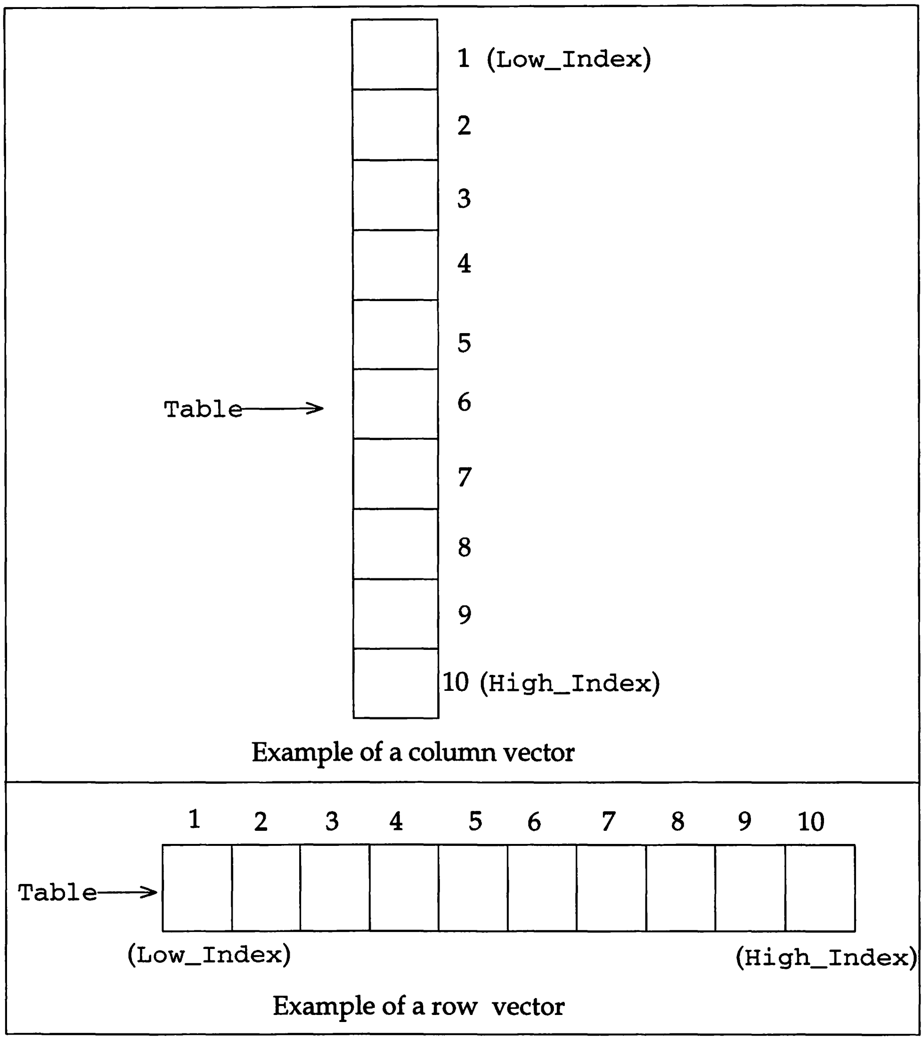


Figure 9.1 A one-dimensional array represented first as a column vector and then as a row vector.

The bounds of the array are expressed by the subrange Low_Index .. High_Index; the constant Low_Index is the lowest index value for the array variable, and the constant High_Index is the highest index value. Data_type can be an ordinal type, string , real, or even another structure type, such as an array. In Section 9.4 we will consider the case of the data type being an array. The following

Pascal statements show two different approaches to declaring the one-dimensional array of Figure 9.1:

```
var
     Table : array[1..100] of real;
```

or

```
const
      Low_Index = 1;
      High_Index = 100;
var
     Table : array[Low_Index..High_Index] of real;
```

The number of elements within a one-dimensional array is given by the value of the expression

```
High_Index - Low_Index + 1
```

In addition, an array is homogeneous; that is, each element of the array has the same data type. Pascal requires that the values of the constants `Low_Index` and `High_Index` be known when the declaration statement is translated. Otherwise, a translation error is reported specifying that there is an undeclared identifier or an improper constant.

Although it is convenient at times to reference the complete array by using its declared name, an element within a one-dimensional array is referenced or accessed by using a subscripted variable. A subscripted variable is composed of the array variable name followed by a subscript (an *integer* expression) contained within square brackets. For example, we reference the *i*th element of our array `Table` by the subscripted variable `Table[i]` (read "Table sub i"). Keep in mind that the subscript itself must have a value within the subrange `Low_Index..High_Index`. If during execution a subscripted variable is found to be out of range, the program is interrupted, and an error message is displayed indicating that the value of a subscript is out of range. Although this error will occur at execution time, the fault generally lies with the design of the algorithm, not the program code.

Subscripted variables are like other simple variables. For instance, a subscripted variable can be assigned a value through execution of a `read` or `readln` command or through an assignment statement. A subscripted variable can be part of an expression, an expression by itself, or an argument within an actual parameter list. For example, consider the following Pascal program for adding a set of positive numbers after they have been entered from the keyboard and stored in a one-dimensional array called `Storage`. The total set of numbers read from the keyboard may never exceed 100.

```
program Sum_Numbers(input, output);
{ Purpose:  This program will sum N positive numbers entered }
{           from the keyboard. The value of N can never exceed }
{           100. }
     var
          Storage : array[1..100] of real;
         Counter, N : integer;
         Sum : real;
begin
{ Prompt the user for the total count of positive numbers to }
{ be added. }
   repeat
      HideAll;
      ShowText;
      Page;
      writeln(' How many positive numbers will you be adding? ');
      write(' Number entered must be in the range of 1 to 100: ');
      readln( N );
   until( N > 0 ) and ( N <= 100 );
   writeln;
{ Enter N positive numbers from the keyboard. }
   for Counter := 1 to N do
      begin
         write(' Enter your next positive number: ');
         readln( Storage[Counter] );
      end;
   writeln;
{ Compute the sum of all N positive numbers. }
   Sum := 0.0;
   for Counter := 1 to N do
      Sum := Sum + Storage[Counter];
{ Report the sum of all N positive numbers. }
   writeln(' Sum of ', N:3 , ' positive numbers is ', Sum:8:3 )
end.
```

This program is not to be admired for its brevity, but it does provide a simple example of the use of a subscripted variable. `Sum_Numbers` will execute under both Macintosh and THINK Pascal.

As an additional example of a one-dimensional array, assume that we need to record the noontime temperatures for each day of the week, Sunday through Saturday. The noontime temperatures are to be stored in a one-dimensional `real` array called `Temperature`, with each day of the week represented by an index number. The array `Temperature` will be a seven-element array having a subscript range from 1 through 7. Therefore, the noontime temperature for Sunday is represented by the subscripted variable `Temperature[1]`, and the noontime temperature for Saturday is represented by `Temperature[7]`. Assume that our algorithm calls for computing and reporting the mean noontime temperature, the days of the week with temperatures greater than the computed mean, and the days of the week with temperatures less than the computed mean. Further, if the noontime temperature never differed from the mean, the algorithm is required to report a message indicating this fact. All inputs and outputs involved in referencing a day of the week will use the actual name instead of an index value.

Initially the algorithm is developed by dividing it into three basic parts: input, computation, and output. The following represents the initial steps for our algorithm:

Input information

1. Enter the noontime temperatures for the days Sunday through Saturday.

Perform Computations

2. Compute the mean noontime temperature.
3. Determine the days of the week having noontime temperatures above the mean.
4. Determine the days of the week having noontime temperatures below the mean.

Output information

5. Report the mean noontime temperature.
6. If any days were above the mean, report those days; else report a message that there were no days above the mean.
7. If any days were below the mean, report those days; else report a message that there were no days below the mean.

For input we need only the variable called `Temperature`, while for output we will need three variables: `Mean_Temperature` and two additional tables called `Days_Above` and `Days_Below`. Figure 9.2 shows the structure representing all three tables. Notice that the two additional tables for storing the weekdays as strings have only six elements. Why? Because the maximum number of days having either temperatures above or below the mean is six, no more than six elements are necessary. Three counters will be required: `Index`, to provide the index position within an array; `Number_Days_Above`, for counting the number of days above the mean; and `Number_Days_Below`, for counting the number of days below the mean.

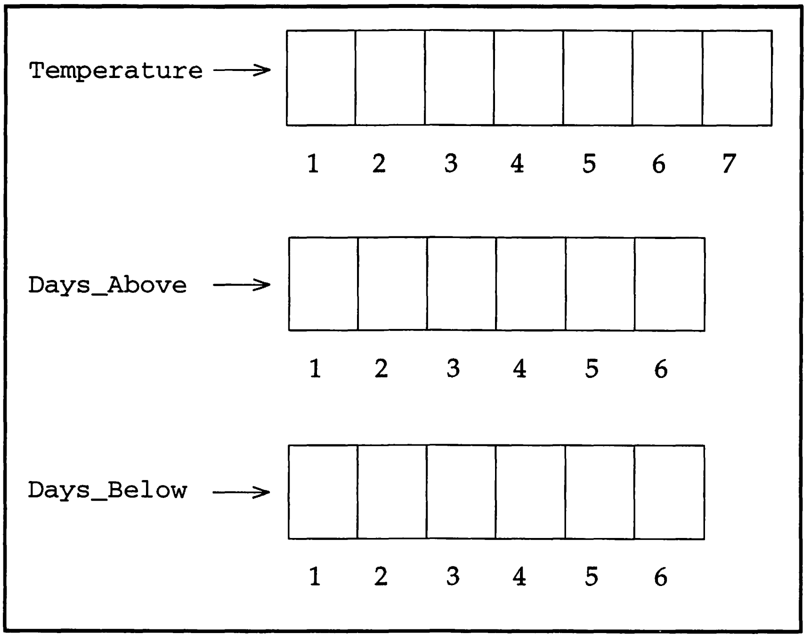


Figure 9.2 Representation of three arrays: `Temperature`, `Days_Above`, and `Days_Below`.

Let us now move to the next level of abstraction for our problem by refining the steps of our algorithm. In writing this algorithm, we treated all arrays as global variables. Although this may contradict our use of parameters discussed in Section 7.3, we say more in Section 9.2 about the passing of arrays to formal parameters. Each step is represented by a call to a procedure:

```
Algorithm Days_of_Week;
{ Purpose: This algorithm computes the number of days above and
   the number of days below the mean noontime temperature for the
   days from Sunday through Saturday and reports the day of each
   occurrence.}
begin
{ Enter noontime temperatures for Sunday through Saturday. }
   Input_Temperatures;
{ Compute the mean noontime temperature. }
   Compute_Mean_Temperature(Mean_Temperature);
{ Determine the days of the week having noontime temperatures
   above the mean temperature. }
   Days_Above_Mean(Mean_Temperature, Number_Days_Above);
{ Determine the days of the week having noontime temperatures
   below the mean temperature. }
   Days_Below_Mean(Mean_Temperature, Number_Days_Below);
{ Report the mean noontime temperature. }
   Report_Mean_Temperature(Mean_Temperature);
{ Report the days of the week having temperatures above the mean
   temperature. }
   Report_Days_Above(Number_Days_Above);
{ Report the days of the week having temperatures below the mean
   temperature. }
   Report_Days_Below(Number_Days_Below);
end. { Days_of_Week }
```

Let us increase our level of abstraction by writing procedures for each of the steps in the algorithm.

```
procedure Input_Temperatures;
{ Purpose: This procedure reads a temperature typed from the
   keyboard and stores it in an array called Temperature. The
   array Temperature is global to this procedure.  This procedure
   requires a function called Name_of_Day for returning a string
   value representing the day of the week. }
begin
{ Enter noon temperatures for Sunday through Saturday. }
   for  Index <-- 1 to 7 do
      begin
      { Prompt the user for the next day's noon temperature. }
         write('Enter temperature for ',Name_of_Day( Index ),':');
         readln( Temperature[ Index ] )
      end;
end; { Input_Temperatures }
```

```
procedure Compute_Mean_Temperature;
{ Purpose: This procedure computes the mean temperature from the
   array Temperature. The identifier Mean_Temp is a parameter for
   returning the value for the mean temperature. The array
   Temperature is global to this procedure. }
begin
{ Compute the total of all noontime temperatures. }
   Partial_Sum <-- 0.0;
   for Index <-- 1 to 7  do
      Partial_Sum <-- Partial_Sum + Temperature[ Index ] ;
{ Compute the mean temperature. }
   Mean_Temp <-- Partial_Sum/7;
end; { Compute_Mean_Temperature }

procedure  Days_Above_Mean;
{ Purpose: This procedure determines the days when the noontime
   temperature is above the mean temperature. It requires one
   value parameter, Mean_Temp, representing the mean temperature,
   and one variable parameter, called Count_Above, representing
   Number_Days_Above. Two global arrays used by this procedure are
   Temperature and Days_Above. This procedure requires a function
   called Name_of_Day for returning a string value representing
   the day of the week. }
begin
{ Determine the days of the week having noon temperatures above
   the mean temperature. }
   Count_Above <-- 0;
   for Index <-- 1 to 7 do
      begin
         if Temperature[ Index ] > Mean_Temp then
            begin
               Count_Above <-- succ( Count_Above );
               Days_Above[ Count_Above ] <-- Name_of_Day( Index )
            end;
      end;
end; { Days_Above_Mean }

procedure  Days_Below_Mean;
{ Purpose: This procedure determines the days when the noontime
   temperature is below the mean temperature. It requires one
   value parameter, Mean_Temp, representing the mean temperature,
   and one variable parameter, called Count_Below, representing
   Number_Days_Below. Two global arrays used by this procedure are
   Temperature and Days_Below.  This procedure requires a function
   called Name_of_Day for returning a string value representing
   the day of the week. }
begin
{ Determine the days of the week having noon temperatures below
   the mean temperature. }
   Count_Below <-- 0;
   for Index <-- 1 to 7 do
```

```
      begin
         if Temperature[ Index ] < Mean_Temp then
            begin
               Count_Below <-- succ( Count_Below );
               Days_Below[ Count_Below ] <-- Name_of_Day( Index )
            end;
      end;
end; { Days_Below_Mean }

procedure Report_Mean_Temperature;
{ Purpose: This procedure displays the value of the mean
   temperature to the Text window. It requires only a value
   parameter, Mean_Temp, representing the mean temperature. }
begin
{ Report the mean temperature. }
   writeln( 'Mean noon temperature: ', Mean_Temp);
end; { Report_Mean_Temperature };

procedure Report_Days_Above;
{ Purpose: This procedure will report the days of the week when
   the noon temperature is above the mean. It requires only a
   value parameter, Count_Above, representing the value for
   Number_Days_Above. This procedure also requires a global array
   called Days_Above. }
begin
{ Report the days of the week having temperatures above the mean }
   temperature. }
   if  Number  > 0 then
      begin
         write( ' Days with temperatures above the mean ');
         write( ' temperature: ');
         for Index <-- 1 to Count_Above do
            writeln( Days_Above[ Index ] )
      end
   else write( ' No noon temperature above the mean. ');
end; { Report_Days_Above }

procedure Report_Days_Below;
{ Purpose: This procedure will report the days of the week when
   the noon temperature is below the mean. It requires only a
   value parameter, Count_Below, representing the value for
   Number_Days_Below. This procedure also requires a global array
   called Days_Below. }
begin
{ Report the days of the week having temperatures below the mean
   temperature. }
   if Count_Below > 0 then
      begin
         write( 'Days with temperatures below the mean ');
         write( 'temperature:');
         for Index <-- 1 to Count_Below do
```

```
            writeln( Days_Below[ Index ] )
      end
   else write('No noon temperatures below the mean. ');
end; { Report_Days_Below }
```

As stated in the comments, our example requires the use of a special function called Name_of_Day for returning a string value representing the day of the week. The definition of the function Name_of_Day is as follows:

```
function Name_of_Day;
{ Purpose: This function computes the day of week and returns this
   as a string value. It requires only a value parameter called
   Number. Number represents the value of Index in the procedures
   Days_Above_Mean and Days_Below_Mean. }
begin
   case Number of
      1 :  Name_of_Day <-- 'Sunday';
      2 :  Name_of_Day <-- 'Monday';
      3 :  Name_of_Day <-- 'Tuesday';
      4 :  Name_of_Day <-- 'Wednesday';
      5 :  Name_of_Day <-- 'Thursday';
      6 :  Name_of_Day <-- 'Friday';
      7 :  Name_of_Day <-- 'Saturday'
   end { case }
end; { Name_of_Day }
```

The program Days_of_Week follows. An additional procedure has been added for hiding all of the Macintosh windows, setting the boundaries of the Text window, and then opening the Text window. This Macintosh Pascal program will execute under THINK Pascal with no change other than removal of the **uses** clause.

```
program Days_of_Week(input, output);
{ Purpose:  This program computes the number of days above and }
{           the number of days below the mean noontime }
{           temperature for the days from Sunday through }
{           Saturday and reports the day of each occurrence. }
   uses
      QuickDraw1;
   var
      Temperature : array[1..7] of real;
      Number_Days_Above, Number_Days_Below : integer;
      Mean_Temperature : real;
      Days_Above, Days_Below : array[1..6] of string;
{ ********************************************************** }
   procedure Set_Text_Window;
   { Purpose:  This procedure sets the boundaries for the Text }
   {           window. }
      var
         Window : Rect;
   begin
   { Hide all Macintosh Pascal windows and then open the Text }
   { window. }
      HideAll;
      SetRect(Window, 0, 50, 500, 300);
      SetTextRect(Window);
      ShowText;
   end;
{ ******************************************************** }
  function Name_of_Day (Number : integer) : string;
  { Purpose:  This function returns the day of the week as a }
  {           string value. It requires only a value parameter }
  {           called Number. Number represents the value of }
  {           Index in the procedures Days_Above_Mean and }
  {           Days_Below_Mean. }
  begin
     case Number of
        1 :  Name_of_Day := 'Sunday';
        2 :  Name_of_Day := 'Monday';
        3 :  Name_of_Day := 'Tuesday';
        4 :  Name_of_Day := 'Wednesday';
        5 :  Name_of_Day := 'Thursday';
        6 :  Name_of_Day := 'Friday';
        7 :  Name_of_Day := 'Saturday'
     end { case }
  end;
{ ******************************************************** }
  procedure Input_Temperatures;
  { Purpose:  This procedure reads a temperature typed from }
  {           the keyboard and stores it in an array called }
  {           Temperature. The array Temperature is global to }
  {           this procedure. This procedure requires a }
  {           function called Name_of_Day for returning a }
  {           string value representing the day of the week. }
     var
        Index : integer;
  begin
  { Enter noon temperatures for Sunday through Saturday. }
     for Index := 1 to 7 do
        begin
        { Prompt the user for the next day's noon temperature. }
           write('Enter temperature for ', Name_of_Day(Index),
                    ': ');
           readln(Temperature[Index])
        end;
     writeln;
  end;
{ ******************************************************** }
  procedure Compute_Mean_Temperature (var Mean_Temp : real);
  { Purpose:  This procedure computes the mean temperature }
  {           from the array Temperature. The identifier }
  {           Mean_Temp is a parameter for returning the value }
  {           for mean temperature. The array Temperature is }
  {           global to this procedure. }
     var
        Partial_Sum : real;
        Index : integer;
  begin
  { Compute the total of all noontime temperatures. }
     Partial_Sum := 0.0;
     for Index := 1 to 7 do
        Partial_Sum := Partial_Sum + Temperature[Index];
  { Compute the mean temperature. }
     Mean_Temp := Partial_Sum / 7;
  end;
{ ******************************************************** }
  procedure Days_Above_Mean (Mean_Temp : real;
                             var Count_Above : integer);
  { Purpose:  This procedure determines the days when the }
  {           noontime temperature is above the mean }
  {           temperature. It requires one value parameter, }
  {           Mean_Temp, representing the mean temperature, }
  {           and one variable parameter, called Count_Above, }
  {           representing Number_Days_Above. Two global arrays }
  {           used by this procedure are Temperature and }
  {           Days_Above. This procedure requires a function }
  {           called Name_of_Day for returning a  string value }
  {           representing the day of the week. }
     var
        Index : integer;
  begin
  { Determine the days of the week having noon temperatures }
  { above the mean temperature. }
     Count_Above := 0;
     for Index := 1 to 7 do
        begin
           if Temperature[Index] > Mean_Temp then
              begin
                 Count_Above := succ(Count_Above);
                 Days_Above[Count_Above] := Name_of_Day(Index)
              end;
        end;
  end;
{ ******************************************************** }
  procedure Days_Below_Mean (Mean_Temp : real;
                             var Count_Below : integer);
  { Purpose:  This procedure determines the days when the }
  {           noontime temperature is below the mean }
  {           temperature. It requires one value parameter, }
  {           Mean_Temp, representing the mean temperature, }
  {           and one variable parameter, called Count_Below, }
  {           representing Number_Days_Below. Two global arrays }
  {           used by this procedure are Temperature and }
  {           Days_Below. This procedure requires a function }
  {           called Name_of_Day for returning a string value }
  {           representing the day of the week. }
     var
        Index : integer;
  begin
  { Determine the days of the week having noon temperatures }
  { below the mean temperature. }
     Count_Below := 0;
     for Index := 1 to 7 do
        begin
           if Temperature[Index] < Mean_Temp then
              begin
                 Count_Below := succ(Count_Below);
                 Days_Below[Count_Below] := Name_of_Day(Index)
              end;
        end;
  end;
 { ********************************************************** }
  procedure Report_Mean_Temperature (Mean_Temp : real);
  { Purpose:  This procedure displays the value of the mean }
  {           temperature to the Text window. It requires only }
  {           a value parameter, Mean_Temp, representing }
  {           the mean temperature. }
  begin
  { Report the mean temperature. }
     writeln('Mean noon temperature: ', Mean_Temp : 5 : 1);
     writeln;
  end;
{ ********************************************************** }
  procedure Report_Days_Above (Count_Above : integer);
  { Purpose:  This procedure will report the days of the week }
  {           when the noontime temperature is above the mean. }
  {           It requires only a value parameter, }
  {           Count_Above, representing the value for }
  {           Number_Days_Above. This procedure also requires }
  {           a global array called Days_Above. }
     var
        Index : integer;
  begin
  { Report the days of the week having temperatures above the }
  { mean temperature. }
     if Count_Above > 0 then
        begin
           write(' Days with temperatures above the mean ');
           writeln('temperature: ');
           for Index := 1 to Count_Above do
              writeln(Days_Above[Index])
        end
     else
        write(' No noon temperature above the mean. ');
        writeln;
```

```
   end;
{ ********************************************************* }
   procedure Report_Days_Below (Count_Below : integer);
   { Purpose:  This procedure will report the days of the week }
   {           when the noontime temperature is below the mean. }
   {           It requires only a value parameter, Count_Below, }
   {           representing the value for Number_Days_Below. }
   {           This procedure also requires a global array }
   {           called Days_Below. }
      var
         Index : integer;
   begin
   { Report the days of the week having temperatures below the }
   { mean  temperature. }
      if Count_Below > 0 then
         begin
            write('Days with temperatures below the mean ');
            writeln('temperature:');
           for Index := 1 to Count_Below do
              writeln(Days_Below[Index])
         end
      else
      write('No noon temperatures below the mean. ');
      writeln;
   end;
{ ********************************************************* }
begin { Body of the main program. }
{ Open the Text window for viewing data. }
   Set_Text_Window;
{ Enter noontime temperatures for Sunday through Saturday. }
   Input_Temperatures;
{ Compute the mean noontime temperature. }
   Compute_Mean_Temperature(Mean_Temperature);
{ Determine the days of the week having noontime temperatures }
{ above the mean temperature. }
   Days_Above_Mean(Mean_Temperature, Number_Days_Above);
{ Determine the days of the week having noontime temperatures }
{ below the mean temperature. }
   Days_Below_Mean(Mean_Temperature, Number_Days_Below);
{ Report the mean noontime temperature. }
   Report_Mean_Temperature(Mean_Temperature);
{ Report the days of the week having temperatures above the mean }
{ temperature. }
   Report_Days_Above(Number_Days_Above);
{ Report the days of the week having temperatures below the mean }
{ temperature. }
   Report_Days_Below(Number_Days_Below);
end.
```

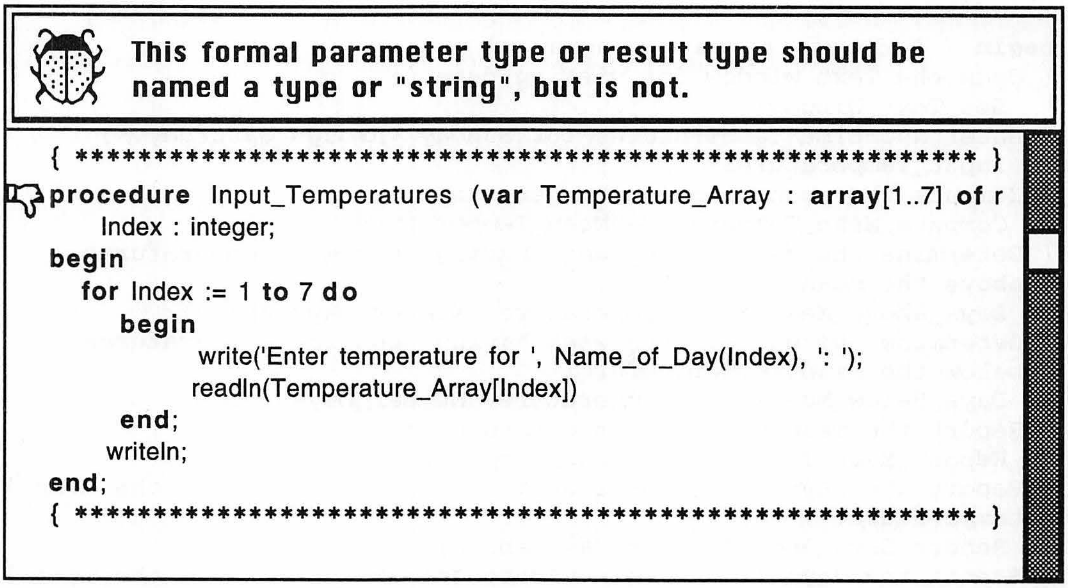
Notice that in our Pascal program `Days_of_Week`, square brackets, rather than curved brackets, surround the subscript for a subscripted variable. Curved brackets would create an ambiguity in the language between referencing a subscripted variable and calling on a procedure or function.

9.2 FORMAL PARAMETERS DECLARED AS ARRAY TYPES

As you may have noticed, the program Days_of_Week fails to take advantage of formal parameters declared as array types. For example, you can use the following to replace the procedure Input_Temperatures in the program Days_of_Week:

```
procedure Input_Temperatures(var Temperature_Array :
                                 array[1 .. 7] of real);
   var
      Index : integer;
begin
   for Index := 1 to 7 do
      begin
         write('Enter temperature for ',Name_of_Day(Index),': ' );
         readln( Temperature_Array[ Index ] )
      end;
end;
```

Entering this and applying the **Check** mode of the **Run** menu results in the dialog box shown in Figure 9.3.



This formal parameter type or result type should be named a type or "string", but is not.

```
  { ******************************************************* }
procedure Input_Temperatures (var Temperature_Array : array[1..7] of r
     Index : integer;
  begin
    for Index := 1 to 7 do
      begin
           write('Enter temperature for ', Name_of_Day(Index), ': ');
           readln(Temperature_Array[Index])
      end;
     writeln;
  end;
  { ******************************************************* }
```

Figure 9.3 The result of an attempt to use an array declaration for a formal parameter.

The error message indicates that the formal parameter Temperature_Array is improperly declared and that it must be declared with a type other than an array. This means that we must define a user-defined data type equivalent to the **array**[1..7]

of real in the type-declaration part of the main program. As explained in Section 2.5, the type-declaration part is composed of the reserved word **type** followed by a type declaration, which consists of an identifier, followed by = , followed by a data type, followed by a semicolon. For the program Days_of_Week, we require that the identifier Temperature be declared as a predefined data type called Table_of_Temperatures, and that this new data type be equated to an **array**[1..7] **of** real. The following Pascal code demonstrates this concept,

```
type
   Table_of_Temperatures = array[1..7] of real;
var
   Temperature : Table_of_Temperatures;
```

For the procedure-header of Input_Temperatures, the formal parameter Temperature_Array is now declared to be of type Table_of_Temperatures. The following statement corrects the syntax error in Figure 9.3.

```
procedure
   Input_Temperatures( var Temperature_Array :
                                       Table_of_Temperatures );
```

The program Days_of_Week_Revised demonstrates the concept, showing how arrays can be passed between the calling point of the procedure and the procedure-header, using the name of an array as an actual parameter. Program comments have been removed to reduced the length of the listing. Days_of_Week_Revised will execute under both Macintosh and THINK Pascal.

```
program Days_of_Week_Revised(input, output);
{ Purpose:  This program demonstrates the use of user-defined }
{           types for the purpose of passing arrays between }
{           actual and formal parameters. }
   type
      Table_of_Temperatures = array[1..7] of real;
      Table_of_Days = array[1..6] of string;
   var
      Temperature : Table_of_Temperatures;
      Days_Above, Days_Below : Table_of_Days;
      Number_Days_Above, Number_Days_Below: integer;
      Mean_Temperature : real;
{ ******************************************************** }
   procedure Set_Text_Window;
      var
         Window : Rect;
   begin
      HideAll;
      SetRect(Window, 0, 50, 500, 300);
      SetTextRect(Window);
      ShowText;
   end;
{ ******************************************************** }
   function Name_of_Day (Number : integer) : string;
```

```
  begin
      case Number of
        1 :  Name_of_Day := 'Sunday';
        2 :  Name_of_Day := 'Monday';
        3 :  Name_of_Day := 'Tuesday';
        4 :  Name_of_Day := 'Wednesday';
        5 :  Name_of_Day := 'Thursday';
        6 :  Name_of_Day := 'Friday';
        7 :  Name_of_Day := 'Saturday'
    end {case }
  end;
{ ******************************************************** }
  procedure Input_Temperatures (var Temperature_Array :
                                    Table_of_Temperatures);
     var
        Index : integer;
  begin
     for Index := 1 to 7 do
        begin
           write('Enter temperature for ', Name_of_Day(Index),
                        ': ');
           readln(Temperature_Array[Index])
        end;
     writeln;
  end;
{ ******************************************************** }
  procedure Compute_Mean_Temperature (Temperature_Array :
               Table_of_Temperatures; var Mean_Temp : real);
     var
        Partial_Sum : real;
        Index : integer;
  begin
     Partial_Sum := 0.0;
     for Index := 1 to 7 do
        Partial_Sum := Partial_Sum + Temperature_Array[Index] ;
     Mean_Temp := Partial_Sum / 7;
  end;
{ ******************************************************** }
  procedure Days_Above_Mean (Temperature_Array :
               Table_of_Temperatures;
               Mean_Temp  : real;  var Count_Above : integer;
               var Day_Array : Table_of_Days);
     var
        Index : integer;
  begin
     Count_Above := 0;
     for Index := 1 to 7 do
        begin
           if Temperature_Array[Index] > Mean_Temp then
              begin
                 Count_Above := succ(Count_Above);
```

```
                Day_Array[Count_Above] := Name_of_Day(Index)
             end;
         end;
     end;
{ ****************************************************** }
   procedure  Days_Below_Mean (Temperature_Array :
      Table_of_Temperatures; Mean_Temp : real; var Count_Below :
      integer;  var Day_Array : Table_of_Days);
      var
         Index : integer;
   begin
      Count_Below := 0;
      for Index := 1 to 7 do
         begin
            if Temperature_Array[Index] < Mean_Temp then
               begin
                  Count_Below := succ(Count_Below);
                  Day_Array[Count_Below] := Name_of_Day(Index)
               end;
         end;
   end;
 { ****************************************************** }
   procedure Report_Mean_Temperature (Mean_Temp : real);
   begin
      writeln('Mean noon temperature: ', Mean_Temp : 5 : 1);
      writeln;
   end;
{ ****************************************************** }
   procedure Report_Days_Above (Day_Array : Table_of_Days;
                                 Count_Above : integer);
      var
         Index : integer;
   begin
      if Count_Above > 0 then
         begin
            writeln(' Days with temperatures above the mean: ');
            for Index := 1 to Count_Above do
               writeln(Day_Array[Index])
         end
      else
         writeln('No noon temperature above the mean. ');
      writeln;
   end;
{ ****************************************************** }
   procedure Report_Days_Below (Day_Array : Table_of_Days;
                                       Count_Below : integer);
      var
         Index : integer;
   begin
      if Count_Below > 0 then
         begin
```

```
            writeln(' Days with temperatures below the mean: ');
            for Index := 1 to Count_Below do
                  writeln(Day_Array[Index])
         end
      else
         writeln('No noon temperatures below the mean. ');
      writeln;
   end;
{ ********************************************************* }
begin { Body of the main program. }
{ Open the Text window for viewing data. }
   Set_Text_Window;
{ Enter noontime temperatures for Sunday through Saturday. }
   Input_Temperatures(Temperature);
{ Compute the mean noontime temperature. }
   Compute_Mean_Temperature(Temperature, Mean_Temperature);
{ Determine the days of the week having noontime temperatures }
{ above the mean temperature. }
   Days_Above_Mean(Temperature, Mean_Temperature,
            Number_Days_Above, Days_Above);
{ Determine the days of the week having noontime temperatures }
{ below the mean temperature. }
   Days_Below_Mean(Temperature, Mean_Temperature,
            Number_Days_Below, Days_Below);
{ Report the mean noontime temperature. }
   Report_Mean_Temperature( Mean_Temperature );
{ Report the days of the week having temperatures above the mean }
{ temperature. }
   Report_Days_Above(Days_Above, Number_Days_Above);
{ Report the days of the week having temperatures below the mean }
{ temperature. }
   Report_Days_Below(Days_Below, Number_Days_Below);
end.
```

This program includes a second user-defined type called `Table_of_Days`. This type is needed for the arrays `Days_Above` and `Days_Below` when values for these arrays are passed between the calling points in the main program and the procedures `Days_Above_Mean` and `Days_Below_Mean`, respectively. By using this approach, we can write the body of the main program as a sequential set of statements, with each statement representing a call to a procedure. No loops and no conditional statements are required within the body of the the main program. We have eliminated three global arrays, and we have declared two user-defined data types that are global to the procedures in which they are used. This demonstrates that, like variables and constants, user-defined types can be local to the modules in which they are declared as well as global to others.

9.3 MULTIDIMENSIONAL ARRAYS

Pascal supports arrays having more than one dimension. Figure 9.4 shows a two-dimensional array, called a *matrix*. As you can see, a matrix is different from a one-dimensional array in that it is composed of multiple rows and columns. In Pascal we can declare a matrix using the following syntax:

```
var
   Variable_Name : array [Low_Row.. High_Row,
                          Low_Col..High_Col] of data_type;
```

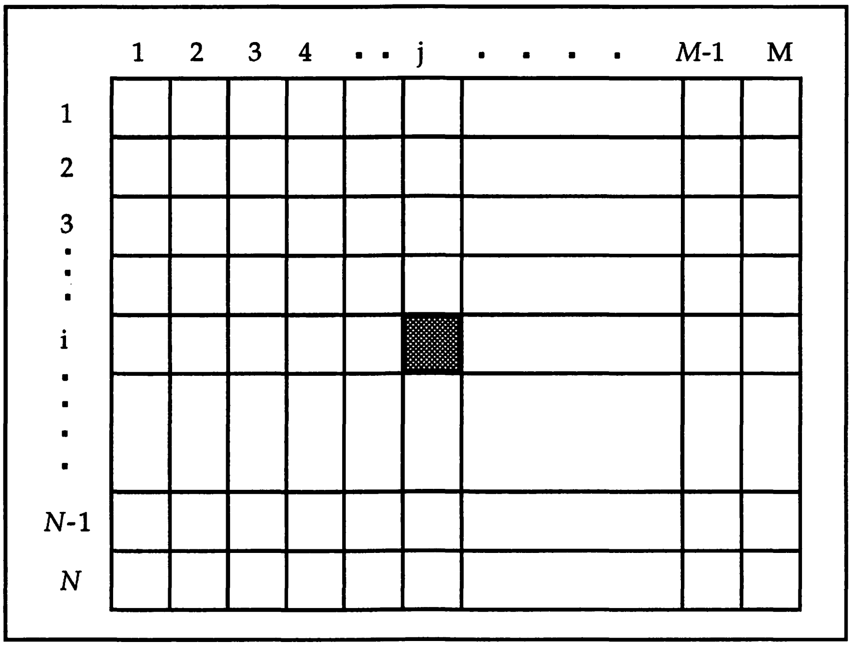


Figure 9.4 A two-dimensional array, called a *matrix*, composed of *N* rows and *M* columns.

A list of subrange types appears between the square brackets. The first pair represents the bounds for the row indices, and the second pair represents the bounds for the column indices. An element within a matrix is referenced through a subscripted variable having two indices, the first specify'ng the row position, and the second specifying the column position. In the matrix representation in Figure 9.4, the intersection of a row and column designates an element of the matrix. We reference an element of the array at the *i*th row and *j*th column by the subscripted variable `Variable_Name[i,j]` or `Variable_Name[i][j]`.

The number of elements in the matrix is given by the product

```
( High_Row - Low_Row + 1 ) * ( High_Col - Low_Col + 1 ).
```

As with a one-dimensional array, when we need to pass the information in a matrix using the actual parameter/formal parameter list, we must declare the formal parameter representing the matrix as a `user-defined` type, not directly as a two-dimensional array.

For example, consider the matrix shown in Figure 9.5, which is used for storing a person's full name, street address, city, state, and zip code. It is composed of N entries, N being in the range 2 through 100, with each row storing information for one person.

Initially, the user is asked to enter a value for N, representing the total number of names with corresponding addresses for entry from the keyboard. Following this step, N names with addresses are entered, sorted alphabetically by name, and then displayed to the Text window.

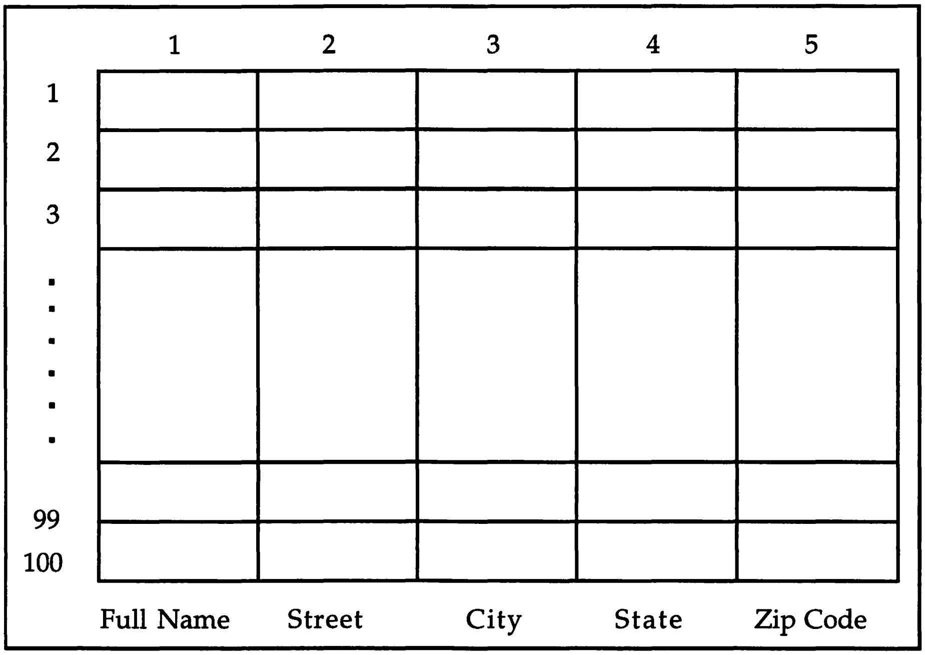


Figure 9.5 A matrix where each element is of type **string**[30].

Initial steps for an algorithm follow:

1. Prompt the user for the number of names, and then read this value.
2. Enter N names with addresses from the keyboard.
3. Sort the names alphabetically from A to Z.
4. Report the names and addresses to the screen.

Each step is refined in a separate module, with the main module being a sequential set of statements calling on the subordinate modules.

At the main level we need only two variables. The first, called N, represents the total number of names with addresses; the second, a matrix called Names, represents our table of addresses.

Let us increase our level of abstraction by defining the modules for each of the major steps. Our first module, called Enter_Number_of_Names, prompts the user for the value of N and then tests to see if this value is within the allowable range. If N is out of range, the user is informed that an error has been made and is again prompted for the same information. This protects the program against a "subscript-out-of-range" error during execution, in accord with the principle that critical input data should always be checked. The following steps define this module.

```
procedure Enter_Number_of_Names;
{ Purpose: This modules prompts the user for the number of names
   with addresses to be entered from the keyboard. It requires
   only one formal variable parameter, called N. }
begin
{ Prompt the user for number of names. }
   writeln('Type the number of names for entry from keyboard. ');
   write('This value must be greater than 1 and less than 101: ');
   readln( N );
   while ( N < 2 ) or ( N > 100 ) do
      { Check if this value is out of range. }
      begin
         writeln(' ***Sorry, this value is out of range.***');
         write('Type the number of names for entry');
         writeln(' from the keyboard. ');
         write('This value must be greater than 1 and less ');
         write( 'than 101: ');
         readln( N )
      end;
end; { Enter_Number_of_Names }
```

The second module prompts the user for N names with related addresses. Checking input data is not required at this stage. The following steps define this module.

```
procedure Enter_Names;
{ Purpose: This module accepts N names with addresses from the
   keyboard. It requires two formal parameters: a variable
   parameter called T, representing a matrix, and a value
   parameter N, representing the number of names to be read from
   the keyboard. This procedure uses a local string array called
   Prompt. }
begin
{ Initialize a temporary array Prompt with prompting messages. }
   Prompt[1] <-- 'Enter full name: ';
   Prompt[2] <-- 'Street address:  ';
   Prompt[3] <-- 'City: ';
   Prompt[4] <-- 'State: ';
   Prompt[5] <-- 'Zip code: ';
{ Enter names with addresses. }
   for  Row_Index <-- 1 to N do
      begin
         for Col_Index <-- 1 to 5 do
            begin
               write( Prompt[ Col_Index ] );
               readln( T[ Row_Index, Col_Index ] )
            end;
         writeln
      end;
```

```
end; { Enter_Names }
```

Notice that we are using a read statement to assign a value to an array element; the array element is referenced by index Row_Index for row and Column_Index for column. For each value of Row_Index the inner loop executes five times, during which it prompts the user and reads the full name, street address, city, state, and zip code.

The third module requires the sorting of names alphabetically from A through Z. For simplicity we will use the bubble sort algorithm shown in Figure 9.6.

```
procedure Bubble_Sort ( var A : Table; N : integer );
{ Purpose:  This algorithm sorts elements of table A from
            smallest to largest value. Table A is a
            variable type, since its elements may need to
            be exchanged. }
   var
      Pass, Row : integer;
begin
{ Perform N-1 passes through table A. }
   Pass <-- 1;
   repeat  { passing through table A until Pass > N-1 }
      Row <-- N;
      repeat
      { comparing elements of table A until Row < Pass + 1 }
         if A[ Row ] < A[ Row-1 ]
            then Swap( A[ Row ] , A[ Row-1 ] );
         Row <-- Row - 1;
      until Row < ( Pass + 1 );
      Pass <-- Pass + 1;
   until Pass > N-1
end; { Bubble_Sort }

procedure Swap( var X, Y : data_type );
{ This short algorithm interchanges the values of X and Y. }
   var
      Temporary : data_type;
begin
      Temporary <-- X ;  X <-- Y ;  Y <-- Temporary
end; { Swap }
```

Figure 9.6 The bubble sort algorithm.

This is called a *bubble sort* algorithm because the smallest values are bubbled to the top. We assume that the table called A is composed of N elements and that each element is of the same type.

1. The bubble sort algorithm begins by comparing A[N] with A[N-1]. If the value of A[N] is smaller than A[N-1], the two values are interchanged; otherwise, they are left unchanged. This process continues with the comparison of A[N-1] and A[N-2]. If the value of A[N-1] is smaller than the value of A[N-2], the two values are interchanged; otherwise, they are left unchanged.

This process continues until the last two values A[2] and A[1] are compared and, if the value of A[2] is less than A[1], they are interchanged. This completes the first pass through table A with N-1 comparisons being performed. What has occurred at the completion of this first pass? The smallest value has been "bubbled" to the top of the table, and placed in its proper sorted position. Unfortunately, the remaining elements in table A may still be unsorted, and the process must continue.

2. A second pass is performed by comparing the elements A[N] and A[N-1] and interchanging their values, if necessary, and continuing, until we have reached the second row of A by comparing A[3] and A[2]. At this point element A[2] will contain the second-smallest value of table A. No further comparisons are necessary in this pass through table A. Why not compare A[2] and A[1]? When performing this second pass, element A[1] already has the smallest value for the initial table A, and the value at A[1] has been placed in its proper sorted position. Comparing A[2] and A[1] would not change this fact. Notice that this second pass requires only N-2 comparisons.

3. As shown in the algorithm in Figure 9.6, we continue making passes through table A until the pass count exceeds N-1. Each time we perform a pass through table A, we begin at row N and perform comparisons on elements of A until the row index becomes less than the value of (Pass + 1). When necessary, the swap algorithm is called on for interchanging (swapping) two elements. How can the bubble sort algorithm be modified to sort in the reverse order, that is, from largest to smallest? The key is in the conditional statement that compares the values of adjacent elements.

Figure 9.7 shows an example of the actions of the bubble sort algorithm. The areas filled with black indicate where elements have been exchanged, and gray-shaded areas indicate where elements have been tested but not exchanged. The lightly shaded areas show where elements have been properly sorted.

The bubble sort and swap algorithms in Figure 9.6 require only minor changes for inclusion in our program for storing names and addresses. For the bubble sort algorithm the formal parameter A will be of type Matrix. In addition, the statement A[Row] < A[Row-1] is replaced with A[Row,1] < A[Row-1,1], because we sort the matrix A by comparing names from the first column elements of each row.

When any two names re uire swapping, all five column elements are interchanged, because the remaining four column elements of any particular row have information that is directly related to the name stored in the first column. The following module uses the bubble sort algorithm to sort the names into alphabetical order.

```
procedure Sort_Names ( var A : Matrix; N : integer );
{ Purpose: This algorithm sorts the elements of table A from
   smallest value to largest. This procedure requires two formal
   parameters: a variable parameter called A, representing a
   matrix, and a value parameter called N. Parameter A is a
   variable type, since its elements may need to be exchanged. N
   represents the total number of elements to be sorted. }
begin
{ Perform N-1 passes through table A. }
   Pass <-- 1;
```

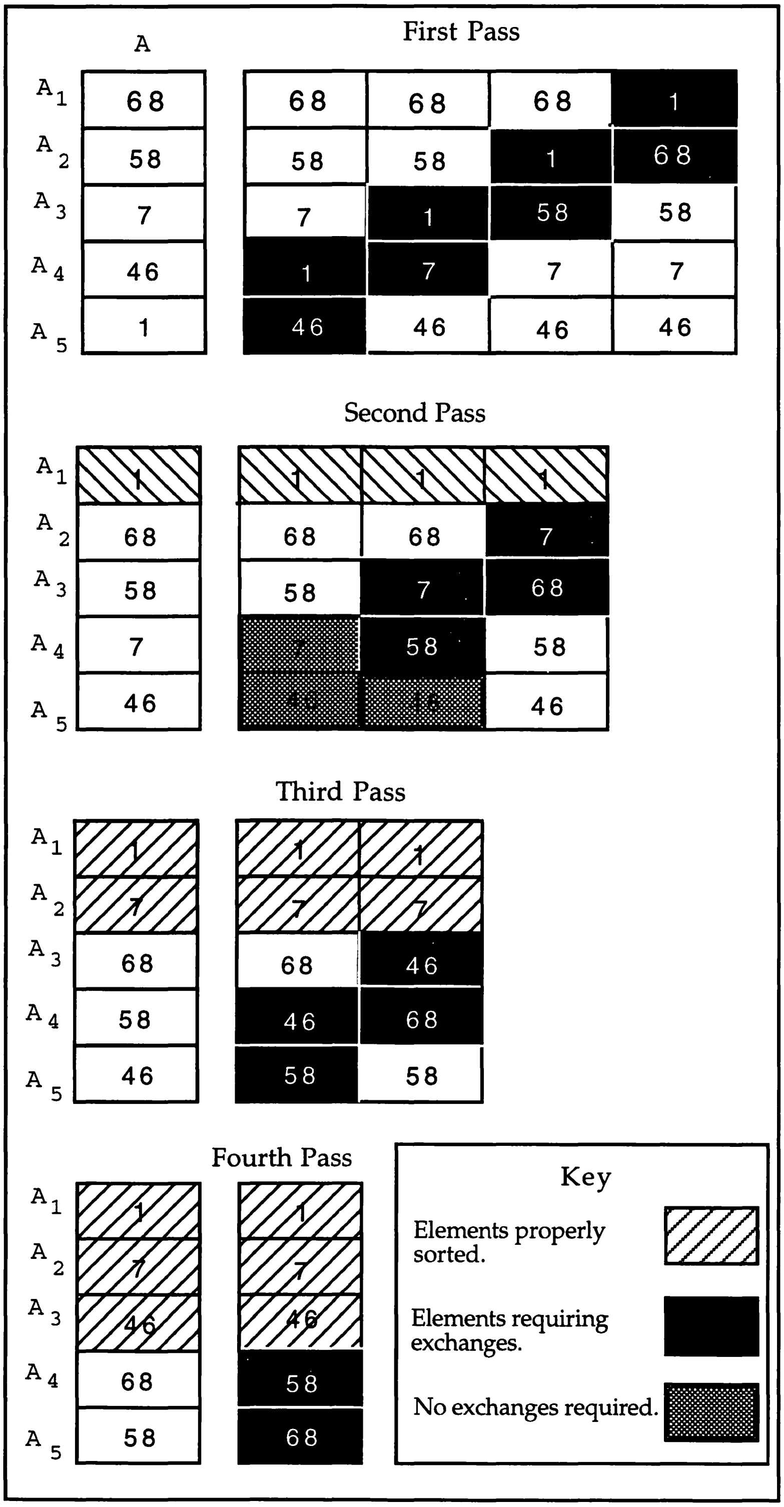


Figure 9.7 Using the bubble sort algorithm to sort an array having five elements.

```
   repeat  { passing through table A until Pass > N-1. }
      Row_Index <-- N;
      repeat{ comparing elements of table A until Row < Pass + 1 }
         if A[ Row_Index,1 ] < A[ Row_Index -1, 1 ]  then
            begin
               for Col_Index <-- 1 to 5 do
                  Swap( A[ Row_Index, Col_Index ],
                           A[ Row_Index -1, Col_Index]);
            end;
         Row_Index <-- Row_Index - 1;
      until Row_Index < ( Pass + 1 );
      Pass <-- Pass + 1;
   until Pass > N-1
end; { Sort_Names }
```

In the module `Swap` we must make only two minor changes by replacing each occurrence of the word `data_type` with `string`. Why? In `Bubble_Sort` the formal parameter `A` is a user-defined type called `Matrix`. In turn `Matrix` is equated with an **array**`[1..100, 1..5]` **of string**; that is, `A` represents a two-dimensional array composed of 100 rows and 5 columns, with each element of the matrix being a `string` type.

The last module is for reporting the alphabetically sorted list of names and the associated addresses. It requires two loops: an outer loop for each row position and an inner loop for displaying each column item. The steps for displaying this information are as follows:

```
procedure Report_Names;
{ Purpose: This module reports the names and related information
   for each person stored in matrix A. It requires two formal
   value parameters: A, representing a matrix, and N, representing
   the number of rows in matrix A. This procedure requires a local
   string array called Message. }
begin
{ Initialize message array. }
   Message[1] <-- 'Full name:  ';
   Message[2] <-- 'Street address: ';
   Message[3] <-- 'City: ';
   Message[4] <-- 'State: ';
   Message[5] <-- 'Zip code: ';
{ Display N names with addresses from table A. }
   for Row_Index <-- 1 to N do
      begin
         for Col_Index <-- 1 to 5 do
            writeln( Message[ Col_Index ] , A[ Row_Index,
                     Col_Index ] );
         writeln;
      end;
end; { Report_Names }
```

We can declare an *n*-dimensional array by using either a user-defined type or through **var** declaration. The following shows these steps:

```
type
  Large_Array = array[Low_1.. High_1, Low_2.. High_2,... ,
                          Low_N.. High_N] of type;
var
    Variable_Name : Large_Array;
```

or

```
var
  Variable_Name : array[Low_1.. High_1, Low_2.. High_2, ... ,
                            Low_N.. High_N] of type;
```

Notice that there is a list of N subrange types for the array declaration with the index range for the first dimension on the left, followed by the second dimension, and so forth, and the highest index range on the right. The identifiers Low_1, High_1, Low_2, High_2,..., Low_N, High_N *must* have actual values when the array is translated by the Pascal system. These can be either explicit unsigned integer constants or values defined through **const** statements.

A subscripted variable for referencing an element in an *n*-dimensional array has the format

```
Variable_Name[ j1, j2, j3, . . .,jN-1, jN ]
```

or

```
Variable_Name[j1] [j2] [j3] . . . [jN-1] [jN]
```

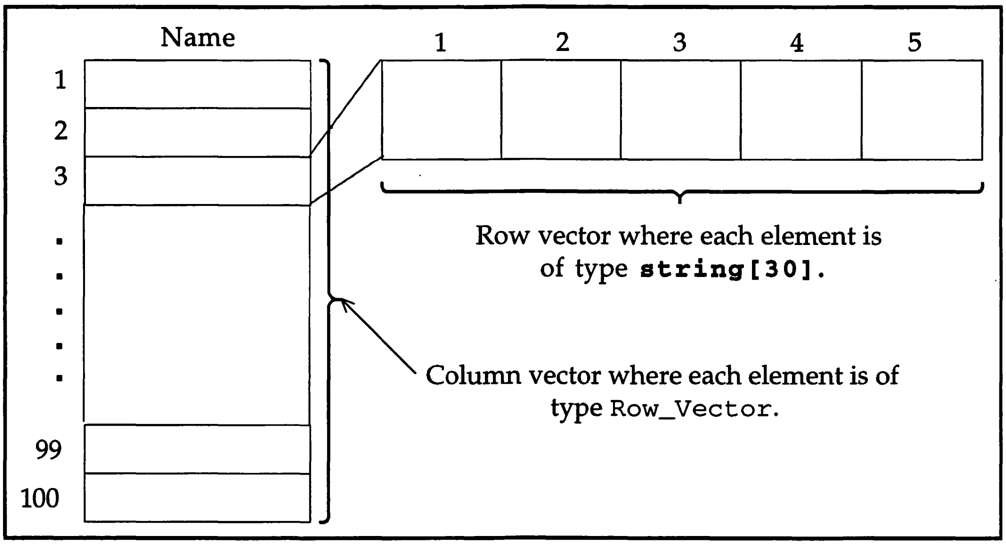
with each subscript representing the index position of each dimension.

9.4 AN ARRAY OF ARRAYS

By declaring a user-defined type as an array type, it becomes possible to have a variable declared as an array where each element of the array is itself and array. For example, consider the following statements for representing the matrix discussed in Section 9.3.

```
type
   Row_Vector  = array[1..5] of string;
   Matrix = array[1..100] of Row_Vector;
var
   Name : Matrix;
```

Figure 9.8 illustrates this concept by using a column and row vector. This shows that the variable called Name is declared as a user-defined type called Matrix, where Matrix is defined as a 100-element column array, with each element being of type Row_Vector. This demonstrates how a variable declared as an array may have elements that are themselves declared as array types. In turn, the type Row_Vector represents a one-dimensional array composed of five elements, each element being of type **string**[30].



Name
1
2
3
99
100
1
2
3
4
5
Row vector where each element is of type **string[30]**.
Column vector where each element is of type Row_Vector.

Figure 9.8 An example of an array of arrays.

How can we reference elements within the array, such as Name? The subscripted variable Name[i] references the *i*th element (*i*th row) of the array Name; its value is a five-element array of strings. The subscripted variable Name[i, j] or Name[i][j] references the *i*th row of the array Name and, within the *i*th row, the *j*th element of a five-element array.

For example, consider the following statement taken from the module Sort_Names:

```
if A[ Row_Index,1 ] < A[ Row_Index -1, 1 ] then
   for Col_Index <-- 1 to 5 do
      Swap(A[Row_Index, Col_Index], A[Row_Index -1, Col_Index] );
```

Rather than interchanging each column element of A one at a time, we can now interchange two complete row vectors. Simply replace the code with the following statement:

```
if A[ Row_Index,1 ] < A[ Row_Index -1, 1 ] then
   Swap( A[ Row_Index ] , A[ Row_Index -1] );
```

The module Swap now interchanges a complete row vector in one step, making it unnecessary to call on this module from the body of a **for** loop. For the module Swap two minor changes are required: the data types associated with the identifier Temporary and the formal parameters X and Y are changed from type string to the user-defined type called Row_Vector. The following shows these changes.

```
procedure Swap( var X, Y : Row_Vector );
{ Interchanges the values of the two row vectors X and Y. }
   var
      Temporary : Row_Vector;
begin
   Temporary := X;  X := Y;  Y := Temporary
end;  { Swap }
```

The complete Pascal program for the sorting problem discussed in Section 9.3, titled `Sorting_Table_of_Names`, follows. It declares the matrix Name to be an array of row vectors. This is a Macintosh Pascal program, but it will execute under THINK Pascal with minor modifications. First, the **uses** clause must be removed. Second, we must increase the stack size under the **Run Options...** command option from the THINK Pascal **Run** menu. Try increasing the stack size from 16K to 32K.

```
program Sorting_Table_of_Names(input, output);
{ Purpose:  This program enters names and addresses and sorts }
{           the names alphabetically from A to Z. }
   uses
      QuickDraw1;
   type
      Row_Vector = array[1..5] of string[30];
      Matrix = array[1..100] of Row_Vector;
   var
      Name : Matrix;
      Number_of_Names : integer;
{ ********************************************************** }
   procedure Set_Text_Window;
      var
         Window : Rect;
   begin
      HideAll;
      SetRect(Window, 0, 20, 512, 300);
      SetTextRect(Window);
      ShowText
      end;
{ ********************************************************** }
   procedure Enter_Number_of_Names (var N : integer);
   { Purpose:  This module prompts the user for the number }
   {           of names with addresses to be entered from the }
   {           keyboard. }
   begin
   { Prompt user for number of names. }
      write ('Type the number of names for entry from the ');
      writeln ('keyboard. ');
      write('This value must be greater than 0 and less ');
      write('than 101: ');
      readln(N);
      while (N < 2) or (N > 100) do
      { Check if this value is out of range. }
         begin
```

```
          writeln;
          writeln(' ***Sorry, this value is out of range.***');
          write('Type the number of names for entry');
          writeln(' from the keyboard. ');
          write('This value must be greater than 1 and less ');
          write('than 101: ');
          readln(N);
        end;
      writeln
   end;
{ ******************************************************** }
   procedure Enter_Names (var A : Matrix; N : integer);
   { Purpose:  This module enters N names with addresses from }
   {           the keyboard. }
      var
        Row_Index, Col_Index : integer;
        Prompt : array[1..5] of string;
   begin
   { Initialize temporary array Prompt with prompting messages. }
      Prompt[1] := 'Enter full name: ';
      Prompt[2] := 'Street address:  ';
      Prompt[3] := 'City: ';
      Prompt[4] := 'State: ';
      Prompt[5] := 'Zip code: ';
   { Enter names with addresses. }
      for Row_Index := 1 to N do
        begin
          for Col_Index := 1 to 5 do
            begin
              write(Prompt[Col_Index]);
              readln( A[Row_Index, Col_Index])
            end;
          writeln
        end;
   end;
{ ******************************************************** }
   procedure Swap (var X, Y : Row_Vector);
      var
        Temporary : Row_Vector;
   begin
      Temporary := X;
      X := Y;
      Y := Temporary
   end;
{ ******************************************************** }
   procedure Sort_Names (var A : Matrix;  N : integer);
   { Purpose:  This algorithm sorts elements of table A from }
   {           smallest value to largest. Table A is a variable }
   {           type, since its elements may require exchange. }
      var
        Pass, Row_Index, Col_Index : integer;
```

```
  begin
  { Perform N-1 passes through table A. }
     Pass := 1;
     repeat  { passing through table A until Pass > N-1 }
        Row_Index := N;
        repeat
        { Compare elements of table A until Row < Pass + 1. }
           if A[Row_Index, 1] < A[Row_Index - 1, 1] then
              for Col_Index := 1 to 5 do
                 Swap( A[Row_Index] , A[Row_Index - 1]);
              Row_Index := Row_Index - 1;
        until Row_Index < (Pass + 1);
            Pass := Pass + 1;
     until Pass > N - 1
  end;
{ ********************************************************** }
  procedure Display_Title;
     var
       J : integer;
  begin
     for J := 1 to 60 do
        write('_');
     writeln;
     writeln('   LIST OF PERSONS: ');
     writeln
  end;
{ ********************************************************** }
  procedure Report_Names ( A : Matrix;  N : integer);
  { Purpose:  This module reports the names and related }
  {           information for each person stored in matrix A. }
     var
        Row_Index, Col_Index : integer;
        Message : array[1..5] of string;
  begin
  { Initialize message array. }
     Message[1] := 'Full name:  ';
     Message[2] := 'Street address: ';
     Message[3] := 'City: ';
     Message[4] := 'State: ';
     Message[5] := 'Zip code: ';
  { Display title before displaying names.}
     Display_Title;
  { Display N names with addresses from table A. }
     for Row_Index := 1 to N do
        begin
           for Col_Index := 1 to 5 do
              writeln(Message[Col_Index] , A[Row_Index,
                     Col_Index]);
              writeln;
           end;
  end;
```

```
{ ******************************************************** }
begin
{ Set the Text window and show it for viewing. }
   Set_Text_Window;
{ Prompt the user for the number of names and read this value. }
   Enter_Number_of_Names(Number_of_Names);
{ Enter N names with addresses from the terminal. }
   Enter_Names(Name, Number_of_Names);
{ Sort the names alphabetically from A to Z. }
   Sort_Names(Name, Number_of_Names);
{ Report the names with addresses to the screen. }
   Report_Names(Name, Number_of_Names)
end.
```

Notice that the data type for `Row_Vector` is an array of type `string` with each element allocated a maximum length of 30 characters. Allowing the default of 255 characters causes the declaration for `Matrix` to be pointed to by the hand symbol, indicating that insufficient memory exists for creating an object of this size. This occurs because, while `string` types are dynamically stored internally in memory, the estimated storage for an array having `string` types at translation time is based on the maximum string length of each array element.

9.5 APPLICATION OF ARRAYS: SORTING AND SEARCH ALGORITHMS

9.5.1 Sorting Algorithms

Numerous algorithms exist for sorting data stored in arrays. We will consider three sort algorithms: straight insertion sort, Shellsort, and quicksort. Figure 9.9 shows the steps for the straight insertion sort algorithm.

Like the bubble sort, the straight insertion sort algorithm sorts N elements stored in a table. During the process of sorting array elements, A[0] is used for storing values temporarily. The algorithm begins by picking the second element of A, assigning it to X and A[0]. It then sifts through the array elements, starting with the *i*th −1 position and continuing until an element in A is found having a value greater than or equal to the value of X. When X is less than the *j*th value of A, the *j*th element of A is moved to the *j*th + 1 position of A, with J then being decremented. After leaving the inner loop, the value of X is inserted in its proper sorted position among the elements of A that have presently been sorted. These steps are repeated while I remains less than or equal to N.

```
procedure Straight_Insertion_Sort (var A: Table; N: integer );
{ Purpose:  Table A represents an array[0..N] of item, where }
{           A[0] is used to store the temporary value. }
   var
      I, J : integer;
      X : item;
begin
   I := 2;
   while ( I <= N ) do
      begin
```

```
         X := A[I];
         A[0] := X;
         J := I - 1;
         while (  X < A[J]  ) do
            begin
               A[J+1] := A[J];
               J := J - 1
            end { inside while-do loop}
         A[J+1] := X;
         I := I + 1;
      end; { outside while-do loop}
end;  { Straight_Insertion_Sort }
```

Figure 9.9 The straight insertion sort algorithm for sorting elements from smallest to largest.

The following table shows a trace of the algorithm for an array A having four elements:

A[J]	0	1	2	3	4	
original A -->		12	9	11	8	
I =2	9	9	12	11	8	
I =3	11	9	11	12	8	
I =4	8	8	9	11	12	<-- sorted A

This algorithm uses a sifting-down technique by comparing the value of X with A[J] and either inserting X to the right of A[J] if X is greater than or equal to A[J] or proceeding to the left in search of a larger value. As the trace table shows, our array A becomes sorted with the smaller values being positioned toward the top (left) and the larger values toward the bottom (right).

Unfortunately, both the bubble sort and the straight insertion sort algorithms are not fast sorting techniques for large values of N. For these algorithms the maximum number of comparisons, like the maximum number of assignments, is directly proportional to N^2.

In 1959 D. L. Shell proposed a refinement to the straight insertion sort. Figure 9.10 shows the steps for this sorting algorithm, Shellsort.

```
procedure Shellsort ( var A : Table ;  N : integer );
   var
      H, I, J, Temp : integer;
      Flag : Boolean;
begin
{ Compute the smallest value of H for which H > N. }
   H := 1
   repeat
      H := 2 * H + 1;
   until ( H > N );
```

```
{ Sort elements in table A spaced a distance H apart, using the }
{ concept of the straight insertion sort. }
   repeat
      H := H div 2;
      I :=  H + 1;
      while  ( I <= N ) do
         begin
            A[0] := A[I];
            J := I - H;
            Flag := true;
            while ( A[0]  <  A[J] ) and Flag do
               begin
                  A[ J + H ]  :=  A[J];
                  J :=  J - H;
                  if  J < H  then { Exit this inside loop. }
                     begin
                        Temp  :=  J;
                        J := J + H;
                        Flag  :=  false
                     end; { if-then }
               end; { inside while-do loop }
            J := Temp;
            A[J + H] :=  A[0];
            I  :=  I + 1
         end;  { outside while-do loop }
   until  ( H = 1 );
end; { Shellsort }
```

Figure 9.10 The Shellsort algorithm.

Shell suggested that elements located a distance H apart be sorted by first repeating the steps of the straight insertion sort, as the value of H is decreased. In the last step H will finally be 1, with all the elements now being sorted as a straight insertion sort over N elements. This seems strange because we are using the steps of the straight insertion sort several times over. Why should this algorithm be an improvement? First, the algorithm sorts fewer elements of the array when H is greater than 1, and when H does become 1 most of the elements have already been sorted. It has been observed that when H is 1 and N is large, the total number of comparisons is directly proportional to N, with the total number of assignments being directly proportional to 2N.

Second, we choose values for H from the sequence of numbers 1, 3, 7, 15, 31, ... When the value of N is large, the number of comparisons is directly proportional to $N^{1.5}$. This means that when an array has a large number of elements (for example, N = 1000), and the initial elements of the array are randomly assigned, the Shellsort will execute faster than either the bubble sort or the straight insertion sort.

A third algorithm using the principle of exchanging elements is the quicksort algorithm, which has two basic steps. First, the algorithm requires the array to be subdivided into left and right partitions. These partitions are then sorted recursively by repeating the same two steps. Figure 9.11 shows the procedure for the quicksort algorithm.

```
procedure Quicksort ( var A : Table;  N : integer )
begin
   Sort( A, 1, N )
end;  { Quicksort }
{ ************************************************************ }
procedure Sort( var A : Table;  L, R : integer );
   var
      I, J : integer;
      X, Temp : item;
begin
{ Initialize local indices and select a pivot point represented }
{ by X. }
   I := L;
   J := R;
   X := A[ ( L + R ) div 2 ];
{ Partition array A into two subarrays. }
   repeat
      while ( A[I] < X ) do
         I := I + 1;
      while ( A[J] > X ) do
         J := J - 1;
   { Swap the values A[J] and A[I]. }
      if I <= J then
         begin
            Temp := A[I] ;  A[I]  :=  A[J] ;  A[J]  :=  Temp;
            I :=  I + 1;  J := J - 1
         end;  { if-then }
   until I > J;
{ Sort the left partition of the array or subarray. }
   if L < J  then  Sort( A, L, J );
{ Sort the right partition of the array or subarray. }
   if R > I  then  Sort( A, I, R );
end;  { Sort }
```

Figure 9.11 The quicksort algorithm.

Notice that it contains the steps necessary for partitioning array A into two smaller arrays. This is accomplished by first selecting an element from A represented by X, scanning the array A from the left until an element A[I] is found where A[I] > X, then scanning the array from the right until an element A[J] is found where A[J] < X. At this point we have found an element of A larger than X and a second element smaller than X. If the left index I is less than or equal to the right index J, the values A[J] and A[I] are exchanged (swapped). The left index is then incremented and the right index decremented, with the steps for scanning and exchanging being repeated. Once the value of the left index exceeds the value of the right index, array A is divided into two subarrays: a left partition containing elements of A less than X, and a right partition containing elements of A greater than X. When the left and right indices are equal, the value of X has been sorted to its proper position, with the subarrays to the right and left needing to be sorted.

Quicksort is now executed recursively by first sorting the left partition for as long as the right index J remains greater than the low index bound of the left partition. This is followed by sorting the right partition for as long as the left index remains less than the high bound of the right partition.

The quicksort algorithm saves time when sorting a table of random elements. When N is large, both the number of comparisons and the number of exchanges are directly proportional to $N \log_2 N$. This is much faster than the factor N^2 required by the bubble sort and straight insertion sort algorithms. Unfortunately, if most of the elements of table A are sorted, execution time for the quicksort algorithm is directly proportional to N^2.

Trace by hand each of the sorting algorithms, using tables of unsorted numbers having from 5 to 10 elements. By tracing each of the sort algorithms and establishing a trace table, you will better understand each step in the sorting algorithm. In addition, it reinforces your skill in reading and understanding the steps of an algorithm.

9.5.2 Search Algorithms

In addition to sorting data, an algorithm may require a search for one or more elements stored within a table, using an object called a *key*. For example, let us assume that a telephone directory exists in RAM when our program is executing. Given a person's full name as the key, we need a fast search procedure for finding the name and reporting the associated phone number. One method is to use a simple procedure known as the *linear search algorithm* shown in Figure 9.12.

```
procedure  Linear_Search( A : Table;  Key : item;  N : integer;
                          var Position : integer);
{ Element A[0] will store the value of Key during execution. }
  var
    I : integer;
begin
  I := N;
  A[0] := Key;
  while ( A[I] <> Key ) do
    I := I - 1;
  Position := I
end; { Linear_Search }
```

Figure 9.12 The linear search algorithm.

First, this algorithm does not require that table A be sorted. It begins execution by assigning the value of the key to A[0]; it then points to one end of the table and continues to search for the location having the key. The search is performed by comparing the value of Key with each successive element of table A. For the algorithm shown in Figure 9.12, the value returned is either the position where the key is found or zero if the key is not found.

Although this algorithm is simple, its average search time (the average time to find a key) is directly proportional to the factor N/2. This means that if we perform numerous searches using this algorithm, on the average we will need to search almost half the table before finding the key. The average search time seems smaller when compared to the execution time needed in our other sorting algorithms, but execution time for searching becomes costly as N becomes larger. Using this algorithm to search a table with 20 names is simple, but using it to search the New York City phone directory would be incredibly slow.

A second algorithm for searching that is more efficient with respect to execution time is the binary search algorithm shown in Figure 9.13.

```
procedure Binary_Search( A : Table; Key : integer; N : integer;
                               var Position : integer );
   var
      Low_Bound, High_Bound, Middle : integer;
      Found : Boolean;
begin
{ Initialize the low- and high-bound indices. }
   Low_Bound := 1;
   High_Bound := N;
   repeat
   { Compute the center position of table A or subtable. }
      Middle  :=  ( Low_Bound + High_Bound ) div 2;
   { Check if the item is at A[Middle]. }
      if A[Middle] = Key  then
         begin
            Position := Middle;
            Found := true
         end
      else
      { Continue searching either lower or upper table or }
      { subtable. }
         if A[Middle] < Key  then
            High_Bound := Middle - 1
         else
            Low_Bound := Middle + 1;
   until Found or ( Low_Bound > High_Bound );
end;  { Binary_Search }
```

Figure 9.13 The binary search algorithm.

Table A must be ordered when using this algorithm. The basis for the algorithm is similar to searching the telephone directory for a phone listing given the person's name as a key. For example, if we were searching for a phone number, we might turn to the middle of the phone book to see if the person's name is at the top of the left page. If not, knowing that the book is arranged in alphabetical order, we would ask ourselves if the name should be listed in the first half or the second half of the directory. Selecting the appropriate half, we continue the search by dividing that portion of the telephone directory in half. We again check to see if the person's name is at the top of the left page and if not, ask if the key comes before or after this page. We then divide the selected portion of the telephone directory into halves, continuing these steps until we eventually locate the person or find that the person is not listed.

The binary search algorithm has three basic steps. First, initialize the indices Low_Bound and High_Bound and determine the middle of the table. Then compare the key with A[Middle] and, if the key is not found, check to see if it is ordered less than A[Middle]. If it is, modify High_Bound, because the key may exist between Low_Bound and the position Middle - 1. Otherwise, modify Low_Bound, because the key may exist between Middle + 1 and High_Bound. These steps are repeated by computing a new middle index and again checking the key with A[Middle], until either Low_Bound exceeds the value of High_Bound, or the key has been found.

The binary search algorithm has a major advantage over the linear search algorithm, because it subdivides a table into two parts, searching only the subtable that may contain the key and excluding a second subtable that is no longer relevant. With this approach, the average search time is directly proportional to $\log_2(N + 1)/2$. For example, if table A has 1024 (2^{10}) elements, the average search time for the linear search algorithm is directly proportional to 512, whereas for the binary search algorithm this factor is only 5. For numerous searches, therefore, we would need to examine almost half the elements in table A using the linear search approach before finding the key, compared with only five elements using the binary search algorithm. This implies a substantial saving in execution time for a table having a large number of elements.

9.6 AN INHOMOGENEOUS STRUCTURE: THE PASCAL RECORD

In an array structure, all elements are of the same type; the Pascal record allows the creation of a structure having several different fields, with each field representing a different data type. Here is a format for declaring an object as a record type:

```
record
   field1 : data type;
   field2 : data type;
          . . .
   fieldn : data type
end;
```

Note that the declaration begins with the reserved word **record**, followed by a field list composed of a label representing an identifier name, followed by a colon, followed by a data type, followed by the reserved word **end**. The fields can be viewed as having actions similar to the subscript of an array.

For example, consider a record that will be used for storing a date composed of the month as a string, day of the week as both a string and an unsigned number, time of the day as a long integer, and year as an integer. Here is the structure for our record type, called Date, along with an identifier of type Date:

```
type
   Date =    record
                Month: string[8];
                Dayname: string[9];
                Day: 1..31;
                Year:  integer;
                Time:  longint
             end;
var
   Access_Date : Date
```

The identifier Access_Date is declared to be a record structure of type Date, which has five different fields with names Month, Dayname, Day, Year, and Time. The concept of a record as an inhomogeneous structure derives from the fact that each field of the structure is declared as a different data type. How does one reference a field of the variable Access_Date? This is done using the syntax Variable_Name.field (variable name, period, field name). For example, we may want to display the present date using only three fields and the statement

```
writeln( Access_Date.Month,  Access_Date.Day, ',' ,
            Access_Date.Year:4 );
```

We can extend this example by using our variable Access_Date for first storing the present date and time, then displaying all the fields of Access_Date to the Text window. To extract the date and time, we will use the Macintosh Pascal library procedure **GetTime.** This procedure gets the current date and time, generated by the Macintosh system clock, and returns it through a formal variable having a Pascal system type called **DateTimeRec**, which has the following record structure:

```
DateTimeRec =  record
                  Year,  Month,  Day,  Hour,  Minute,  Second,
                        DayOfWeek : integer
               end;
```

This record does not need to be explicitly declared, because it is known to the Pascal system when a program is translated. We now consider the major steps for displaying the time of day.

1. Close all windows, and then establish the Text window for viewing.
2. Fetch the date and time using the procedure **GetTime** by assigning this information to a temporary variable called Temp_Date.
3. Convert the date and time from the format in **DateTimeRec** to the format of Date.
4. Display the date and time to the Text window.

The main module requires only two variables: Access_Date of record type Date, and Temp_Date of record type **DateTimeRec**. The first two steps are straightforward; let us expand Step 3 by developing a module called Convert_Date. This module will take values from Temp_Date and transform them to a form acceptable for Access_Date.

```
procedure Convert_Date ( var Out_Date : Date; In_Date :
                              DateTimeRec );
{ Purpose: This module takes the temporary date given by In_Date
   and converts values in its fields to a form understood by
   Out_Date. This procedure requires two formal parameters:
   Out_Date, a variable parameter of type Date, and In_Date, a
   value parameter of type DateTimeRec. Two functions are called
   from this procedure: Present_Month for returning a string value
   representing one of the 12 months of the year and
   Present_Dayname for returning a string value representing one
   of the seven days of the week. }
```

```
begin
{ Convert numeric month to a string representation. }
   Out_Date.Month <-- Present_Month( In_Date.Month );
{ Convert numeric day of the week to a string representation. }
   Out_Date.Dayname <-- Present_Dayname( In_Date.DayOfWeek );
{ Copy numeric day to Out_Date. }
   Out_Date.Day <-- In_Date.Day;
{ Copy numeric year to Out_Date. }
   Out_Date.Year <-- In_Date.Year;
{ Copy time of day to Out_Date. }
   Out_Date.Time := In_Date.Hour * 10000 +
                        In_Date.Minute * 100 + In_Date.Second;
end;  { Convert_Date }
```

Although the time in hours, minutes, and seconds is stored in separate fields for the record **DateTimeRec**, these three values are combined into a `long integer` and stored in the field `Time` of the record `Date`. Keep in mind that the Macintosh system clock is a 24-hour clock. The identifier `Hour` will have a range of 0 through 23 (0 for midnight, 23 for the twenty-third hour of the day). When displaying time, we must extract the hours, minutes, and seconds and determine whether or not we have passed noon. The program titled `Computing_the_Date` provides a detailed listing for all procedures and functions. This Macintosh Pascal program will execute under THINK Pascal if the **uses** clause is removed.

```
program Computing_the_Date(input, output);
{ Purpose:  This program shows a simple application of a record. }

   uses
      QuickDraw1;
   type { Declare user-defined type called Date. }
      Date =   record
                  Month : string[8];
                  Dayname : string[9];
                  Day : 1..31;
                  Year : integer;
                  Time : longint
               end;
   var
      Access_Date : Date;
      Temp_Date : DateTimeRec;
      { DateTimeRec is a record type known to Macintosh Pascal. }
{ ********************************************************** }
   procedure Set_Text_Window;
      var
         Window : Rect;
   begin
      HideAll;
      SetRect(Window, 150, 100, 350, 160);
      SetTextRect(Window);
      ShowText
   end;
```

```
{ ********************************************************** }
  function Present_Month (Number : integer) : string;
  begin
  { Purpose:  Assign string equivalents for numeric months. }
     case Number of
        1 :    Present_Month := 'January';
        2 :    Present_Month := 'February';
        3 :    Present_Month := 'March';
        4 :    Present_Month := 'April';
        5 :    Present_Month := 'May';
        6 :    Present_Month := 'June';
        7 :    Present_Month := 'July';
        8 :    Present_Month := 'August';
        9 :    Present_Month := 'September';
        10 :   Present_Month := 'October';
        11 :   Present_Month := 'November';
        12 :   Present_Month := 'December';
     end;
  end;
{ ********************************************************** }
  function Present_Dayname (Number : integer) : string;
  begin
  { Purpose:  Look up a string day of the week matching the }
  {           numeric day of the week. }
         case Number of
           1 :   Present_Dayname := 'Sunday';
           2 :   Present_Dayname := 'Monday ';
           3 :   Present_Dayname := 'Tuesday';
           4 :   Present_Dayname := 'Wednesday';
           5 :   Present_Dayname := 'Thursday';
           6 :   Present_Dayname := 'Friday';
           7 :   Present_Dayname := 'Saturday';
        end;
     end;
{ ********************************************************** }
  procedure Convert_Date (var Out_Date : Date; In_Date :
                            DateTimeRec);
  { Purpose:  This module takes the temporary date given by }
  {           In_Date and converts values in its fields to a }
  {           form understood by Out_Date. }
  begin
  { Convert numeric month to a string representation. }
     Out_Date.Month := Present_Month(In_Date.Month);
  { Convert numeric day of the week to a string representation. }
     Out_Date.Dayname := Present_Dayname(In_Date.DayOfWeek);
  { Copy numeric day to Out_Date. }
     Out_Date.Day := In_Date.Day;
  { Copy year to Out_Date. }
     Out_Date.Year := In_Date.Year;
  { Copy time of day to Out_Date. }
     Out_Date.Time := In_Date.Hour * 10000 +
                              In_Date.Minute * 100 + In_Date.Second;
   end;
{ ******************************************************** }
   procedure Report_Date_Time (In_Date : Date);
      var
         Hour, Minute, Second : integer;
   begin
   { Display the date. }
      writeln;
      write(In_Date.Dayname, ', ', In_Date.Month);
      writeln( In_Date.Day : 3, ',', In_Date.Year : 5);
      Hour := In_Date.Time div 10000;
      Minute := (In_Date.Time - Hour * 10000) div 100;
      Second := In_Date.Time - (Hour * 10000 + Minute * 100);
   { Display the time in either A.M. or P.M. }
      writeln;
      if Hour < 12 then
         writeln(Hour: 2, ':', Minute: 2, ':', Second: 2, ' A.M.')
      else
         begin
            if Hour > 12 then
               Hour := Hour mod 12;
            write(Hour: 2, ':', Minute: 2, ':', Second: 2, 'P.M.')
         end
   end;
{ ******************************************************** }
begin   { Body of the main program. }
{ Establish Text window for viewing screen. }
   Set_Text_Window;
{ Access system date and time. }
   GetTime(Temp_Date);
{ Convert fields in Temp_Date for Access_Date. }
   Convert_Date(Access_Date, Temp_Date);
{ Display present date and time to Text window. }
   Report_Date_Time(Access_Date)
end.
```

Notice that we use two functions for simulating a table lookup: one for returning a string representation of the month and one for returning a string representation of the day of the week.

In the procedure `Report_Date_Time`, we use three local variables for storing hours, minutes, and seconds. Although the Macintosh has a 24-hour clock, we are reporting time using a 12-hour clock. If the value of `Hour` is less than 12, it is morning, and the string `A.M.` is attached at the end of the displayed time. From the twelfth to the thirteenth hour, the value of `Hour` must be 12. Beyond the twelfth hour we modify the value of `Hour` by executing the statement `Hour := Hour` **`mod`** `12`. Notice that the `write` statements have minimum-width fields for enhancing the appearance of both the date and time.

Pascal supports a statement that can reduce the need to explicitly reference the name of a record variable. It is referred to as the **`with`** statement and has the following format:

```
with  record variable list do statement;
```

For example, the following can replace all of the statements in the body of the procedure Report_Date_Time of the program Computing_the_Date:

```
with In_Date do
   begin  { Display the date. }
      writeln;
      writeln(Dayname, ', ', Month, Day : 3, ',', Year : 5);
      Hour := Time div 10000;
      Minute := (Time - Hour * 10000) div 100;
      Second := Time - (Hour * 10000 + Minute * 100);
      { Display the time in either A.M. or P.M. }
      writeln;
      if Hour < 12 then
         write(Hour: 2, ':', Minute: 2, ':', Second: 2, ' A.M.')
            else
               begin
                  if Hour > 12 then
                     Hour := Hour mod 12;
                  write(Hour: 2, ':', Minute: 2, ':', Second: 2,
                           'P.M.')
               end
   end;
```

As you can see, we apply the **with** statement to reduce the need to reference the record variable In_Date. The body of the procedure Convert_Date can also be modified to reduce references to both In_Date and Out_Date. The following, for example, can replace the body of procedure Convert_Date:

```
with Out_Date, In_Date do
   begin
   { Convert numeric month to a string representation. }
      Out_Date.Month := Present_Month( In_Date.Month );
   { Convert numeric day  of the week into a string }
   { representation. }
      Dayname := Present_Dayname( DayOfWeek );
   { Copy numeric day to Out_Date. }
      Out_Date.Day := In_Date.Day;
   { Copy year to Out_Date. }
      Out_Date.Year := In_Date.Year;
   { Copy time of day to Out_Date. }
      Time := Hour * 10000 + Minute * 100 + Second;
   end;
```

It is not always possible to remove the referencing of a record variable from the body of the **with** statement when two or more record variables have similar field identifiers. For example, replacing the statement

```
Out_Date.Month := Present_Month( In_Date.Month );
```

with

```
Month := Present_Month( Month );
```

will produce neither a syntax error nor a semantic error for Macintosh Pascal. However, though executable, it has no semantic meaning, because it is impossible to relate the field identifier `Month` with either record variable `Out_Date` or `In_Date`. In some instances there are expressions and statements in the body of the **with** statement so that explicit referencing of record variables is unnecessary.

Pascal supports an extension to declaring a record. The following format shows how a record type can be composed of a fixed part followed by a variant part.

```
record
   fixed part;
   variant part;
end;
```

Up to this point, we have seen only examples of record declarations composed of fixed parts. A record declaration having a variant part allows the selection of one or more fields based on the value of a variable field. For example, consider the user-defined record type:

```
type
   Geometric_Shape = (Rectangle, Circle, Triangle,
                          Point_to_Point);
   Geometric_Object =
                          record
                             case Figure : Geometric_Shape of
                                Rectangle    : ( Boundary : Rect);
                                Circle       : ( Center : Point;
                                                 Radius : integer );
                                Triangle     : ( Corner_1, Corner_2,
                                                 Corner_3: Point);
                                Point_to_Point : ( Point_1, Point_2
                                                   : Point )
                             end; { end case }
   var
      Object :  Geometric_Object;
```

The field variable called `Figure` serves as a variant selector for defining the properties of the declared variable `Object`. It is only during execution that one of four fields of the variant record, `Rectangle`, `Circle`, `Triangle`, or `Point_to_Point`, can be selected, depending on the value assigned to `Figure`. Once a field in a variant record is active, only that field and no other variant-record field can be referenced. This is different from the fixed-record fields, where all fields are considered active during execution. The short program titled `Choosing_Geometric_Shapes` allows the user to select the drawing of a geometric shape. This Macintosh Pascal program needs several modifications before it will execute under THINK Pascal. First, the **uses** clause must be removed. Next we must change the identifier `Object`, because **Object** is a reserved word in THINK Pascal. We can do this quickly with the **Find** and **Replace** command options under the THINK Pascal **Search** menu. For example, you might change each occurrence of Object in the program to Object1 or Data_Object.

```
program Choosing_Geometric_Shapes(input, output);
{ Purpose:  This program allows the user to choose one of }
{           several geometric shapes. }
   uses
      QuickDraw1;
   type
      Geometric_Shape = (Rectangle, Circle, Triangle,
                         Point_to_Point);
      Geometric_Object = record
        case Figure : Geometric_Shape of
           Rectangle : ( Boundary : Rect );
           Circle : ( Center : Point;  Radius : integer );
           Triangle : ( Corner_1, Corner_2, Corner_3 : Point );
           Point_to_Point : ( Point_1, Point_2 : Point )
        end;
   var
      Object : Geometric_Object;
      Choice : char;
{ ****************************************************** }
   procedure Set_Text_Window;
      var
         Window : Rect;
   begin
      HideAll;
      SetRect(Window, 100, 100, 400, 300);
      SetTextRect(Window);
      ShowText
   end;
 { ****************************************************** }
   procedure Choose_Figure (var Response : Char);
      var
         Condition : Boolean;
   begin
      repeat
         writeln;
         writeln(' Select a geometric object for display: ');
         writeln('  [ R ] : Rectangle ');
         writeln('  [ C ] : Circle');
         writeln('  [ T ] : Triangle');
         writeln('  [ P ] : Point to point');
         write(' Enter letter R, C, T, or P: ');
         read(Response);
         writeln;
         Condition := (Response = 'R') or (Response = 'C') or
                         (Response = 'T') or (Response = 'P');
      until Condition
   end;
{ ****************************************************** }
```

```
  procedure Assign_Figure (Choice : char;  var Object :
              Geometric_Object);
  begin
     with Object do
        case Choice of
           'R' :    begin
                       Figure := Rectangle;
                       SetRect(Boundary, 100, 50, 300, 200);
                    end;
           'C' :    begin
                       Figure := Circle;
                       Center.v := 250;
                       Center.h := 150;
                       Radius := 80;
                    end;
           'T' :    begin
                       Figure := Triangle;
                       Corner_1.v := 50;
                       Corner_1.h := 50;
                       Corner_2.v := 100;
                       Corner_2.h := 200;
                       Corner_3.v := 300;
                       Corner_3.h := 300
                    end;
           'P' :    begin
                       Figure := Point_to_Point;
                       Point_1.v := 50;
                       Point_1.h := 50;
                       Point_2.v := 250;
                       Point_2.h := 250
                    end
        end;
  end;
{ ********************************************************** }
  procedure Draw_Figure (Choice : char; Object :
              Geometric_Object);
     var
        Window : Rect;
  begin
     HideAll;
     SetRect(Window, 0, 20, 512, 342);
     SetDrawingRect(Window);
     ShowDrawing;
     PenSize(2, 2);
        with Object do
           case Choice of
              'R' : begin
                       FillRect(Boundary, gray);
                       FrameRect(Boundary);
                    end;
              'C' :  PaintCircle(Center.v, Center.h, Radius);
```

```
                 'T' : begin
                          DrawLine(Corner_1.v, Corner_1.h,
                             Corner_2.v, Corner_2.h);
                          DrawLine(Corner_2.v, Corner_2.h,
                             Corner_3.v, Corner_3.h);
                          DrawLine(Corner_3.v, Corner_3.h,
                             Corner_1.v, Corner_1.h)
                       end;
                 'P' :  DrawLine(Point_1.v, Point_1.h, Point_2.v,
                             Point_2.h);
              end;
   end;
{ ************************************************************ }
begin  { Body of the main program. }
   Set_Text_Window;
{ Prompt user to choose one of four figures. }
   Choose_Figure(Choice);
{ Select the field variant for the figure chosen.}
   Assign_Figure(Choice, Object);
{ Draw the figure chosen by the user. }
   Draw_Figure(Choice, Object);
end.
```

Though procedure `Choose_Figure` allows the user to choose one of four figures, procedure `Assign_Figure` uses the value of `Choice` represented by a single character to dynamically select one of the four parts of the variant a record. Notice that in defining the user-defined record `Geometric_Object`, two special Macintosh Pascal types are employed: **Rect** and **Point**. Although we do not need to declare **Rect** and **Point** explicitly, internally *they are of themselves variant records having the following structures :*

```
Rect =   record
            case integer of
               0 : ( top : integer;  left : integer;
                        bottom : integer;  right :integer );
               1 : ( topleft : Point;  botright : Point );
         end;

Point =  record
            case integer of
               0 : ( v : integer;  h : integer );
               1 : ( vh : array[ VHSelect ] of integer );
         end;
```

There are numerous rules for using the variant record. First, all of the field identifiers must have unique names within the record being declared, regardless of the variant. This is why the field `Triangle` uses the identifiers `Corner_1`, `Corner_2`, and `Corner_3` rather than `Point_1` and `Point_2` (since the former three identifier names have already been chosen). Duplication is not allowed, even though the variant field is selected dynamically during execution. The exception is for a record declaration containing one or more nested records. Second, at least one of the values of the variant selector must be labeled in the **case** statement and cannot share constants of any other type. Third, a syntax error occurs if no labels exist within the **case** statement of the variant record. Fourth, a nonactive variant record has all of its fields undefined. Referencing any field will result in an execution error. Fifth, the identifier of the variant selector cannot be used as an actual parameter for a corresponding formal parameter.

9.7 A STRUCTURE FOR CONTAINING A RANDOM SET OF ELEMENTS: THE PASCAL SET

Pascal also supports a special structure called a *set*. As an abstract structure, a set represents an object where elements of the same type are stored randomly. In Pascal a set can be declared as a user type in terms of an `ordinal` type. The format for declaring a set follows:

```
type
   Set_Name = set of ordinal_type;
```

For example, consider an `ordinal` type called `Names_of_Days` composed of the days Sunday through Saturday. We wish to establish a special set called `Days` and a variable called `Workweek` of type `Days`. Here are these declarations:

```
type
   Names_of_Days = ( Sunday, Monday, Tuesday, Wednesday, Thursday,
                     Friday, Saturday );
   Days = set of Name_of_Days;
var
   Workweek : Days;
   Weekend, Midweek : Days;
```

First, `Names_of_Days` represents an `ordinal` type where the days of the week are ordered from the smallest (Sunday) to the largest (Saturday). Second, `Days` is declared as a set of `Names_of_Days`. This means that the variables `Workweek`, `Weekend`, and `Midweek`, although of type `Days`, are each themselves a set. During execution each set can contain either no elements (an empty set) or one or more of the ordinal values of `Names_of_Days`. The variables `Workweek` and `Weekend` are initially assigned as empty sets at the beginning of program execution.

Only by executing an assignment statement can we add elements to a set. The following statements provide some examples of initializing sets with elements.

```
{ Initialize the set Workweek with elements Monday..Friday.}
   Workweek := [ Monday..Friday ];
{ Initialize the set Weekend with elements Sunday, Saturday. }
   Weekend  := [ Saturday, Sunday ];
{ Initialize Midweek as an empty set. }
   Midweek     := [ ];
```

An expression of type set is represented by a list of ordinal values within a pair of square brackets. For compatibility, the variable on the left side of an assignment statement must also be a set type, with the ordinal values contained within the expression also declared within the set type. During execution the value of the set type variable is replaced by the value of the expression on the right. For instance, if the variable `Workweek` contained the set `[ Sunday, Friday, Saturday ]`, execution of our sample assignment statement would create the set `[ Monday, Tuesday, Wednesday, Thursday, Friday ]`.

How then can elements be added to or removed from a set without destroying the value of a set variable? We do this by using one or more of the three set operators shown in Figure 9.14.

Operator	Name	Action
*	Set intersection	Results in a set composed of elements common only to the two intersecting sets.
+	Set union	Results in a set composed of elements from the union of both sets.
-	Set difference	Results in a set composed of members of the first set and not members of the second set.

Figure 9.14 Set operators in Macintosh and THINK Pascal.

Consider the following Pascal statements as examples of these operations:

```
   Workweek := [ Monday..Friday ];
   Weekend  := [ Saturday, Sunday ];
{ Compute the intersection of Workweek and Weekend. }
   Midweek  := Workweek * Weekend;
{ Compute the union of Workweek and Weekend. }
   Workweek := Workweek + Weekend;
{ Compute a new value for Midweek. }
   Midweek := Workweek * Weekend;
{ Decrease the number of days in Workweek. }
   Workweek := Workweek - [ Friday, Saturday, Sunday ];
```

The first two statements initialize `Workweek` and `Weekend`. The third statement results in `Midweek` being assigned as an empty set, because the sets `Workweek` and `Weekend` contain no common elements. The fourth statement results in the union of the elements `Monday..Friday` with `Saturday` and `Sunday`, the result being assigned as the set `[Sunday..Saturday]` to `Workweek`. The fifth statement takes the intersection of the sets `Workweek` and `Weekend`. The common elements are `Saturday` and `Sunday`, so `Midweek` now becomes the set `[Sunday,Saturday]`. The last statement subtracts the elements `Friday`, `Saturday`, and `Sunday` from the set `Workweek`, yielding a smaller set `[Monday..Thursday]`.

There are also several relational operators for comparing sets, shown in Figure 9.15. Syntactically the sets being compared must be type-compatible for the expression to be translated and executed. In the case of testing for membership, the element must be an `ordinal` type, and its value must be compatible with the `ordinal` types for members of the set being tested.

Operator	Name	Action
=	Set equality	*A* = *B* returns *true* if every member of *A* is in *B* and every member of *B* is in *A*.
< >	Set inequality	*A* <> *B* returns *true* if members of *A* are not members of *B* and members of *B* are not members of *A*.
< =	Is contained by	*A* <= *B* is true if all elements of *A* are also elements in *B*.
> =	Contains	*A* >= *B* is *true* if all elements of *B* are also elements in *A*.
in	Member of	Element **in** *A* is *true* if the element is a member of *A*.

Figure 9.15 `Boolean` operators in Pascal that apply to sets.

For example, consider the steps of a program that allows you to enter several lines of text from the keyboard, count the characters `a` through `z` , both for upper- and lowercase, and, as shown in Figure 9.16, display in the Drawing window a distribution chart of the character count.

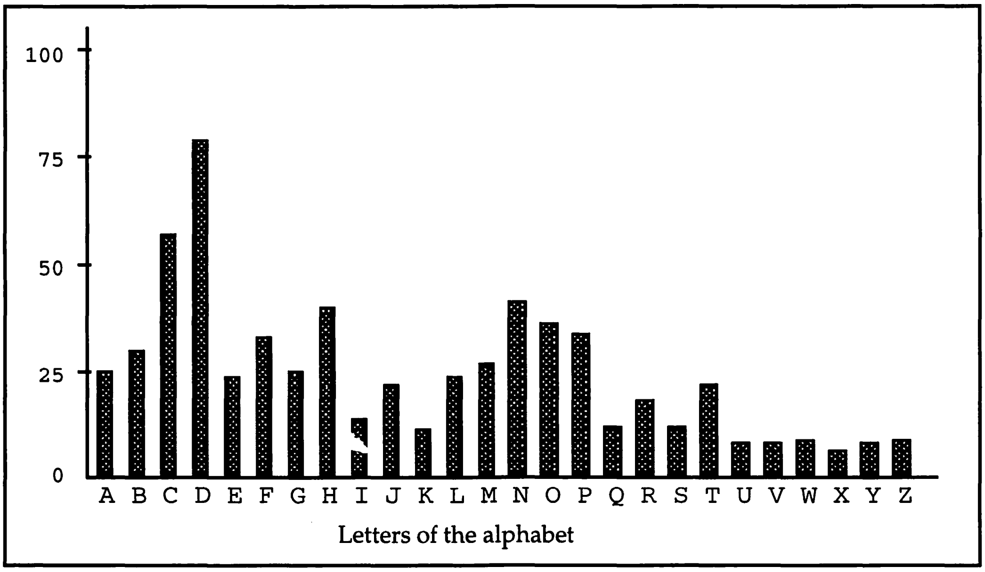


Figure 9.16 Distribution chart of letters of the alphabet read from text entered from the keyboard.

The main module has the following four steps:

1. Set the Text window for viewing.
2. Initialize two arrays, one that stores uppercase letters and one that stores lowercase letters.

3. Prompt the user to enter a message and then, reading one character at a time, count the alphabetic letters.
4. Report the total counts from Step 3 by drawing a distribution chart.

The modules for the first two steps are simple; you can examine them in the program `Distribution_Chart`. The third step uses lazy input from the keyboard for reading and counting alphabetic characters. The algorithm for this module follows.

```
begin
{ Prompt user for a message indicating that the character ' # '
  will terminate counting characters. }
   repeat
   { reading the next line }
      repeat
      { Read the next character, test, and adjust counter. }
         read( Character );
         Test if Character is in the set of uppercase letters and,
         if so,  increment the total count for that letter by 1;
         if not, test if Character is in the set of lowercase
         letters and, if so, increment the total count for that
         letter by 1. }
      until {  reached the end of line or Character = '#' } ;
   { Terminate the present input line. }
      readln;
   until ( Character = '#' );
end;
```

Why is this referred to as *lazy input*? Although a Pascal system can recognize when it has reached the end of a line (the user has pressed the Return key), it cannot detect if the user has typed additional lines of text. In this type of input mode, the Pascal system cannot look ahead to see if it has reached an end-of-file from the keyboard. In our example, the character # is used as a marker to indicate an end to input. The procedure `Read_and_Count` from the program `Distribution_Chart` shows these steps for counting the alphabetic characters. In this module the formal parameter `Table_A` represents the table `Uppercount`, storing the counts of uppercase letters of the alphabet, and the formal parameter `Table_B` represents the table `Lowercount`, storing the counts of lowercase letters of the alphabet. Both parameters `Table_A` and `Uppercount` are of the type `Uppercase_Count ( `**`array`**`['A'..'Z'] `**`of`**` integer)`, and both `Table_B` and `Lowercount` are of the type `Lowercase_Count ( `**`array`**`['a'..'z'] `**`of`**` integer)`. As you can see with both sets of parameters, the ordinal values `'A'..'Z'` and `'a'..'z'` are used as subscript values for arrays rather than integer values. This is more in line with defining a solution to our problem.

For reporting the total counts to the Drawing window, we use the following steps:

1. Set the Drawing window for viewing, and open this window.
2. Draw and label the vertical boundary of the distribution chart.
3. Draw and label the horizontal boundary of the distribution chart.
4. Compute the total count for each letter of the alphabet.
5. Draw the bars for each total count.

The first step is similar to procedure Set_Text_Window. In the remaining steps the Drawing window is assumed to be approximately 500 pixels wide by 300 pixels high, each pixel representing a point. Initially a vertical line is drawn from the point (70, 10) to the point (70, 220). The drawing pen is moved to point (13, 224), and the vertical axis of the chart is labeled with constants 0, 25, 50, 75, and 100. Each time a new label is drawn, the y-coordinate of the drawing point is decremented by 50, and the pen is moved to a new vertical position. Except for the label 0, all labels are drawn with a short dash intersecting the horizontal axis. Here are the necessary program lines:

```
DrawLine( 70, 10, 70, 220);
Number := 0;
X := 13;
Y := 224;
for J := 1 to 5 do
   begin
      MoveTo( X, Y);
      WriteDraw( Number );
      if J > 1 then
         WriteDraw(' -');
      Y := Y - 50;
      Number := Number + 25;
   end;
```

While drawing the horizontal axis, Y remains constant, and X changes. Initially a line is drawn from the point (70, 220) to the point (470, 220). The length of this line is based on the assumption that each count to be drawn will have a bar 8 pixels wide with a 7-pixel gap before drawing the next bar. At the initial point (77, 230) the first letter A is drawn, and at each horizontal point 15 pixels to the right, the successor of the previous alphabetic letter is drawn. The following shows these steps:

```
DrawLine( 70, 220, 470, 220);
X := 77;
Y := 230;
for Letter := 'A' to 'Z' do

   begin
      MoveTo( X, Y);
      WriteDraw( Letter );
      X := X + 15
   end;
MoveTo( 200, 250);
WriteDraw( 'Letters of the Alphabet');
```

The last step involves drawing each of the individual bars. This first involves moving to an initial point (79, 220) represented by X and Y, respectively. For each letter count, we use the routine **SetRect** to establish a rectangle with the left top point of the rectangle defined as (X-4, Y-2 * Total_Count[Letter]) and the bottom right point defined as (X + 4, Y). Notice that the factor 2 is used in computing the top of the left top point. We have assumed a scaling factor of 2, so for every pixel to be drawn, we will multiply by 2. This limits us to an approximate range 0 through 100. Removing this factor would double the vertical scale. This would also require adjustment in labeling our vertical axis. At this point the rectangle is drawn and filled with a background pattern using **FillRect**, and then framed using **FrameRect**. Next the drawing point is moved to the right by 15 pixels for drawing the next bar. The following program lines implement these steps:

```
X := 79;
Y := 220;
   for Letter := 'A' to 'Z' do
      begin
         SetRect(Bar,X - 4,Y - 2 * Total_Count[Letter],X + 4, Y );
         FillRect( Bar );
         FrameRect( Bar );
         X := X + 15
      end;
```

The program Distribution_Chart contains the complete listing for executing the foregoing steps. This Macintosh Pascal program will execute under THINK Pascal if the **uses** clause is removed.

```
program Distribution_Chart(input, output);
{ Purpose:  To count the alphabetic letters contained }
{           in a message and then to draw a distribution chart }
{           of the characters counted. }
   uses
      QuickDraw1;
   type
      Letters_of_Alphabet = set of char;
      Uppercase_Count = array['A'..'Z'] of integer;
      Lowercase_Count = array['a'..'z'] of integer;
      Table = array['A'..'Z'] of integer;
   var
      Uppercount : Uppercase_Count;
      Lowercount : Lowercase_Count;
      Uppercase_Set, Lowercase_Set : Letters_of_Alphabet;
      Index_1, Index_2 : char;
{ ******************************************************** }
   procedure Set_Text_Window;
      var
         Window : Rect;
   begin
      HideAll;
      SetRect(Window, 0, 40, 512, 342);
      SetTextRect(Window);
      ShowText
   end;
{ ******************************************************** }
   procedure Initialize_Sets (var Set_A, Set_B :
                                 Letters_of_Alphabet);
   begin
      Set_A := ['A'..'Z'];
      Set_B := ['a'..'z']
```

```
  end;
{ *********************************************************** }
  procedure Read_and_Count (var Table_A : Uppercase_Count;
                            var Table_B : Lowercase_Count;
                            Set_A, Set_B : Letters_of_Alphabet);
     var
        Character : char;
  begin
  { Prompt the user for a message. }
     write('Enter the message for character counting.');
     writeln(' Terminate by typing the character #  :');
     repeat
     { reading from the next line }
        repeat
        { reading one character at a time until reaching an }
        { end of line }
           read(Character);
           if Character in Set_A then
              Table_A[Character] := Table_A[Character] + 1
           else
              if Character in Set_B then
                 Table_B[Character] := Table_B[Character] + 1;
        until eoln or (Character = '#');
     { Terminate read line and begin reading the next line }
     { unless the termination character has been read.}
        readln;
     until (Character = '#');
     writeln;
  end;
{ *********************************************************** }
  procedure Set_Drawing_Window;
     var
        Window : Rect;
  begin
     HideAll;
     SetRect(Window, 0, 40, 512, 342);
     SetDrawingRect(Window);
     ShowDrawing
  end;
{ *********************************************************** }
  procedure Draw_Vertical_Axis;
     var
        X, Y, Number, J : integer;
  begin
     DrawLine(70, 10, 70, 220);
     Number := 0;
     X := 13;
     Y := 224;
     for J := 1 to 5 do
        begin
           MoveTo(X, Y);
```

```
            WriteDraw(Number);
            if (J > 1) then
               WriteDraw(' -');
            Y := Y - 50;
            Number := Number + 25
         end;
   end;
{ ********************************************************** }
   procedure Draw_Horizontal_Axis;
      var
         X, Y : integer;
         Letter : char;
   begin
      DrawLine(70, 220, 470, 220);
      X := 77;
      Y := 230;
      for Letter := 'A' to 'Z' do
         begin
            MoveTo(X, Y);
            WriteDraw(Letter);
            X := X + 15
         end;
      MoveTo(200, 250);
      WriteDraw('Letters of the Alphabet');
   end;
{ ********************************************************** }
   function Compute_Total_Counts (A : Uppercase_Count;
                                 B : Lowercase_Count) : Table;
      var
         Index_1, Index_2 : char;
         Temp_Array : Table;
   begin
      Index_1 := 'A';
      Index_2 := 'a';
      while Index_1 <= 'Z' do
         begin
            Temp_Array[Index_1] :=
                        Uppercount[Index_1]+Lowercount[Index_2] ;
            Index_1 := succ(Index_1);
            Index_2 := succ(Index_2);
         end;
         Compute_Total_Counts := Temp_Array
   end;
{ ********************************************************** }
   procedure Draw_Bars (Total_Count : Table);
      var
         X, Y : integer;
         Letter : char;
         Bar : Rect;
   begin
      X := 79;
      Y := 220;
      for Letter := 'A' to 'Z' do
         begin
            SetRect(Bar, X - 4, Y - 2 * Total_Count[Letter], X +
                     4, Y);
            FillRect(Bar, gray);
            FrameRect(Bar);
            X := X + 15
         end;
   end;
{ ******************************************************** }
   procedure Report_Counts (A : Uppercase_Count;
                            B : Lowercase_Count);
      var
         Total_Count : Table;
   begin
   { Set Drawing window and open window for viewing. }
      Set_Drawing_Window;
   { Draw and label the vertical boundary of the distribution }
   { chart. }
      Draw_Vertical_Axis;
   { Draw and label the horizontal boundary of the distribution }
   { chart. }
      Draw_Horizontal_Axis;
   { Compute the total count for the alphabetic letters A }
   { through Z. }
      Total_Count := Compute_Total_Counts(A, B);
   { Draw the bars representing the counts for the letters A }
   { through Z. }
      Draw_Bars(Total_Count)
   end;
{ ******************************************************** }
begin { Body of the main program. }
{ Set Text window for viewing and display Text window. }
   Set_Text_Window;
{ Initialize character sets Uppercase and Lowercase. }
   Initialize_Sets(Uppercase_Set, Lowercase_Set);
{ Prompt user for paragraph, then read and count letters of }
{ the alphabet. }
   Read_and_Count(Uppercount, Lowercount,
                  Uppercase_Set, Lowercase_Set);
{ Report the character count to the user. }
   Report_Counts(Uppercount, Lowercount);
end.
```

As you can see, each step of the procedure Report_Counts calls on a module for executing the details. This approach allows for easier maintenance if we need to modify the algorithm for labeling either the horizontal or vertical axis, or both, or for modifying the steps in drawing bars.

9.8 PACKED ARRAY OF CHARACTERS

Standard Pascal supports the word **packed** in its declaration of structured types such as arrays, sets, and records. For example, the following packed types can be declared:

```
type
   Names_of_Days = ( Sun, Mon, Tues, Wed, Thurs, Fri, Sat );
   Matrix = packed array[1..10] of integer;
   Table = packed set of Names_of_Days;
   Personnel = packed array[1..100] of
                      record
                         Name : packed array[1..80] of char;
                         ID_Number : longint;
                         Wage : real
                      end;
```

Packed types economize on storage by allowing the storage of values to be compressed at the expense of accessing a value. Pascal has two special procedures for packing and unpacking structure types. The procedure `pack( A, J, S )` copies all of the elements of the unpacked array `A` to packed array `S`, starting at the *j*th position of array `A`. The second procedure, `unpack( S, A, J )`, copies all of the elements of the packed array `S` to the unpacked array `A` starting with the *j*th position of array `A`. Neither of these procedures offers any advantages for a packed array of characters in Macintosh Pascal, because each character of a packed array is stored within a single byte of memory.

Like many other implementations of Pascal on personal computers, in Macintosh Pascal the attribute **packed** will result in the packing of only one structure type: **packed array of** `char`. Though other structures can be declared as packed, only the packed array of characters is acted on by the translator. A packed array of characters is an alternative to declaring a `string` of a fixed length. For example, the declaration

```
var
     String_1 = string[80] ;
```

represents a `string` variable capable of storing a maximum of 80 characters. Initially the variable `String_1` has a zero length; that is, `String_1` is initially assigned a null string. An alternative to this declaration is that of a packed array of characters:

```
var
     String_2 = packed array[1..80] of char;
```

This also represents a string but in the context of an array, and with each array element being a `char` type. Initially all elements of the packed array of characters have a null character. What is the difference between the variables `String_1` and `String_2`? This can be seen by extracting individual characters from the string. As the program `Packing_Characters` shows, individual characters from `String_1` and `String_2` are accessed by using subscripted variables. This program will execute under both Macintosh Pascal and THINK Pascal.

```
program Packing_Characters(input, output);
{ Purpose:  This program compares the concept of the string }
{           type and packed array of characters. }
   var
      String_1 : string[80];
      String_2 : packed array[1..80] of char;
      J : integer;
begin
{ Enter a first string from the keyboard. }
   writeln(' Enter a sentence for the first string: ');
   readln(String_1);
   writeln;
{ Enter a second string from the keyboard. }
   writeln('Enter a sentence for the second string: ');
   J := 1;
   while not eoln and (J <= 80) do
      begin
         read(String_2[J]);
         J := succ(J)
      end;
   readln;
{ Display the first string one character at a time. }
   writeln;
   writeln('String_1: ');
   for J := 1 to Length(String_1) do
      write(String_1[J]);
   writeln;
{ Display the second string one character at a time. }
   writeln;
   writeln('String_2: ');
   for J := 1 to 80 do
      write(String_2[J]);
   writeln;
end.
```

Whereas String_2 is stored as a static structure, String_1 is stored dynamically in a format composed of its length and the actual characters. It is impossible to access any characters in String_1 beyond its actual length. Another difference between the two types of strings is the maximum string size. String types are limited to a maximum of 255 characters, whereas a packed array of characters is limited only by the memory allowed during execution. A packed array of characters can thus be declared beyond the limit of 255.

A third difference is found in compatibility problems with a packed array of characters. For example, the statement

```
if String_1 = 'A'  then statement;
```

is executable, independent of the actual string length of String_1. In this context, both String_1 and 'A' take on the appearance of string types, with 'A' being represented as a string type of length 1. Unfortunately, the statement

```
if String_2  = 'A' then statement;
```

causes a compatibility error when executed, because String_2 in this context is an array structure, whereas 'A' is a simple char type. In general, string types, characters, and packed arrays of characters can be compared with string types. For the statement

```
if String_1 =  String_2 then statement;
```

the value of String_2 having *N* non-null characters is converted into a string type value of length N. In order to compare two packed arrays of characters, both must have values with an equal number of non-null characters for compatibility. For an assignment statement, a string type can take on the value of an expression if the expression is a string type, a packed arrays of characters, or a char type. If both the left and right parts are packed arrays of characters with the same number of non-null characters, the assignment statement is compatible.

String types offer ease of reading for characters entered from the keyboard. In the program Packing_Characters, we only need to execute the statement readln(String_1) to enter the complete string from the keyboard. With packed arrays of characters, we must use lazy input for assigning characters to String_2. Any attempt to execute the command read(String_2) or readln(String_2) causes Macintosh Pascal to report an error indicating that the argument String_2 must be an ordinal type or a string type; it cannot be a packed array of characters. Fortunately, no difficulty exists for displaying a packed array of characters using the commands write or writeln.

What then is the difference between a packed array of characters and an array of characters? In Macintosh Pascal each character within a packed array is allocated a single byte of memory, whereas each character within an array of characters is allocated two bytes of memory. This is somewhat puzzling because a char type in Macintosh Pascal requires only a single byte of allocation. It is obvious that when programming in Macintosh Pascal, using a packed array of characters or a string type saves memory compared to using an array of characters.

9.10 STANDARD PASCAL VERSUS THINK PASCAL

In Standard Pascal, a function can only return a simple type or a pointer type. For both THINK and Macintosh Pascal, functions can return structure types, such as an array or record. This allows us to write functions that can perform basic array operations such as addition, subtraction, and matrix multiplication, and return values that are structures. The following function is a simple example defining the addition of two vectors A and B, both of length N:

```
function Add_Vectors(A, B:Vector; N:integer):Vector;
   var
      C : Vector;
      I : integer;

begin
   for I := 1 to N do
      C[I] := A[I] + B[I];
   Add_Vectors := C;
end;
```

Standard Pascal treats all strings as packed arrays of characters. While a packed array of characters can be written to standard output using either a `write` or `writeln` statement, a packed array can only be read character by character. In Standard Pascal attempting to directly read a string declared as a packed array using a `read` or `readln` statement is either syntactically or semantically illegal. For example, the following `readln` statement in THINK Pascal can be used to read several characters into a packed array of characters:

```
write('Please type a single word: '):
readln( Str );
```

where `Str` is assumed to be declared as a **packed array**`[ 1 .. 40]` **of** `char`. In Standard Pascal (as well as Macintosh Pascal) a string such as `Str` can only be read by using a loop, character by character. Here an index variable is needed for assigning each character that is read its proper index position within a packed array of characters. A special function called `eoln` (short for end-of-line) determines when the Return key has been pressed. As long as the Return key is not pressed, it will continue to be *false*, and the loop continues to read characters and assign them to the packed array. The following demonstrates how Standard Pascal (as well as Macintosh Pascal) reads a string stored as a packed array of characters:

```
I := 1;
while  not eoln(input) do
   begin
      read( Str[I] );
      I := succ(I);
   end;
readln;
```

In both examples we assume that we never exceed the limit of 40 characters.

Although a `string` type may appear to have the same structure as a packed array of characters, they are actually different structures. A packed array of characters has a static storage structure and is assumed to use all of the index positions that are given within its declaration. This means that whether you assign 4 characters or 40 characters to a packed array of 40 characters, it uses all 40 character positions when storing and displaying the value of a string. In some instances a Pascal compiler might initially pack the array positions with either null characters or blank characters. Standard Pascal has no definite rules. In many instances, if the `string` variable declared as a packed array of characters is reassigned a string, older characters not overwritten by the new string may remain in the array as garbage. The rule in working with packed array of characters is to allow the last character position as a string terminator. Often the null character is convenient, because it cannot be seen when it is displayed. This requires special `write` routines for displaying the characters in a string one character at a time until the null character is accessed.

A `string` type is assumed to have an implied record structure composed of three elements: the actual length of the string, the maximum length that is allowed, and a packed array of characters for storing the string. The following demonstrates the structure for the the type **string**`[M]`:

```
String_M =  record
               Actual_Length:integer;
               Maximum_Size:M;
               String_Array:packed array[1 .. M] of char;
            end;
```

For variant records, THINK Pascal does not enforce all of the requirements of Standard Pascal. First, THINK Pascal does not require that every variant of the tag-field be in the case-constant-list. For example, the following variant record would require all four tag-field values to appear in a Standard Pascal declaration, whereas in THINK Pascal we can delete the case constant Vanilla.

```
type
   Flavor = ( Vanilla, Cherry, Chocolate, Banana );
   Ice_Cream_Cone =  record
                        case Flavor of
                           Chocolate:(Height:integer);
                           Cherry:(Number:integer);
                           Banana:(Peels:integer);
                           Vanilla:()
                        end;
```

Although Standard Pascal does not allow for the tag-field of a variant-part as an argument of a variable-parameter, THINK Pascal does not enforce this rule. Thus, procedure Choose_Figure in Section 9.6 might be rewritten so that the formal parameter Response remains a variable parameter, but its type is now Geometric_Shape rather than character, and so that in the body of the procedure one of the enumerated values can be assigned directly to the formal parameter. A call to procedure Choose_Figure would then have as an actual parameter the subscripted variable Object.Figure.

SUMMARY

Pascal allows several different structures for organizing information. The array is a structure in which all elements are homogeneous. Accessing an array element is performed through a subscripted variable, with the subscripts being ordinal types. The ordinal type can go beyond simple integer types and can include subrange values, char types, and values of user-defined ordinal types.

The record presents us with a second type of structure, one that allows in-homogeneous elements. Through the specification of fields, the record provides a structure for storing several different pieces of information. To access a field of a record, list the record name, followed by a period, followed by the field identifier. This acts as the equivalent of a subscripted variable. The record as a structure provides a more natural object for defining a solution to a problem and for presenting a solution to the computer. The structure of a record can be extended to include a variant-record part. The variant-record part allows a portion of a record to become active during execution. Although several variables can be declared to have the same record type, it is at execution time that different fields can be made active, depending on the value of a variant selector.

A set is a structure in which elements are assumed to be stored in random order. Pascal supports the concept of a set, but in reality the values allowed to be stored in sets are defined through `ordinal` types. This forces all sets in Pascal to have a specific ordering that is not random. The power of sets can be expanded through the use of operators for intersection, union, and set difference, and through relational operators. Keep in mind that declaring a set of records or a set of arrays is *not* allowed in Pascal.

The packed array of characters is a special structure for the representation of a string. Macintosh Pascal supports an explicit `string` type, but in many Pascal translators character strings are represented by packed arrays. In Macintosh Pascal, the `string` type is a dynamic structure that can increase or decrease in size as the program is executed. Unfortunately, Macintosh Pascal limits a `string` type to a maximum of 255 characters. A packed array of characters allows you to go beyond that limit, using a static string structure.

Applications often include various sort and search routines. The quicksort algorithm provides one of the fastest techniques for sorting data stored internally in RAM. The binary search algorithm is one of the most popular techniques for searching a presorted table by using a key.

REVIEW QUESTIONS

1. What is the purpose of having an array in a computing language?
2. What are the two ways in which a one-dimensional array can be viewed?
3. What is the form for declaring a one-dimensional array?
4. How can a `subrange` type be used to declare a one-dimensional array?
5. How many elements (array positions) exist for the following array declarations?

```
(a)  var
      T : array[-10..20] of real;
(b)  var
      X : array[10..25] of integer;
(c)  var
      S : array[-20..-6] of char;
```

6. What is meant by the term *subscripted variable*?
7. For the declarations given in Question 5, which of the following represent valid subscripted variables?

```
T(-6)     X(i)     X[i+j)     T[k]     S[0]     T[5-i]
```

8. Why does Pascal use square brackets instead of parentheses to surround an index?
9. What types of indices other than integers can be used in declaring an array and in specifying the index of a subscripted variable?

10. Are the following declarations syntactically correct?

```
type
   Months_of_Year = ( Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep,
                       Oct, Nov, Dec );
   Range_of_Months = (Jan..Dec);
```

```
var
   Table_of_Dates  :  array[Range_of_Months]  of  string;
```

11. Is the following procedure-header syntactically correct?

```
procedure   Experiment_Four( var Table: array[1 .. 7] of real);
```

Explain your answer. How can you check your answer to support your position?

12. If the procedure-header in Question 11 is wrong, what must be added to correct the problem?
13. What is meant by the term *multidimensional array*?
14. What is the special name given to a two-dimensional array?
15. Write the format for declaring a two-dimensional array.
16. Consider the following declaration of a two-dimensional array:

```
var
   A : array[-10..10, 5..15] of integer;
```

How many rows are there? What is the range of row indices? How many columns are there? What is the range of column indices? How many integer numbers does array A store in memory?

17. Are the following declarations syntactically correct?

```
type
   Row_Range = 10..100;
   Column_Range = 5..50;
var
   A : array[Row_Range, Column_Range] of integer;
```

18. Are the following declarations syntactically correct?

```
type
   Months_of_Year = ( Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep,
                      Oct, Nov, Dec );
   Range_of_Months = (Jan..Dec);
   Range_of_Days = 1..31;
var
   Message_Dates:array[Range_of_Days,Range_of_Months] of
                        string;
```

19. How can the declaration for `Message_Dates` in Question 18 be modified to handle the years 1987 through 1999 as well as months and days of the month?
20. For the declaration

```
var
   X : array[1..10, 9..25] of real;
```

are the following subscripted variables syntactically correct?

```
X[i],[j]        X[5,k]        X[k][p]        X(j,t)
```

21. What is meant by the concept of *an array of arrays*?
22. Why is it important that data stored within an array be sorted?
23. What advantage is there in using the bubble sort algorithm?
24. Show the steps for swapping two array elements in a one-dimensional array.
25. Write a procedure for swapping two rows of a two-dimensional array.
26. What is the basic difference between the bubble sort algorithm and the straight insertion sort?
27. Why does Shellsort offer an advantage over the bubble sort and the straight insertion sort?
28. Why is the quicksort one of the best internal sort algorithms?
29. What is the primary difference between the linear search algorithm and the binary search algorithm?
30. Why is a searching algorithm like the binary search algorithm important for finding data within an array?
31. Why can the binary search algorithm fail if the array being searched has not been sorted?
32. What is meant by the terms *homogeneous arrays* and *inhomogeneous arrays*?
33. In Pascal, how can an inhomogeneous array be represented?
34. Define the format for declaring a user-defined type called a *record.*
35. Define the structure for the internal record type **DateTimeRec**.
36. For the declaration

```
var
   A :   record
            Name : string[30] ;
            Age : integer;
            SSnumber : string[10] ;
         end;
```

how is the field Name referenced? How is the field Age referenced? How is the field SSnumber referenced?

37. For the declaration

```
var
   A : array [1..100] of   record
                              Name : string[30] ;
                              Age : integer;
                              SSnumber : string[10] ;
                           end;
```

how is the *j* th element of array A referenced? How is the field Age of the *i*th element of array A referenced? Are the references `A.SSnumber[i]` and `A[i].SSnumber` equivalent?

38. What statement in Pascal can be used to reduce the explicit referencing of a record name when accessing the fields of a record?
39. When is it not possible to remove the referencing of a record variable from the body of the **with** statement?
40. What extension does Pascal support for declaring a record?
41. What rules must we follow when writing a record having a fixed part and a variant part?

42. How can the **case** construct be used with the declaration of a record?
43. Define the **QuickDraw1** types **Rect** and **Point** using the **case** construct.
44. How does a set differ from an array and a record structure?
45. How is an empty set assigned to a variable declared as a set of a data type?
46. For the declarations

```
type
   Months = ( Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,
              Nov, Dec );
var
   Dates : Months;
   Summer_Months : Months;
```

is it true that the variable `Dates` has the elements `Jan..Dec`?

47. Show how `Summer_Months` in Question 46 can be assigned the values `Jun, Jul, Aug, Sept`.
48. List the set operators and the relational operators that can be applied to sets.
49. Can `read` and `readln` commands be used to enter sets from the keyboard?
50. Can `write` and `writeln` commands be used to output sets to the Text window?
51. Do the three set operators union, intersection, and difference have operator precedence among themselves? with arithmetic and `Boolean` operators?
52. What is important about the data type *packed array of characters*?
53. Is the data type **packed array**`[1..30]` **of char** equivalent to the data type **string**`[30]`? Explain your answer.
54. What is the purpose of the procedures `Pack` and `Unpack`?
55. Is there any benefit in Macintosh Pascal in declaring a packed array of integers, a packed array of reals, or a packed array of Booleans?
56. What is meant by the statement that a subscript is out of range?
57. Consider the following declarations:

```
type
   Matrix  =  array[1..100, 1..10] of integer;
   Table   =  array[1..10] of Matrix;
var
   T : Table;
```

What is the data type for the variable `T`? What data type is associated with each element of table `T`? How many elements exist in table `T`? How many integer numbers can be accessed through table `T`?

58. Consider the following declarations:

```
type
   School_Months = ( Jan, Feb, Mar, Apr );
var
   Student :   record
                  ID : string[11];
                  Name : string[20];
                  Year : integer;
```

```
                case Month: School_Months of
                   Jan : ( Jan_Grade : char );
                   Feb : ( Midterm : char;
                            Average : integer );
                   Mar : ( Mar_Grade : char );
                   Apr : ( Final_Grade : char;
                            Final_Average : real )
             end;
```

What represents the fixed part and variant part of a variant-record type?

59. How is one of the four fields Jan, Feb, Mar, Apr made active?
60. Can an inactive field of a variant record be referenced during execution?

PROGRAMMING EXERCISES

Although not all programming exercises require you to write an algorithm, you may better understand the problem and what is required by first writing an algorithm and tracing it by hand with several examples before writing a Pascal program.

1. Consider that daily temperatures are measured at midnight, 6 A.M., noon, and 6 P.M. Develop an algorithm that will prompt for all four of these temperatures for each of the seven days of the week and compute the mean midnight, mean 6 A.M., mean noon, and mean 6 P.M. temperatures for the week. Test your algorithm by writing a program and submitting your own data. *Hint* : Use a two-dimensional array for storing the temperature values.

2. Suppose that table A has N random elements. The following algorithm computes the maximum value of N elements.

```
procedure Max_Value( A : Table, N : integer; var Maximum : item );
{ The  purpose of this module is to compute the maximum value of
   array A, representing a table of N unsorted elements of type
   item.  Parameter Maximum will serve as a variable for returning
   the maximum value to where the module is called. }
   var
      Index : integer;
begin
{ Let Maximum be assigned the first value of array A. }
   Maximum <-- A[1];
   Index <-- 2;
   while  Index <= N do
      begin
      { Check if the next array element is larger than Maximum. }
         if A[Index] > Maximum then
            Maximum <-- A[Index];
         { endif }
         Index <-- succ( Index )
   end { while-do};
end { Max_Value };
```

Write a program using procedures that perform the following steps:

(a) Assign 30 random integers to array A.
(b) Determine the maximum value of these from array A.
(c) Display the values of array A across the Text window and the maximum value.

3. Based on the concepts of Exercise 2, write a new algorithm called Max_and_Min to determine the index positions of maximum and minimum values in an array called A. This new module requires four formal parameters: array A, integer N, Max_Index representing the index position of the maximum value, and Min_Index representing the index position of the minimum value. Modify the program in Exercise 2 for testing this new algorithm by displaying the values of array A as well as the maximum and minimum values, using the relevant index values.

4. Modify the program in Exercise 1 to determine the maximum and minimum temperatures for each day of the week and the maximum and minimum temperatures for the entire week from all temperatures entered upon input.

5. Assume that a one-dimensional array A stores N real numbers. This array is indexed from 1 to N, with N less than or equal to 50. Write a function called Largest_Difference for computing the largest difference between two consecutive elements of A and the index position within matrix A.

6. Consider the following declaration:

```
type
   Matrix = array[1..100, 1..50] of integer;
var
   T : Matrix;
```

Write a procedure called Initial_Elements having three formal parameters: A representing a two-dimensional array, M representing the number of rows, and N representing the number columns. This procedure is required to initialize to 1 all elements in the first M rows and N columns of array A.

7. Write a procedure called Transpose_Array that has three formal parameters: A representing a two-dimensional array, M representing the number of rows, and N representing the number columns. This procedure is required to interchange the rows and columns of array A. After execution of Transpose_Array(B, I, J), the first row of array B has now become the first column, the second row of B has now become the second column, and so on.

8. Assume that a data pair is to be entered from the keyboard. The first part is a student's full name, represented by a string, and the second is an integer number representing an examination score. It is assumed that all examination grades are positive, and input is terminated by entering the string STOP. Student names are to be stored in a one-dimensional array of type **string**[20], and examination names are to be stored in a one-dimensional array of type integer.

Assume that the number of students is limited to a maximum 50. Write an algorithm that will perform the following steps, using procedures:

(a) After hiding all windows, set and open the Text window.
(b) Enter student names and examination scores.
(c) Compute the average of the examination scores.
(d) Using the straight insertion sort, sort the grades from highest to lowest.
(e) Clear the Text window by executing `Page`.
(f) Report the average examination score and the grades and student names to the Text window.

After testing your algorithm, convert it to a Macintosh program, and test to see if it is functional. What happens if the first input is `STOP`? No global variables are allowed.

9. Instead of using two arrays in Exercise 8, replace these with a one-dimensional array of records having the following declarations:

```
type
   Person  =   record
                  Name : string[20];
                  Score : integer
               end;
   Table = array[1..50] of Person;
var
   Student : Table;
```

Rewrite the algorithm and program for Exercise 8 to perform the following steps:

(a) After hiding all windows, set and open the Text window.
(b) Enter student names and examination scores.
(c) Compute the average of the examination scores.
(d) Using the straight insertion sort, sort the grades from highest to lowest.
(e) Clear the Text window by executing `Page`.
(f) Report the average examination score and the grades and student names to the Text window.
(g) Using the straight insertion sort, sort the names of students in alphabetical order.
(h) Clear the Text window by executing `Page`.
(i) Report the student names and grades to the Text window.

No global variables are allowed.

10. Develop a special Pascal function called `Clock`. This function is to return a longint representing the time of the day from midnight, measured in seconds. In defining the function `Clock`, use the procedure **`GetTime`** to return the current date. Then compute the total seconds after midnight by adding the total number of seconds from fields in data type **`DateTimeRec`**.

11. Test the algorithms `Straight_Insertion_Sort`, `Shellsort`, and `Quicksort` by having a program sort 100 random integers. In each case use the function `Clock` from Exercise 10 to compare the sorting times of these three algorithms. Before beginning a sort, measure the clock time, and repeat this step when execution of a sort has been completed. The sorting time is simply the difference between these two times.

12. Repeat Exercise 11 for an array containing 1000 random integers. *Note* : If you are interested in checking to see if the sorting times are proportional to N^2 for `Straight_Insertion_Sort`, $N^{1.5}$ for the Shellsort, and less than $N^{1.5}$ for `Quicksort`, repeat Exercise 11 for an array containing 10,000 random integers. After collecting these data, use a sheet of logarithmic paper with the horizontal axis representing the values of `N` and the vertical axis representing the sorting times. Check the slopes to see if they are close to 2, 1.5, or less.

13. Rewrite the binary search algorithm so that it can be executed recursively. Name this new procedure `Recursive_Binary_Search`. This procedure should have five formal parameters: `A`, `Low`, `High`, `Key`, and `Position`. The parameter `A` is a table being searched from index position `Low` to index position `High`. If the key is not located, return a value less than the value of the index `Low`. Assume that the indices of `A` need not be from 1 to `N` but in general can be in the range `M` to `N`.

14. A company has employee records stored in the memory of the computer. Each record contains the following information: employee name, address of residence, residence phone number, birth date, and annual salary. Write a program using several procedures for entering employee records (maximum 10 records) from the keyboard, then sorting these records with several tables: an array containing records sorted alphabetically by name, an array containing records sorted by age, and an array containing records sorted by annual salary. How could the binary search algorithm be used to search for all employees having either a particular age or a particular salary? *Hint* : Be concerned with how you structure the storage of an employee's birth date, because your program must be able to compute an employee's age as a whole number from this data.

15. Write a program that counts the total individual vowels as well as the total number of alphabetical letters entered from the keyboard. Assume that one or more lines of text can be typed from the keyboard. After the input has been completed, have this program plot a bar chart giving the percentage of vowels found in the input text string, as shown in Figure 9.17.

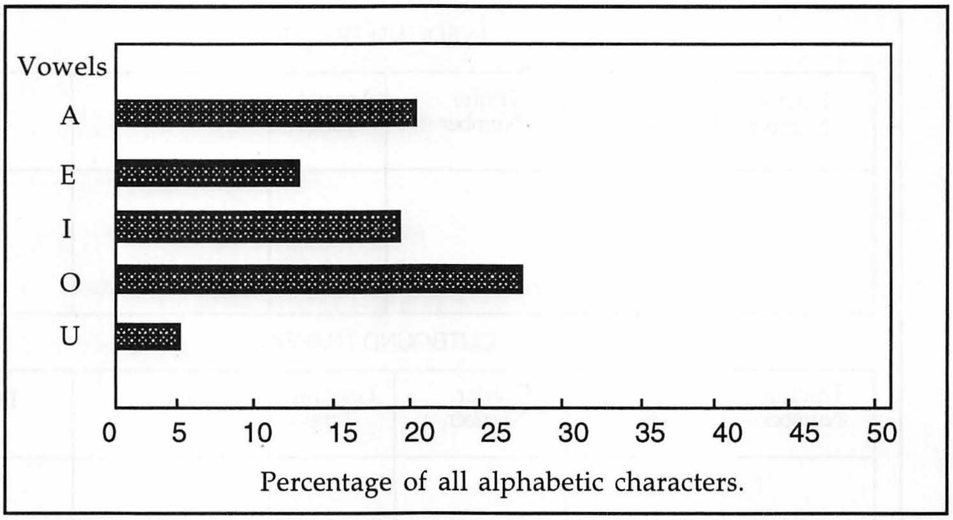


Figure 9.17

You may want to consider how to scale the horizontal axis so that the individual bars can be better displayed in the Drawing window:

16. The ABC Trucking Company keeps its dispatch record for the day on a sheet like the one shown in Figure 9.18. Write a small system having two record arrays, `Inbound` and `Outbound`. This program must be able to use a menu so that the dispatcher can choose from one of the following options:

    (a) Enter incoming truck into INBOUND TRAFFIC.
    (b) Enter outgoing truck into OUTBOUND TRAFFIC.
    (c) Search for driver in INBOUND TRAFFIC.
    (d) Search for driver in OUTBOUND TRAFFIC.
    (e) Display INBOUND TRAFFIC.
    (f) Display OUTBOUND TRAFFIC.
    (g) Display trailer information for INBOUND and OUTBOUND TRAFFIC.
    (h) Exit program.

    Trailer information must be displayed in alphabetical order with trailer number and status (whether the trailer remains inbound or outbound). If inbound, report the city where the trailer's load originated; if outbound, report the city of destination.

INBOUND TRAFFIC

Tractor Number	Trailer Name	Trailer Number	Arrival Time	From	Driver

OUTBOUND TRAFFIC

Tractor Number	Trailer Name	Trailer Number	Leaving Time	To	Driver

FIGURE 9.18

17. A complex number is represented by the notation $a + j\,b$. In this notation the number a represents the real part, and the number b represents the imaginary part. The special constant j represents the square root of -1. In Pascal we can represent a complex number by the following record format:

```
Complex  =  record
               Real_Part : real;
               Imaginary_Part : real
            end;
```

Consider the following rules for complex algebra:

$$(a+jb)+(c+jd)=(a+c)+j(b+d)$$

$$(a+jb)-(c+jd)=(a-c)+j(b-d)$$

$$(a+jb)(c+jd)=(ac-bd)+j(ad+bc)$$

The conjugate of $a+jb=a-jb$

The modulus $a+jb=(a^2+b^2)^{1/2}$

$$\frac{a+jb}{c+jd} = \frac{(a+jb) * \text{conjugate of } c+jd}{(\text{modulus of } c+jd)^2}$$

Write the following set of procedures for performing complex algebra:

(a) **procedure** `Add_Complex(`**var** `X: Complex; Y, Z: Complex);` This procedure will add the complex number `Y` to the complex number `Z` and return the the result through `X`.
(b) **procedure** `Sub_Complex(`**var** `X: Complex; Y, Z: Complex);` This procedure will subtract the complex number `Z` from the complex number `Y` and return the the result through `X`.
(c) **procedure** `Mult_Complex(`**var** `X: Complex; Y, Z: Complex);` This procedure will multiply the complex number `Y` by the complex number a and return the the result through `X`.
(d) **procedure** `Div_Complex(`**var** `X: Complex; Y, Z: Complex) ;` This procedure will divide the complex number `Y` by the complex number `Z` and return the the result through `X`.
(e) **procedure** `Conjugate(`**var** `X: Complex, Y: Complex);` This procedure will take the conjugate of the complex number `Y` and return the the result through `X`.
(f) **procedure** `Modulus(`**var** `R:real; Y : Complex);` This procedure will take the modulus of the complex number `Y` and return the the result through `R`.
(g) **procedure** `Write_Complex(X : Complex );` This procedure will display to the Text window a complex number, given by `X`, in the form `a + j b`.
(h) **procedure** `Read_Complex(`**var** `X : Complex) ;` This procedure will read a complex number typed at the keyboard in the form $a\ +j\ b$ and assign this to `X`.

Test your procedures by using several sets of complex numbers.

18. A cubic equation is given by the following identity:

$$x^3 + a_2x^2 + a_1x + a_0 = 0$$

Let $Q = \frac{a_1}{3} - \frac{a_2{}^2}{9}$, and $R = \frac{(a_1a_2 - 3a_0)}{6} - \frac{a_2{}^3}{27}$

If $Q^3 + R^2 > 0$, the cubic equation has one real root and two complex conjugate roots.

If $Q^3 + R^2 = 0$, all three roots are real, and at least two are equal.

If $Q^3 + R^2 < 0$, all roots are real.

By letting $S_1 = [R + (Q^3 + R^2)^{1/2}]^{1/3}$

$S_2 = [R - (Q^3 + R^2)^{1/2}]^{1/3}$

the roots for the cubic equation are defined as follows:

$$X_1 = (S_1 + S_2) - \frac{a_2}{3}$$

$$X_2 = \frac{(S_1 + S_2)}{2} - \frac{a_2}{3} + \frac{j\sqrt{3}}{2}(S_1 + S_2)$$

$$X_3 = \frac{(S_1 + S_2)}{2} - \frac{a_2}{3} - \frac{j\sqrt{3}}{2}(S_1 + S_2)$$

Using the procedures from Exercise 17, and the function from Exercise 13 of Chapter 7, write a new procedure called `Cube_Roots` for computing the roots of a cubic equation when $Q^3 + R^2 > 0$, and when $Q^3 + R^2 = 0$, given the values for the coefficients a_2, a_1, and a_0.

The third case, where $Q^3 + R^2 < 0$, requires more thought and analysis. If you want an added challenge, try to determine how procedure `Cube_Roots` can be extended for computing the roots when $Q^3 + R^2 < 0$.

19. For a better understanding of the basics of character manipulation, assume that all strings will be represented by a **packed array**`[1..M]` **of** `char`. Develop algorithms for the following set of procedures and functions:

`N <-- Length_S( X )` This function measures the length of string `X`. A string `X` is empty if the first character in the string is a null character. The length of `X` is found by counting characters up to the first null character.

`Z <-- Concat_S( X, Y )` Concatenate a copy of string `Y` to the right of a copy of string `X`. The sum of lengths `X` and `Y` cannot exceed the maximum length allowed for `Z`.

`Z <-- Substr( X, I, J )` Copy the `Ith` through the `Jth` characters of `X` to `Z`. If `J = 0 , I < 0`, or `J <= I` string `Z` remains unchanged.

`P <-- Index_S( X, Y )` Search string `X` with pattern `Y` for the first occurrence of string `Y`, setting `P` to this position; otherwise, `P` is set to zero.

`Z <-- Insert_S( X, Y, S )` For pattern `Y` in `X`, replace the first occurrence of `Y` with `S`. If the pattern does not exist, leave `Z` as a null string. What happens if pattern `Y` is a null string? What if `X` is a null string? `Delete_S( X, I, N )` Starting at position `I` in string `X`, delete `N` characters.

When convenient, use one or more of these functions or procedures to define one of the other functions. After testing your functions and procedures, convert your algorithms into Macintosh or THINK Pascal programs and show that they are functional.

Chapter 10

Files

OBJECTIVES

After completing Chapter 10, you will know the following:

1. The basic elements of Pascal files: the file pointer, the file buffer, and file components.
2. The differences between nontext and text files.
3. The differences between sequential-access and random-access files.
4. Commands for opening and closing files: `open`, `close`, `rewrite`, and `reset`.
5. Commands for reading and writing to Pascal files: `read`, `readln`, `get`, `write`, `writeln`, and `put`.
6. The miscellaneous commands `seek` and `filepos`.
7. Steps for using a printer as a file device for printing text files.

10.1 ADVANTAGES OF USING FILES

In Chapter 9 we saw how data can be structured and made available during the execution of a program. But how can the information in a structure such as an array be kept when the program ends execution? An approach that requires the user to enter information from the keyboard each time the program is executed is not convenient, nor does it make full use of the computer's capabilities. This becomes more evident when the array has hundreds of elements.

Files can provide a useful medium for output during the execution of a program by allowing you to save the values associated with simple and structured variables. When files are employed, information associated with variables is not lost when the program terminates execution. Upon reexecution of the program, files can serve as an input medium for entering the values associated with variables. If RAM (main memory) becomes limited due to the storage of structures involving a large number of elements, files provide temporary storage while the program is in execution. When using an array in Pascal, you must explicitly declare what the maximum size of the array will be when the program is executed. RAM storage can thus be wasted if not all of the array is required. It also means that an execution error will occur if we attempt to store data beyond the bounds of the array. Using a file to store the data rather than an array relieves us of such problems. Files serve as an efficient and compact form for storage, both during the execution of a program and after execution has terminated.

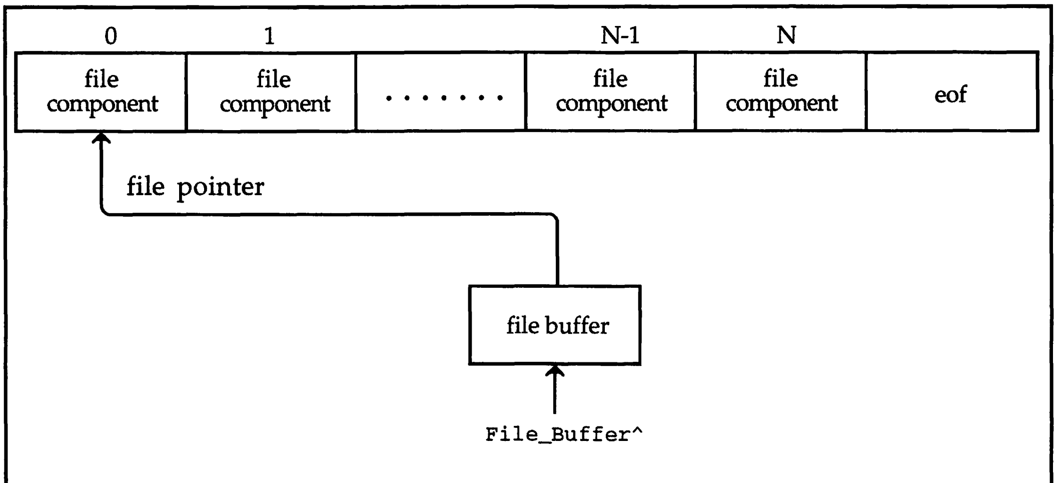
In this chapter we introduce the concepts of sequential and random-access files and explain how they relate to Macintosh/THINK Pascal files. We first discuss sequential files, opening and closing such files, reading from and writing to such files, and the steps needed to test for an end-of-file marker. In the context of sequential files, a special file type called *text* is examined and compared with the concept of a file of characters. In addition, we review the concept of random-access files, opening and closing such files, reading from and writing to such files, and the steps needed to test for an end-of-file marker. The command `seek` is discussed as an important mechanism for accessing a component of a random-access file.

Finally, we should stress that there is little practical difference between Macintosh Pascal and THINK Pascal when it comes to the handling of files. All of the programs in this chapter will execute under either of these versions of Pascal.

10.2 BASIC CONCEPT OF A PASCAL FILE

In Chapter 1 we saw how to create a simple Macintosh Pascal program and then save it as a program file on a diskette or hard disk. It is also possible, when executing a Pascal program, to use commands in the source code of the program for creating, writing to, or reading from one or more data files. Rather than storing programs, these files store data such as integers, reals, characters, strings, arrays, sets, or records. How can a file be understood by an individual who knows little about the organization and architecture of a Macintosh computer? A simple model of a file can be represented as an organized collection of related data objects stored on an accessible storage medium such as a diskette. Each related data object is called a *file component*. In relation to Pascal, a file is similar to an array of records, with the array subscript having a range from 0 through N. Figure 10.1 illustrates this concept. A file is different from an array, because it can increase or decrease in size during the execution of a program. Arrays in Pascal are always a fixed size determined at the time the program is translated.

As Figure 10.1 indicates, each file component is associated with a unique index number, the first file component having an index value of zero. Each file is terminated with a special file component called an end-of-file marker (`eof`). When a file is first created, it contains only the end-of-file marker and no file components. Pascal files can be divided into two broad categories: text files and nontext files (machine-language files). When a file is used to store an `ordinal` type, a `real` type, an `array`, a `packed array`, a `set`, a `string` type, or a `record` type, each file component will have a size equivalent to the data type that is being stored. In addition, each file component will store the data type in its machine format. For example, a file storing real numbers has each file component fixed at 4 bytes, because a real number in Macintosh Pascal requires only 4 bytes for representing its value at the machine level. File components for any nontext files are always of a fixed size relative to the data type that is being stored. Text files are different, because they can store character data of varying string lengths, that is, lines of text. Text files are always written to by executing the command `writeln` and are always read from by executing the command `readln`. File components of text files can vary in length simply because lines of text produced from executing a `writeln` command can vary in length. Section 10.8 provides further discussion on the use of text files in Pascal.



The file buffer represents a physical link between the Pascal file represented by the pointer `File_Buffer` and the physical file represented by file components.

Figure 10.1 A simple model of a file.

During the execution of a Pascal program, a file component is accessible through an object hidden from the programmer, called a *file pointer*. When a nonempty file is opened, the file pointer is initially positioned so that it points to file component zero, that is, the beginning of the file. During the execution of a Pascal program that is reading from a file, the file pointer will always point to the next file component to be read. While executing a Pascal program that is writing to a file, the file pointer will always be positioned where the next file component is to be written. A file pointer can be repositioned in Macintosh Pascal by executing one of three commands: `reset`, `rewrite`, or `seek`.

In Pascal a nontext file is declared by using the syntax **`file of`** `data type`. For example, the following code declares an identifier called `Table` to be a **`file of`** `integers`:

```
var
   Table : file of integer;
```

Rules for naming a Pascal file are the same as those for any other Pascal identifier. In our example, `Table` represents a logical file in the Pascal program containing this declaration. This declaration does not create the actual or physical file for storage on the diskette. Rather, it only implies that in the logic of the Pascal program, a file by the name `Table` is to be employed as a Pascal variable. The actual physical file is created and linked to the logical file through the execution of one of three commands: `open`, reset, or rewrite. The name of the physical file can be seen when we examine the desktop and folders within the desktop. A special object called the *file buffer* links the contents of the physical file with those of the logical file within a Pascal program. As Figure 10.1 shows, the file buffer serves as an intermediate storage area between the physical file, represented by the collection of file components, and the logical file, represented by the Pascal file variable. The file buffer is accessed by using a pointer variable having the format Pascal_File_Name^ (the concept of a pointer as a data type is discussed in Chapter 12). For example, the Pascal file Table would have a file buffer represented by Table^. The file buffer, along with the file pointer, is assumed to be pointing to the next file component. This link remains in effect either until the program terminates execution or until a command is executed closing the file. Execution of the command close breaks this link. Information from a file component is accessed through the execution of an assignment statement involving either the file buffer or through input commands such as read or readln. Recording information to a file component is done either by an assignment statement involving the file buffer, or by output commands such as write or writeln.

Macintosh and THINK Pascal support two general types of files: sequential and random. A sequential file is similar in concept to recording information on a cassette tape. For example, consider a cassette tape that contains a recording of a debate between two candidates for elective office. If we want to listen to the later part of the debate, we have to start at the beginning and wind the tape until we reach the desired portion. In Pascal a sequential file represents a data object storing information in a linear order. That is, initial information is written at the beginning of the file and later information is written toward the end of the file. If we need to access an item stored in a file component of a sequential file, we must begin reading the first file component and continue by reading succeeding file components until either the item is found or an end-of-file marker is encountered.

A random file represents a data object with each file component represented by a component number. This number can be specified as an actual argument to a procedure called seek. Thus, with a random file the file buffer can be directed to point to a file component without following a linear sequence. Once the file pointer has been positioned, we are able to read from or write to the file component. This is the major advantage of a random-access file.

10.3 ACCESSING SEQUENTIAL FILES

Before we can read from or write to a sequential file, we must open it by using one of three commands: open, rewrite, or reset. We first consider the command open.

The command open opens a file having a name and links the Pascal file with the physical file. The format for using this command is

```
open ( Pascal_File, Physical_File );
```

Pascal_File represents an identifier declared as a Pascal file type, and Physical_File is a string type representing the name of the physical file. In both Macintosh and THINK Pascal, names of physical files are limited to a maximum of 31 characters and can use any of the characters on the keyboard except for the colon (:).[1] If no

1 An attempt to use a file name longer than 31 characters causes the error message "Macintosh System Error –37: Bad name" in THINK Pascal. The message raised in Macintosh Pascal is "The File Manager has detected an invalid file or volume name (possibly a zero-length

file exists with the name given by `Physical_File`, an empty file is created and then opened. If a physical file by that name already exists, that file is opened. Whenever a file is opened, the file buffer points to the first file component.

It is important that a file be closed prior to ending program execution. The command `close` breaks the connection between a Pascal file and its corresponding physical file. The format for calling this procedure is

```
close( Pascal_File );
```

Once this command is executed, any further references to the physical file linked to the Pascal file are invalid unless the physical file is again opened. It is a good programming practice to keep a file open only as long as it is needed. For this reason use of the command `close` is encouraged. However, all files are automatically closed when a Macintosh or THINK Pascal program terminates normal execution. There is no guarantee that files will be closed by any abnormal program termination.

Before writing to a sequential file, it is necessary that the file pointer be positioned to point to the beginning of the file. In this regard the command `rewrite` can be useful. The syntax for the `rewrite` command is

```
rewrite( Pascal_File, Physical_File );
```

`Rewrite` creates an empty file with the name given by `Physical_File`, links `Pascal_File` with `Physical_File`, and positions the file buffer at the beginning of the file. If the physical file already exists when this command is executed, it is deleted and an empty file having the same name is created. If the rewrite command is executed and no physical file name is associated with the Pascal file, an empty unnamed file is created for use during the lifetime that the Pascal file is in existence. Understand that `close` command cannot be executed on a Pascal file that is linked with an unnamed file.

When the command

```
rewrite( Pascal_File );
```

is executed, it positions the file buffer at the beginning of the file and any file components previously associated with the file are lost. Notice that the two commands

```
open( Pascal_File, Physical_File );
rewrite( Pascal_File );
```

and the command

```
rewrite( Pascal_File, Physical_File );
```

are equivalent in their effects.

Writing to a file requires execution of either the command `write` or the command `put`. The command `write` will copy a value of an expression to a file given by the Pascal file name. The format for using this command is

```
write( Pascal_File, expression  )
```

name)". An attempt to use a physical file name that includes a colon causes an error message saying that no such disk or volume can be found.

The first argument is the Pascal file name, and the second is an expression. Here the data type of the expression must be type-compatible with the component type of the Pascal file. When executed, the value of the expression is written to the current file component, and the file pointer is advanced to the next file component, where it is ready to write. Although this appears to be the same `write` statement discussed in Chapter 3, there is a difference. With `Pascal_File` replaced with the standard output file name `Output`, the command `write` displays the value of the expression as characters to the standard output file, the Text window. Where `Pascal_File` is the file variable, data is written to an external data file but in machine format.

For example, consider a program titled `Storing_Random_Numbers`. The program assigns 10 random numbers to an array called `A`. After completing this step, a new file is opened by executing the command `open`, and the file pointer is positioned at the beginning of the file by executing the command `rewrite`. Each real number in array `A` is then copied and written (stored) in a physical file called `Test File`, not as characters to the screen but in a machine form represented by binary data.

```
program Storing_Random_Numbers(input, output);
{ Purpose:  This is a simple program for testing the commands }
{           open, rewrite, and close. }
   type
      Number_File = file of real;
   var
      Data_Block : Number_File;
      Counter : integer;
      A : array[1..10] of real;
begin
{ Generate 10 random numbers for array A. }
   for Counter := 1 to 10 do
      A[Counter] := random;
{ Open the test file and position the file pointer at the front }
{ of the file. }
   open(Data_Block, 'Test File');
   rewrite(Data_Block);
{ Write 10 real numbers to the test file. }
   for Counter := 1 to 10 do
      write(Data_Block, A[Counter]);
{ Close the test file. }
   close(Data_Block);
end.
```

An alternative to the command `write` is the command `put`. When using this command, the Pascal file buffer must be assigned the value of the expression before execution of `put`. To compare the commands `write` and `put`, note that

```
write( Pascal_File, expression );
```

is equivalent to

```
Pascal_File^  :=  expression;
```

```
put( Pascal_File );
```

Notice that the file buffer represented by Pascal_File^ is assigned the value of expression. The put command copies the contents of the file buffer and writes this data to the current file component, and the file pointer advances to where the next file component will be located. In our program Storing_Random_Numbers, the **for** loop using a write command to store 10 numbers can be replaced with the following code:

```
 for Counter := 1 to 10 do
      begin
         Data_Block^ := A[Counter];
         put(Data_Block)
     end;
```

An additional example employing a sequential file is the following program, titled Saving_Name_Age. This program allows four records, each containing a name and age, to be entered from the keyboard and written to a physical file called Name & Age.

```
program Saving_Name_Age(input, output);
{ Purpose:  This program enters four names and ages from the }
{           keyboard and stores this data in four records of a }
{           physical file. }
   uses
      QuickDraw1;
   type
      Person = record
                  Name : string[30];
                  Age : integer
               end;
      Output_File = file of Person;
   var
      Pascal_File : Output_File;
{ ******************************************************** }
   procedure Set_Text_Window;
      var
         Window : Rect;
   begin
      HideAll;
      SetRect(Window, 0, 40, 512, 342);
      SetTextRect(Window);
      ShowText
   end;
{ ******************************************************** }
   procedure Open_File (var Pasf : Output_File);
      var
         Phf : string;
   begin
      Phf := 'Name & Age';
      rewrite(Pasf, Phf);
   end;
```

```
{ ******************************************************** }
   procedure Enter_Records (var Pasf : Output_File);
      var
         Index : integer;
         Individual : Person;
   begin
      writeln;
      for Index := 1 to 4 do
         begin
            write(' Enter person`s name : ');
            readln(Individual.Name);
            write(' Enter person`s age: ');
            readln(Individual.Age);
            write(Pasf, Individual);
            writeln;
         end;
   end;
{ ******************************************************** }
begin { Body of the main program.}
{ Set Text window for viewing. }
   Set_Text_Window;
{ Open physical file for output. }
   Open_File(Pascal_File);
{ Prompt user for the number of names and the names themselves. }
   Enter_Records(Pascal_File);
{ Close the Pascal file. }
   close(Pascal_File);
end.
```

As you can see, the major steps for opening a file and entering data into the file components are performed by two separate procedures. The procedure Open_File uses the command rewrite rather than open, because we are creating a new file. The procedure Enter_Records prompts the user for name and address and, as they are read, assigns the values of Name and Age to the local variable called Individual. It is the record variable called Individual that is written to the current file component. In both cases, the formal parameter called Pasf, short for *Pascal file*, is a variable type. *Pascal requires that all formal parameters of type* ***file*** *be declared in the formal-parameter list as variable types.*

When reading from an existing file, the command reset can be executed for opening that file. There are two possible formats when using reset. The first is

```
reset( Pascal_File, Physical_File );
```

This invokes a search for the physical file having a name given by Physical_File and, when found, links the Pascal file name with the physical file. After the link has been made, the file pointer is positioned at the beginning of the file. An error occurs if no file exists with the name given by Physical_File. This command can be executed on an unnamed file only if the rewrite command has previously been executed for the purpose of establishing an anonymous file.

The second format assumes that the Pascal file has already been opened:

```
reset( Pascal_File );
```

This command positions the file pointer at the beginning of the file. If the file had been opened by `rewrite`, it now can only be read. Notice that the two commands

```
open( Pascal_File, Physical_File );
reset( Pascal_File );
```

and the command

```
reset( Pascal_File, Physical_File );
```

are equivalent in their effects.

We can read from a sequential file by executing either of the commands `read` or `get`. The `read` command copies a value from the Pascal file and assigns it to a variable. The format for this command is

```
read( Pascal_File, Variable );
```

The first argument is the Pascal file name, and the second is the name of an identifier. Other than for text files, the data type for the variable must be type-compatible with the component type of the Pascal file. When executed, the current value of the file buffer is copied and assigned to the variable, and the file pointer is advanced to the next file component. While this appears to be the same `read` statement discussed in Chapter 3, there is a difference. With `Pascal_File` replaced with the standard input file name `Input`, the `read` command enters its value as text lines from the keyboard. Where `Pascal_File` is the file variable, data is read from an external data file but in machine format.

To illustrate reading from a file, consider the following program, titled `Reporting_Random_Numbers`. This program performs the reverse actions of `Storing_Random_Numbers`, in that it opens a physical file called `Test File`, and then resets the file position so that the file pointer is pointing to the beginning of the file. It then reads 10 real numbers from the physical file and, after each number is read, displays the value of the number to the standard output file, the Text window.

```
program Reporting_Random_Numbers(input, output);
{ Purpose:  This is a simple program for testing the routines }
{           open, reset, and close. }
   type
      Number_File = file of real;
   var
      Data_Block : Number_File;
      Counter : integer;
      Number : real;
begin
{ Open test file and position the pointer at the front of file. }
   open(Data_Block, 'Test File');
   reset(Data_Block);
{ Read all 10 numbers from the test file. }
   for Counter := 1 to 10 do
      begin
   { Read the next number from the test file. }
         read(Data_Block, Number);
   { Display the number just read from the test file and display }
   { its value in the Text window. }
         write(Output, Number : 10 : 4, ' ');
      end;
   writeln;
{ Close the test file. }
   close(Data_Block);
end.
```

An alternative to the command `read` is the `get` command. A file component is read by assigning a variable the value of the file buffer, followed by execution of the command `get`. The command

```
read( Pascal_File, Variable );
```

is equivalent to

```
Variable := Pascal_File^;
get( Pascal_File );
```

Notice that the variable is assigned the value of the Pascal file buffer, because this buffer contains a copy of the current file component. The variable's type must be type-compatible with the Pascal file type. Execution of the `get` command advances the file pointer to the next file component, so that the file buffer contains a copy of the contents of what is now the current file component. In our program `Reporting_Random_Numbers`, the **for** loop for reading each random number and displaying it to the Text window can be written as follows:

```
   for Counter := 1 to 10 do
      begin
      { Read the next number from the test file. }
         Number := Data_Block^;
         get(Data_Block);
      { Display the number just read from the test file and }
      { display its value in the Text window. }
         write(Output, Number : 10 : 4, ' ');
      end;
```

The program titled `Reporting_Name_Age` shows the reverse process of the program `Saving_Name_Age`. In this program the four records stored in the physical file called `Name & Age` are read and displayed to the Text window. The procedure `Read_File_Records` reads each record from the physical file.

```
program Reporting_Name_Age(input, output);
{ Purpose:  This program reads four records from a physical file }
{           and displays the contents of these records to the }
{           Text window. }
{ Insert the following uses clause for Macintosh Pascal. }
{  uses              }
{     QuickDraw1;    }
   type
      Person = record
                  Name  : string[30] ;
                  Age : integer
               end;
      Input_File = file of Person;
  var
      Pascal_File : Input_File;
{ ******************************************************** }
  procedure Set_Text_Window;
      var
         Window : Rect;
  begin
      HideAll;
      SetRect(Window, 0, 40, 512, 342);
      SetTextRect(Window);
      ShowText
  end;
{ ********************************************************  }
  procedure Open_File (var Pasf : Input_File);
      var
      Phf  : string;
  begin
      Phf := 'Name & Age';
      reset(Pasf, Phf);
  end;
{ ******************************************************** }
  procedure Read_File_Records (var Pasf : Input_File);
      var
         Index : integer;
         Individual : Person;
  begin
      writeln;
      for Index := 1 to 4 do
         begin
            read(Pasf, Individual);
            writeln(' Name : ', Individual.Name);
            writeln(' Age: ', Individual.Age);
            writeln;
         end;
  end;
{ ******************************************************** }
begin { Body of the main program. }
{ Set Text window for viewing. }
  Set_Text_Window;
{ Open physical file for output. }
  Open_File(Pascal_File);
{ Prompt user for the number of names and the names themselves. }
  Read_File_Records(Pascal_File);
```

```
{ Close the Pascal file. }
   close(Pascal_File);
end.
```

What about using the commands `writeln` and `readln`? These commands pertain only to special file types called *text* files. Commands such as `read`, `write`, `get`, and `put` deal with information stored in a machine-language format. *Text* files store all information in a character format and are discussed in Section 10.9.

10.4 MERGING A RECORD INTO A SEQUENTIAL FILE OF RECORDS

Suppose that we have a sequential file of `N` records, with each record containing two fields: full name and telephone number. We will assume that the records are stored alphabetically by name. Our purpose is to develop an algorithm that prompts the user for a new record and, after the required information has been entered from the keyboard, merges this new record into the sequential file of names while keeping the file alphabetically ordered.

The following steps might be considered as an initial solution.

1. Open the sequential file by resetting the file pointer.
2. Read `N` records from the sequential file into an array capable of storing `N + 1` records.
3. Add a new record to the `N + 1` position of the array.
4. Sort the `N + 1` records in the array.
5. Reset the file pointer for writing to the sequential file.
6. Write `N + 1` records to the sequential file.
7. Close the sequential file.

The solution seems simple, but what is the maximum size for the internal array? There is no definite answer to this question, unless we fix the maximum number of records for the sequential file, that is, unless we establish an upper limit to the number of array elements exceeding or equal to the value of `N + 1`. Setting such a limit makes it impossible to add more records if the limit is exceeded, unless we return to the Pascal program and make an explicit change in the bounds of the array declaration. An alternative is to make the bounds large enough so that our internal array will always be able to handle the desired number of records. But the user may still need to add one more record, exceeding the bounds of the array. It is also possible that by having such a large internal array, the program may exceed available free memory during translation and not be able to be executed.

Obviously, we should abandon using an internal array and consider an alternative method of merging a single record into the sequential file of records. One such approach is to read each record from the sequential file, compare its full-name field with that of the new record, and, if ordered less alphabetically, write the old record to a temporary file. Once an old record is found where its full-name field is greater alphabetically than that of the new record, the new record is written to the temporary file followed by writing the old record. All remaining records of the sequential file are then read and written to the temporary file. Once this has been completed, the temporary file now contains the `N + 1` records in alphabetical order. By resetting the file pointers of both files (`reset` for the temporary file, `rewrite` for the sequential file), each record in the temporary file is read and then written to the sequential file.

Let us refine our abstraction, using the following steps:

```
procedure Merge_Sort;

{  Purpose: This module will require three Pascal files:  one
called Directory, a second called Temporary,  and a third called
Total_Count.  The Pascal file called Directory will be linked with
the physical file called Telephone Directory.  The Pascal file
called Temporary will only exist for the length of execution of
the module Merge_Sort. Total_Count will be linked with a physical
file called Total Number of Records.  This file contains only one
file component,  the value of the number of records in file
Directory.  Two local variables called New_Record and Old_Record
will be used for keeping a single record of information. }

begin
{ Prompt user for a new record. }
   Prompt_New_Record( New_Record );
{ Open three files for accessing information. }
   Open_Files( Directory, Temporary, Total_Count );
{ Assign new record to the file Temporary. }
   read( Total_Count, Number_Records );
{ Check if Directory is an empty file. }
   if Number_Records = 0 then
   { Write a new record to Directory and update the file }
   { Total_Count. }
      begin
         rewrite( Directory );
         write( Directory, New_Record );
         rewrite( Total_Count );
         write( Total_Count, Number_Records + 1 )
      end
   else
      begin
         Counter <-- 1;
         read( Directory, Old_Record );
         while  ( Old_Record.Full_Name <  New_Record.Full_Name )
                 and ( Counter <= Number_Records   ) do
            begin
            { Write old record to Temporary. }
               write( Temporary, Old_Record );
            { Read next record in Directory if Counter does }
            { not exceed number of records. }
               Counter <-- Counter + 1;
               if Counter <= Number_Records then
                  read( Directory, Old_Record );
            end;
      { Now write new record to temporary file. }
         write( Temporary, New_Record );
      { Read the remaining records from Directory and write }
      { to Temporary. }
```

```
         while ( Counter <= Number_Records ) do
            begin
               write( Temporary, Old_Record );
               Counter <-- Counter + 1;
               if Counter <= Number_Records then
                  read( Directory, Old_Record );
            end;
      { Update file Directory and Total_Count. }
         rewrite( Total_Count );
         write( Total_Count, Number_Records + 1 );
         rewrite( Directory );
         reset( Temporary );
         for Counter <-- 1 to Number_Records + 1 do
            begin
               read( Temporary, Old_Record );
               write( Directory, Old_Record )
            end
      end;
{ Close the nontemporary Pascal file. }
   Close_Files( Directory )
end; { Merge_Sort }
```

Following is a complete listing of a THINK Pascal program called `Merging_Record`. The program includes the procedure `Merge_Sort` along with three internal files: `Prompt_New_Record`, `Open_Files`, and `Close_Files`. Notice that several internal files are used in both the procedures `Merge_Sort` and `Display_Records`.

```
program Merging_Record(input, output);
{ Purpose:  This program demonstrates how a record can be merged }
{           into a file of records. }
{ ******************************************************** }
   procedure Set_Text_Window;
      var
         Window : Rect;
    begin
      HideAll;
      SetRect(Window, 0, 40, 512, 342);
      SetTextRect(Window);
      ShowText
    end;
{ ******************************************************** }
   procedure Merge_Sort;
      type
         Listing = record
                       Full_Name : string[30] ;
                       Phone_Number : string[12]
                     end;
         Phone_Directory = file of Listing;
         Record_Count = file of integer;
      var
```

```
        Directory, Temporary : Phone_Directory;
        New_Record, Old_Record : Listing;
        Total_Count : Record_Count;
        Counter, Number_Records : integer;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
  procedure Prompt_New_Record (var Item : Listing);
     begin
      write(' Enter person`s full name: ');
      readln(Item.Full_Name);
      write(' Enter telephone number: ');
      readln(Item.Phone_Number);
    end;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
  procedure Open_Files (var F1, F2 : Phone_Directory;
                                  var F3 : Record_Count);
    begin
       open(F1), 'Telephone Directory');
       reset(F1);
       rewrite(F2);
       reset(F3, 'Total Number of Records')
    end;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
  procedure Close_Files (var F1 : Phone_Directory;
                                      var F2 : Record_Count);
    begin
       close(F1);
       close(F2)
    end;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
    begin { Body of procedure Merge_Sort. }
    { Prompt user for a new record. }
       Prompt_New_Record(New_Record);
    { Open three files for accessing information. }
       Open_Files(Directory, Temporary, Total_Count);
    { Assign new record to the file Temporary. }
       read(Total_Count, Number_Records);
    { Check if Directory is an empty file. }
       if Number_Records = 0 then
       { Write new record to Directory and update the file }
       { Total_Count. }
          begin
             rewrite(Directory);
             write(Directory, New_Record);
             rewrite(Total_Count);
             write(Total_Count, Number_Records + 1)
          end
       else
          begin
             Counter := 1;
             read(Directory, Old_Record);
             while(Old_Record.Full_Name < New_Record.Full_Name)
```

```
                  and (Counter <= Number_Records) do
               begin
               { Write old record to Temporary. }
                  write(Temporary, Old_Record);
               { Read next record in Directory if Counter }
               { does not exceed number of records. }
                  Counter := Counter + 1;
                  if Counter <= Number_Records then
                     read(Directory, Old_Record);
               end;
         { Now write new record to temporary file. }
            write(Temporary, New_Record);
         { Read the remaining records from Directory and }
         { write  to Temporary.}
            while (Counter <= Number_Records) do
               begin
                  write(Temporary, Old_Record);
                  Counter := Counter + 1;
                  if Counter <= Number_Records then
                     read(Directory, Old_Record);
               end;
         { Update the files Directory and Total_Count. }
            rewrite(Total_Count);
            write(Total_Count, Number_Records + 1);
            rewrite(Directory);
            reset(Temporary);
            for Counter := 1 to Number_Records + 1 do
               begin
                  read(Temporary, Old_Record);
                  write(Directory, Old_Record)
               end
         end;
     { Close the nontemporary Pascal files. }
        Close_Files(Directory, Total_Count)
  end;
{ ******************************************************* }
  procedure Display_Records;
     type
        Listing =   record
                       Full_Name : string[30];
                       Phone_Number : string[12]
                    end;
        Phone_Directory = file of Listing;
        Record_Count = file of integer;
     var
        Directory : Phone_Directory;
        Old_Record : Listing;
        Total_Count : Record_Count;
        Counter, Number_Records : integer;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
  procedure Open_Files (var F1 : Phone_Directory;
                                     var F2 : Record_Count);
     begin
        reset(F1, 'Telephone Directory');
        reset(F2, 'Total Number of Records')
     end;
{ - - - - - - - - - - - - - - - - - - - - -  - - - - - - - - - - - }
  begin { Body of procedure Display_Records. }
     Open_Files(Directory, Total_Count);
     read(Total_Count, Number_Records);
     writeln;
     for Counter := 1 to Number_Records do
        begin
           read(Directory, Old_Record);
           write(Old_Record.Full_Name, ' ',
                       Old_Record.Phone_Number, '/')
        end;
     writeln;
     close(Directory);
     close(Total_Count);
  end;
{ ********************************************************* }
begin { Body of the main program. }
{ Set Text window for viewing. }
  Set_Text_Window;
{ Merge a record with a  file. }
  Merge_Sort;
{ Display all records of the sequential file. }
  Display_Records;
end.
```

As you can see, `Temporary` serves as a temporary file only for the length of time that the module is in execution. Whereas the procedure `Open_Files` opens all three files, the procedure `Close_Files` closes only the two Pascal files `Directory` and `Total_Count`. Why? Remember that the purpose of the command `close` is to close a file by breaking the link of a Pascal file with its corresponding physical file. The Pascal file `Temporary` has no true association with any physical file and is automatically closed when the procedure `Merge_Sort` terminates execution. Attempting to close `Temporary` by executing the command `close(Temporary)` results in an execution error.

Although the algorithm for inserting a new record seems practical, it is not an efficient algorithm in the worst case. For example, what if a file contains n records and in the worst case the record to be inserted is appended at the end of the file. This requires n records to be read from the present data file and written to the temporary file. After the new record is written to the temporary, $n + 1$ records from the temporary file are now written to the Directory. This requires an order of $2n + 1$ records to be read, not always an efficient algorithm. Section 10.8 discusses a natural merge algorithm for sorting a random access file where execution is in the order of $n\ [\ \log_2 n\ ]$.

10.5 ACCESSING RANDOM FILES

As stated earlier, a random-access file in Macintosh Pascal is distinguished by the ability to reference a file component directly without the necessity of performing a linear search sequence from the beginning of the file. Although the model given in Figure 10.1 serves as a representation for a random-access file, only the command open can open a random-access file for direct access. The commands `reset` and `rewrite` are special to Macintosh and THINK Pascal, because they can only be applied to sequential files. In addition, we can use the commands `read` and `get` to read information from a file component, and the commands `write` and `put` to write information to a file component.

Two additional routines exist for accessing file components. The first, called `seek`, allows access to a file component by referencing the component number. The format for calling this command is

```
seek( Pascal_File, Component_Number );
```

To access any Pascal file with this command, we must first open the file by executing the command open. The actual parameter `Component_Number` must be an `integer` type expression and is used to specify the file component. When `seek` is executed, the file pointer is directed to point to the file component given by the value of `Component_Number`. This file component is now the current component, so the file buffer contains a copy of the contents of the current component. It is possible to write to a file component by executing `seek` followed by an assignment operation involving a file buffer and the command `put`, or by executing the command `write`. In the program `Seeking_File_Component`, the command `seek` is used to find a file component in a file of 10 integers, replacing each file component with a random number.

```
program Seeking_File_Component(input, output);
   type
      Test_File = file of integer;
      Table = array[1..10] of integer;
   var
      Data_Block : Test_File;
      Index, Number : integer;
      Component_Number : integer;
      A : Table;
{ ********************************************************** }
   procedure Set_Text_Window;
      var
         Window : Rect;
   begin
      HideAll;
      SetRect(Window, 0, 40, 512, 342);
      SetTextRect(Window);
      ShowText
   end;
{ ********************************************************** }
   procedure Display_File (var F : Test_File);
      var
         Index, Number : integer;
   begin
      for Index := 0 to 9 do
         begin
            seek(F, Index);
            read(F, Number );
            write(Number : 6, ' ');
         end;
      writeln;
   end;
{ ********************************************************** }
   procedure Select_Components (var A : Table);
   begin
      A[1] := 1;
      A[2] := 6;
      A[3] := 4;
      A[4] := 0;
      A[5] := 7;
      A[6] := 3;
      A[7] := 8;
      A[8] := 2;
      A[9] := 5;
      A[10] := 9
   end;
{ ********************************************************** }
begin { Body of the main program. }
{ Set Text window for viewing. }
   Set_Text_Window;
{ Open the Pascal file Data_Block and link with a physical file. }
   open(Data_Block, 'File of Integer Numbers');
{ Write 10 numbers to the Pascal file Data_Block. }
   for Index := 1 to 10 do
      write(Data_Block, Index);
{ Reset the file pointer and display the values of the file }
{ components. }
   Display_File(Data_Block);
{ Select a set of 10 file components. }
   Select_Components(A);
{ Replace the file components with random values. }
   for Index := 1 to 10 do
      begin
         Component_Number := A[Index];
         seek(Data_Block, Component_Number);
         write(Data_Block, random);
         Display_File(Data_Block);
      end;
{ Close the Pascal file Data_Block.}
   close(Data_Block);
end.
```

Notice that only the command open is executed for opening and setting the file pointer to the beginning of the file. For displaying the values of the file components, the command seek is used for positioning the file pointer, then displaying the value of the file component. In the body of the main program, we use the command seek to set the file position. Either the command write or assignment of a value to the file buffer along with put is required for writing a value to the file component.

A second method for accessing a file component is to use the function filepos. This function returns the component number for the current file position. The format for calling this function is

```
filepos( Pascal_File )
```

Provided that the file is opened, filepos returns a longint value representing the value of the current file component. This function may be useful when the component number of a particular file is to be assigned to an integer variable.

Unfortunately, Macintosh Pascal does not support the ability to remove file components from within a file without deleting the complete file. If one or more file components are to be deleted or inserted, the programmer must develop a scheme for managing the allocation of file components. There are no specific commands for the deletion or insertion of file components within an established Pascal file. This is one reason why the present versions of Macintosh and THINK Pascal do not actually support the management of random files.

10.6 APPLYING THE BINARY SEARCH ALGORITHM TO FILES

Let us consider applying the seek command to searching for a record stored in a file of records. First, assume that in our file, records are ordered alphabetically by name. With this in mind, consider a Macintosh Pascal file as a linear table of file components, with the lowest file component represented by component number zero and the highest file component represented by component number N - 1. Figure 10.2 shows a simple model for this concept.

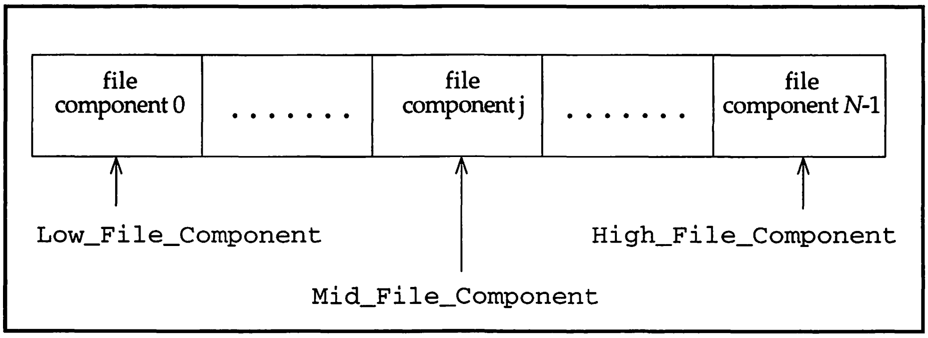


Figure 10.2 A simple model of ordered records.

In Chapter 9 we discussed the binary search algorithm as an efficient algorithm for searching a table with N elements. Our Pascal file represents a table stored externally. By storing the total number of file components in a second file, we can establish what represents the low and high indices with respect to component numbers. The middle of our table is determined by the integer result of the sum of these two indices divided by

2. Assuming that the file has been opened, the middle index is used as the component number for the center of the file. At this point the *key* is compared with a field of a record in the file buffer, and if the item is not a match, the search is continued by examining either the lower or the upper portion of the file.

We will now apply this algorithm to the telephone directory problem. We need an algorithm that can either retrieve a person's phone number or report that the party has no phone number listed. The steps for refining our solution follow:

```
procedure Binary_Search;
{ This algorithm uses two files local to this module: Directory
and Total_Counts.  File Directory contains the telephone listing,
and Total_Counts contains the total number of listings.  Local
variables include the indices Low_File_Component,
High_File_Component, Mid_File_Component, and Record_Position.
Found is used as a Boolean type to end the binary search, and
Person represents a record containing two fields:  Full_Name and
Phone_Number.  The key for searching records is the person's full
name.  Internal procedures are Open_Files, Prompt_for_Key, and
Close_Files.}

begin
{ Open the files for the telephone directory and total number of
records. }
   Open_Files( Directory, Total_Counts );
{ Read the total number of records. }
   read( Total_Count, Number_Records );
{ Enter the key for searching. }
   Prompt_for_Key( Key );
{ Initialize the low and high bounds of the index variables. }
   Low_File_Component <-- 0;
   High_File_Component <-- Number_Records - 1;
   Record_Position <--  -1;
   Found <-- false;
{ Repeat searching for the record until the key is found or no
records remain to be searched. }
   repeat
   { Compute the center of the file or subfile. }
      Mid_File_Component <-- ( Low_File_Component +
      High_File_Component  ) div 2;
   { Adjust the file pointer to center of the file or subfile. }
      seek( Directory, Mid_File_Component );
   { Assign the value of the current file component to Person. }
      Person <-- Directory^;
   { Check if this person is in our listing. }
      if Person.Full_Name = Key then
         begin
         { Record the value of file position and end searching. }
            Record_Position <-- Mid_File_Component;
            Found <-- true
         end
      else
      { Continue searching in either the lower or upper subfile. }
         if Key < Person.Full_Name then
            High_File_Component  <-- Mid_File_Component - 1
         else
            Low_File_Component  <-- Mid_File_Component + 1;
   until Found or ( Low_File_Component > High_File_Component );
{ Report the phone listing. }
   if Record_Position >= 0 then
      begin
      { Report phone number. }
         writeln( ' Person: ', Key );
         writeln( ' Phone number: ', Person.Phone_Number )
      end
   else
   { Report message of an unlisted number. }
      begin
         SysBeep(10);
         writeln(' Sorry, your party has no phone listing.')
      end;
{ Close all files. }
   Close_Files( Directory, Total_Count )
end;  { Binary_Search }
```

The Pascal program for this algorithm includes three internal procedures: Open_Files, Prompt_for_Key, and Close_Files.

```
program Search_Algorithm(input, output);
{ Purpose:  This program applies the binary search algorithm, }
{           using the procedure seek to locate a key stored }
{           within a file component. }
   uses
      QuickDraw1;
{ ****************************************************** }
   procedure Set_Text_Window;
      var
         Window : Rect;
   begin
      HideAll;
      SetRect(Window, 0, 40, 512, 342);
      SetTextRect(Window);
      ShowText
   end;
{ ****************************************************** }
   procedure Binary_Search;
      type
         Listing  =  record
                        Full_Name : string[30];
                        Phone_Number  : string[12]
                     end;
         Phone_Directory = file of Listing;
         Record_Count = file of integer;
      var
         Directory : Phone_Directory;
         Person : Listing;
         Key : string[30];
         Total_Count : Record_Count;
         Low_File_Component, High_File_Component : integer;
         Record_Position, Mid_File_Component, Number_Records :
                     integer;
         Found : Boolean;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
   procedure Open_Files (var F1 : Phone_Directory;
                                   var F2 : Record_Count);
   begin
      open(F1, 'Telephone Directory');
      reset(F2, 'Total Number of Records')
   end;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
   procedure Prompt_for_Key (var Key : string);
   begin
      write(' Enter person`s full name: ');
      readln(Key)
   end;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
   procedure Close_Files (var F1 : Phone_Directory;
                                       var F2 : Record_Count);
   begin
      close(F1);
      close(F2)
   end;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
   begin { Body of procedure Binary_Search. }
   { Open the files for the telephone directory and total number }
   { of records. }
      Open_Files(Directory, Total_Count);
   { Read the total number of records. }
      read(Total_Count, Number_Records);
   { Enter the key for searching. }
      Prompt_for_Key(Key);
   { Initialize the low and high bounds of the index variables. }
      Low_File_Component := 0;
      High_File_Component := Number_Records - 1;
      Record_Position := -1;
      Found := false;
   { Repeat searching for the record until the key is found or }
   { no records remain. }
      repeat
      { Compute the center of the file or subfile. }
         Mid_File_Component := (Low_File_Component +
                                    High_File_Component) div 2;
      { Adjust the file pointer to the center of the file or }
      { subfile. }
```

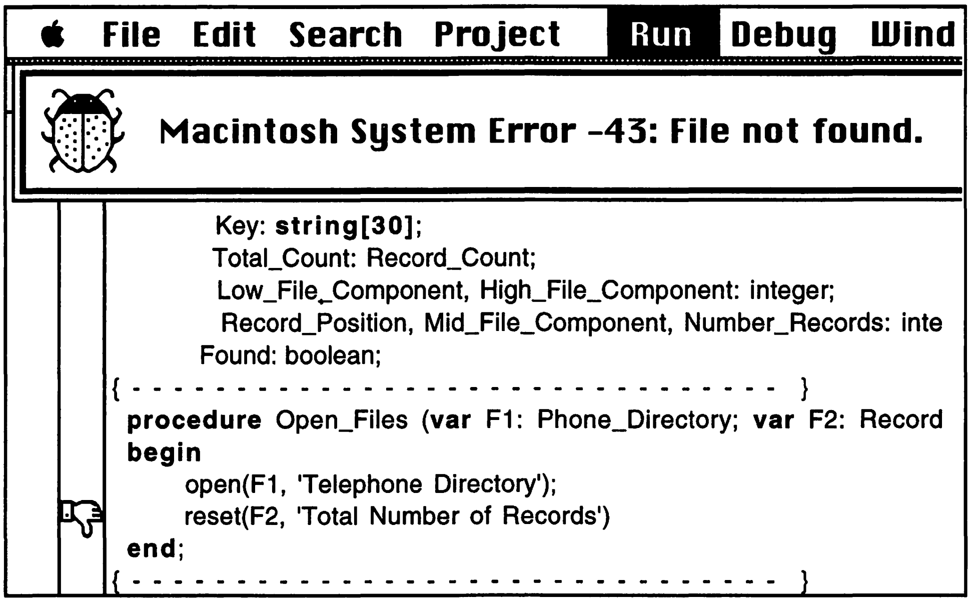
```
         seek(Directory, Mid_File_Component);
      { Assign the value of current file component to Person. }
         Person := Directory^;
      { Check if this person is in our listing. }
         if Person.Full_Name = Key then
            begin
            { Record the value of the file position and end }
            { searching. }
               Record_Position := Mid_File_Component;
               Found := true
            end
         else
         { Continue searching in either the lower or upper file }
         { or subfile. }
            if Key < Person.Full_Name then
               High_File_Component := Mid_File_Component - 1
            else
               Low_File_Component := Mid_File_Component + 1;
      until Found or (Low_File_Component > High_File_Component);
   { Report the phone listing. }
      if Record_Position >= 0 then
         begin
         { Report phone number. }
            writeln(' Person: ', Key);
            writeln(' Phone number: ', Person.Phone_Number)
         end
      else
         begin
         { Report message of unlisted number. }
            SysBeep(10);
            writeln(' Sorry, your party has no phone listing.')
         end;
   { Close all files. }
      Close_Files(Directory, Total_Count);
   end;
{ ********************************************************** }
   procedure Display_Records;
      type
         Listing  =  record
                        Full_Name : string[30];
                        Phone_Number  : string[12]
                     end;
         Phone_Directory = file of Listing;
         Record_Count = file of integer;
      var
         Directory : Phone_Directory;
         Old_Record : Listing;
         Total_Count : Record_Count;
         Counter, Number_Records : integer;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
   procedure Open_Files (var F1 : Phone_Directory;
                                        var F2 : Record_Count);
   begin
      reset(F1, 'Telephone Directory');
      reset(F2, 'Total Number of Records')
   end;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
   begin { Body of procedure Display_Records. }
      Open_Files(Directory, Total_Count);
      read(Total_Count, Number_Records);
      writeln;
      for Counter := 1 to Number_Records do
         begin
            read(Directory, Old_Record);
            write(Old_Record.Full_Name, ' ',
                     Old_Record.Phone_Number, '/')
         end;
      writeln;
      close(Directory);
      close(Total_Count);
   end;
{ ****************************************************** }
begin { Body of the main program. }
{ Set Text window for viewing. }
   Set_Text_Window;
{ Search for the record in the phone directory. }
   Binary_Search;
{ Display all records in the phone directory. }
   Display_Records;
end.
```

Notice that when using the `seek` command, we must open the file only by execution of the `open` command. If a file is opened by executing either `reset` or `rewrite`, `seek` will fail to execute if it is applied to that file.

In order to execute `Search_Algorithm` you will need a an initial file, `Total Number of Records`. An attempt to execute the program without this initial file will result in the problem shown in Figure 10.3.

The following short program establishes this file if it does not already exist.

```
program Establish_Initial_File(input, output);
{ Purpose:  This program must be executed before the }
{           program titled Merging_Record, so that an initial }
{           file can be generated storing a value of zero. }
{           This number represents the total number of records }
{           in the Telephone Directory and will be updated as }
{           records are added.. }
   var
      F3: file of integer;
begin
   rewrite(F3, 'Total Number of Records');
   write(F3, 0);
   close(F3);
end.
```



Macintosh System Error -43: File not found.

```
              Key: string[30];
              Total_Count: Record_Count;
              Low_File_Component, High_File_Component: integer;
              Record_Position, Mid_File_Component, Number_Records: inte
             Found: boolean;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
 procedure Open_Files (var F1: Phone_Directory; var F2: Record
 begin
        open(F1, 'Telephone Directory');
        reset(F2, 'Total Number of Records')
 end;
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
```

Figure 10.3 The error message resulting from a missing file.

10.7 USING THE SPECIAL FUNCTION `eof`

Pascal supports a special `Boolean` function called `eof`. When called, this function will test to see if the file pointer for a Pascal file is pointing to the end-of-file marker and, if so, will return the `Boolean` value *true*. Otherwise, the value returned is *false*. The format for using this function is `eof(Pascal_File)`. For example, consider the following steps for performing a linear search of the Pascal file `Directory`:

```
begin
{ Open the file for the telephone directory. }
   reset(Directory, 'Telephone Directory');
{ Enter the Key for searching. }
   Prompt_for_Key( Key );
{ Search for a person in the file Directory. }
   repeat
      read( Directory, Person);
   until ( Person.Full_Name = Key ) or eof( Directory );
   { Report the phone listing. }
      if ( Person.Full_Name = Key ) then
         begin
            writeln(' Person: ', Key );
            writeln(' Phone number: ', Person.Phone_Number );
         end
      else
         begin
            SysBeep( 10 );
            writeln(' Sorry, your party has no phone listing. ')
         end;
{ Close the file Directory. }
   close(Directory);
end;
```

Using the function eof makes it easy to write the steps for displaying all of the records in the telephone directory:

```
begin
{ Open Directory for displaying all records. }
   reset( Directory, 'Telephone Directory' );
{ Continue to display the records until end of file is reached. }
   repeat
      read( Directory, Person );
      writeln(' Person:             ', Person.Full_Name );
      writeln(' Phone number: ', Person.Phone_Number );
   until eof( Directory );
end;
```

Remember that the commands seek, read, get, put, and write advance the file pointer when executed. If you attempt to execute either get or read when the next file component does not exist, eof(Pascal_File) becomes *true*, and the value of the file buffer is undefined. If you attempt to execute put or write when the file pointer is positioned beyond the end of the last file component, eof(Pascal_File) becomes *true*; the file buffer is now undefined. In the case of executing the command seek(Pascal_File, N) where N exceeds the component number of the last file component, eof(Pascal_File) becomes *true,* and the file buffer is now undefined.

10.8 MERGING TWO FILES INTO A SEQUENTIAL FILE OF RECORDS

Often it is important to merge one or more records with an existing file of records, with the merged file having the unique property of having been sorted. Often referred to as the process of merging and sorting, this process is a procedure for merging m records of one file with n records of a second file and creating a sorted file of m + n records. It requires no explicit arrays, because using them would make the maximum size of the sorted file dependent upon the maximum size of any of the arrays.

This presentation is based upon the concept of *natural merging* described by Niklaus Wirth.[2] To understand natural merging, assume the existence of two sequential files that are to be merged as well as sorted. These two files need not have the same length, and they are not assumed to be sorted. Initially, one file, called A, is appended to the end of a second file, called B, by using a third file, C, as temporary storage. File C is then sorted and, for convenience, uses two temporary files as backup storage during the sorting

2 Niklaus Wirth, Algorithms + Data Structures = Programs, (Englewood Cliffs, N.J.: Prentice Hall, 1976).

process. No internal arrays are required. Procedure Append_Files provides the steps for appending file B to file A with the appended files stored in file C:

```
procedure Append_Files( var File_A, File_B, File_C: File_Type );
{ Purpose:  This procedure appends records from file B to file }
{           A, and the complete appended file is stored in }
{           file C.}
var
   Item : Data_Type_of_File_Component;
begin
{ Establish files A and B for reading and file C for writing.}
   reset(File_A);
   reset(File_B);
   rewrite(File_C);
{ Copy records from file A and file B to File_C.}
   while  not eof(File_A)  do
   begin
      read(File_A, Item);
      write(File_C, Item);
   end;
   while  not eof(File_B)  do
   begin
      read(File_B, Item);
      write(File_C, Item);
   end;
end; { Append_Files }
```

The concept of natural merging is based on an algorithm that can recognize what is called a *run* (an ordered sequence of file components), distribute runs equally from file C over two temporary files, and merge runs from the temporary files into file C.

The term *run* refers to an ordered subsequence of file components C_i .. C_j where

C_k <= C_{k+1} for $k = i$ .. j -1

and where

$C_{i-1} > C_i$ and $C_j > C_{j+1}$

Both C_i and C_j represent end points where new subsequences (runs) begin. As an example consider the following sequence of items in file C:

```
117  131  4  59  310  41  43  77  10  22  28  45  8  3  9  15  570  34  63
```

Here the numbers { 117 131 } represent a run, since they satisfy the property that C_k <= C_{k+1} where k = 1 .. 2. It is here that for i = 3, C_{i-1} , which is 131, is greater than C_i, which is 4. A second run is the sequence { 4 59 310 }, which satisfies the properties that C_k <= C_{k+1} for k = 3 .. 5, C_{i-1}> C_i for i = 3, and C_j > C_{j+1} for j = 5. For file C there are seven runs consisting of { 117 131 }, { 4 59 310 }, { 41 43 77 }, { 10 22 28 45 }, { 8 }, { 3 9 15 570 }, and { 34 63 }. Notice from this example that a run can consist of a single item as well as several items.

The basic algorithm for natural merging requires two major steps; a distribution phase, where a run from file C is written to file A and the next run from file C is written to file B; and a merge phase that merges corresponding runs from files A and B into file C. These two steps represent a pass through file C.

The following shows these basic steps for given sequences of numbers in file C. The combination of distribution and merging is represented as a phase, and distributed runs are separated by a backslash:

```
Pass  1:

Distribution Phase:
File C:  117   131     4    59   310    41    43    77    10
          22    28    45     8     3     9    15   570    34
          63
File A:  117   131  \ 41    43    77  \  8  \ 34    63
File B:    4    59   310  \ 10    22    28    45  \  3     9
          15   570
Merge Phase:
File C:    4    59   117   131   310    10    22    28    41
          43    45    77     3     8     9    15   570    34
          63
Pass  2:

Distribution Phase:
File C:    4    59   117   131   310    10    22    28    41
          43    45    77     3     8     9    15   570    34
          63
File A:    4    59   117   131   310  \  3     8     9    15
         570
File B:   10    22    28    41    43    45    77  \ 34    63
Merge Phase:
File C:    4    10    22    28    41    43    45    59    77
         117   131   310     3     8     9    15    34    63
         570
Pass  3:

Distribution Phase:
File C:    4    10    22    28    41    43    45    59    77
         117   131   310     3     8     9    15    34    63
         570
File A:    4    10    22    28    41    43    45    59    77
         117   131   310
File B:    3     8     9    15    34    63   570
Merge Phase:
File C:    3     4     8     9    10    15    22    28    34
          41    43    45    59    63    77   117   131   310
         570
```

How many more passes are required before the file is sorted? If you examine the result of file C from the last pass, no additional passes are required. Why? Note that file C has only one run; that is, it is now completely sorted. We can conclude that in sorting file C, the phases of distribution and merging are repeated until file C has only one run.

This approach to sorting a file has an interesting property. If each of two files has been distributed N runs, merging these files will result in exactly N runs. This means that on each pass the total number of runs is divided in half, with the required number of moves of items being at worst N [$\log_2 N$]. Wirth reports that on the average the sorting time based on counting moves is even less. The expected number of comparisons is much larger, because comparisons are necessary for selecting items of a run as well as between consecutive items of a run in determining where runs terminate.

The algorithm for natural merging is written as a series of procedures in Pascal. Each procedure is listed as needed when explaining the distribution and merging phases:

```
procedure Natural_Merge( var File_C : File_Type );
{ Purpose:  This is the basic natural merge sort algorithm for }
{           sorting file C. }
var
   File_A, File_B : File_Type;
   Number_of_Runs : integer;
begin
{ Continue to execute passes over file C until file C has one }
{ run.}
   repeat
   { Distribute runs from file C over temporary files A and B. }
      Distribute_Runs(File_C, File_A, File_B);
   { Merge runs from temporary files A and B into file C. }
      Merge_Runs( File_A, File_B, File_C, Number_of_Runs );
   until  Number_of_Runs = 1 ;
end;
```

In this algorithm files A and B are local to this procedure and those procedures through which they are linked with a formal parameter. Therefore, in executing any rewrite or reset statements, these temporary files are never associated with any physical file names.

The following algorithm distributes runs from file C over each of the temporary files A and B. It begins by establishing the temporary files for writing and file C for reading, and then copies runs from file C until it reaches the end of file C.

```
procedure Distribute_Runs(var File_C,File_A,File_B: File_Type );
{ Purpose:  This procedure distributes runs from file C among }
{           files A and B.}
begin
{ Establish files A and B for writing and file C for reading. }
   rewrite(File_A);
   rewrite(File_B);
   reset(File_C);
{ Continue to copy runs from file C until the end of file C is }
{ found. }
   repeat
   { Copy a run from file C to file A. }
      Copy_Run(File_C, File_A);
   { If end of file C is not found, copy a run from file C to }
   { file B.}
      if not eof(File_C) then
         Copy_Run(File_C, File_B);
   until eof(File_C);
end;
```

The algorithm for copying a single run is simple. It will continue to read an element from one file and write it to a temporary file until either it reaches the end of the file being read or encounters an end of a run. The following represents the algorithm for copying a single run:

```
procedure Copy_Run( var File_X, File_Y : File_Type );
{ Purpose:  This routine copies a single run from file X to }
{           file Y.}
var
   Temp_Value : Data_Type_File_Component;
   End_of_Run : Boolean;
begin
{ Initialize the control variable End_of_Run.}
   End_of_Run := false;
{ Continue to read items from file X and write to file Y until }
{ an of run is encountered. }
   repeat
   { Read the next item in a run.}
      read(File_X, Temp_Value);
      write(File_Y, Temp_Value);
   { Check if a run has been completed. }
      if eof(File_X) then
         End_of_Run := true
      else
         End_of_Run := ( Temp_Value > File_X^ );
   until End_of_Run;
end;
```

Notice that the **else** clause of the **if-then-else** statement uses the Boolean expression `(Temp_Value > File_X^ )` to test if the end of a run has been reached. This expression is evaluated because `File_X` is in a read mode, and after reading the content of a file component, the file pointer is moved to the next file component, whose content is now in the file buffer represented by `File_X^`.

The algorithm for `Merge_Run` relies on using `Copy_Run` and a procedure for merging a single run from each of the temporary files. A counter is kept for counting each single run that is merged. When the end of either temporary file is reached, the remaining runs from the other temporary file are copied, and the count is incremented. The algorithm ends after all of the runs have been merged and/or copied.

```
procedure Merge_Runs(var File_A, File_B, File_C : File_Type;
                     var Number_of_Runs : integer);
{Purpose:   This routine merges runs from the temporary files }
{           A and B into file C. It returns a count of the }
{           total number of runs in file C. }
begin
{ Establish the temporary files A and B for reading and file }
{ C for writing. }
   reset(File_A);
   reset(File_B);
   rewrite(File_C);
{ Initialize the counter Number_of_Runs. }
   Number_of_Runs := 0;
{ While both ends of temporary files A and B have not been }
{ reached, merge runs from temporary files A and B into file C. }
   while ( not eof(File_A) ) and ( not eof(File_B) ) do
   begin
      Merge_Single_Run( File_A, File_B, File_C);
      Number_of_Runs := succ(Number_of_Runs);
   end;
{ If the end of file B is reached and not file A, copy the }
{ remaining runs in file A into file C.}
   while not eof(File_A) do
   begin
      Copy_Run(File_A, File_C);
      Number_of_Runs := succ(Number_of_Runs);
   end;
{ If the end of file A is reached and not file B, copy the }
{ remaining runs in file B into file C.}
   while not eof(File_B) do
   begin
      Copy_Run(File_B, File_C);
      Number_of_Runs := succ(Number_of_Runs);
   end;
end;
```

The last procedure that supports this sequence of routines is for merging a single run between two temporary files, A and B. This algorithm reads file components until a single run has been completed. If the next element of file A is less than or equal to the next element in File B, then the next component of file A is copied into file C . If, after copying the next file component of file A to C, the end of the run is encountered in file A, then the run that remains in file B is copied to file C, after which the algorithm is terminated. If the next component of file A is greater than the next component of file B, the next component in file B is copied to file C. If, after copying the next file component of file B to C, the end of the run is encountered in file B, then the run that remains in file A is copied to file C, after which the algorithm is terminated. This algorithm continues to execute until it finds an end-of-run.

```
procedure Merge_Single_Run( var File_A, File_B, File_C :
File_Type );
{Purpose:   This routine will copy and merge the elements of }
{           corresponding runs from files A and B until an end- }
{           of-run occurs in either file A or B. The elements }
{           that are merged and copied to C are sorted in a }
{           proper order.}
  var
    Temp_Value : Data_Type_File_Component;
    End_of_Run : Boolean;
begin
{ Continue to read file components from files A and B until an }
{ end-of-run is encountered. }
   repeat
   { Check if the run to be copied begins in file A or file B. }
      if ( File_A^ <= File_B^ ) then
         begin
         { Copy a single file component from file A to file C. }
            read(File_A, Temp_Value);
            write(File_C, Temp_Value);
            if eof(File_A) then
               End_of_Run := true
            else
               End_of_Run := ( Temp_Value > File_A^ );
         { If an end-of-run is reached in file A, copy the }
         { remainder of the run in file B to file C. }
            if End_of_Run then
               Copy_Run( File_B, File_C );
         end
      else
         begin
         { Copy a single file component from file B to file C. }
            read(File_B, Temp_Value);
            write(File_C, Temp_Value);
            if eof(File_B) then
               End_of_Run := true
            else
               End_of_Run := ( Temp_Value > File_B^ );
         { If an end-of-run is reached in file B, copy the }
         { remainder of the run in file A to file C. }
            if End_of_Run then
               Copy_Run( File_A, File_C );
         end;
   until End_of_Run;
end;
```

The combination of these routines into a Pascal program and the testing of the natural merge algorithm is left to the reader as an exercise.

10.9 TEXT FILES

Pascal supports a predefined file type called *text*. Text files are different from other types of files, in that each file component is a series of zero or more Macintosh characters terminated by an end-of-line marker. In addition, file components of a text file have varying lengths; they are not of a fixed size. In a more general sense we can consider a text file as a file containing a sequence of text lines.

This is not the case for other file types. Although it is common to think of a text file as a file of characters, in Pascal the type **file of** char is not equivalent to the type text. Why? In a **file of** char each file component can contain one and only one character, so that each file component has a fixed size. In a text file, a file component can contain several characters and is always terminated by an end-of-line marker.

For text files we can use the commands read, readln, write, and writeln to read from or write to a Pascal file. If our Pascal file is of type text, the command

```
write( Pascal_File, e1, e2, e3, . . . , en );
```

or

```
writeln( Pascal_File, e1, e2, e3, . . . , en );
```

converts the internal forms of each expression e1, e2, ... , en into an equivalent character-string representation, directing these character representations to a file component of the Pascal file. Although the commands write and writeln are similar in the actions that they perform, the writeln command terminates the file component of a text file with an end-of-line marker. The following program lines represent several equivalent statements for writing values of expressions to a file component of type text.

```
writeln( Pascal_File, e1, e2, . . . , en );
```

is equivalent to

```
begin
   write( Pascal_File, e1, e2, . . . , en );
   writeln( Pascal_File )
end;
```

which is also equivalent to

```
begin
   write( Pascal_File, e1);
   write( Pascal_File, e2);
   . . .
   write( Pascal_File, en);
   writeln( Pascal_File )
end;
```

Failing to terminate the output of a file component by not executing a writeln command can result in later execution of a write or writeln command recording information to the same file component. For nontext files the writeln command *cannot* be employed for writing information to a Pascal file. The command write executes differently for a nontext file; it records values in direct machine form, not as character strings.

If our Pascal file is of type text, the command

```
read( Pascal_File, v1, v2, . . . , vn );
```

or

```
readln( Pascal_File, v1, v2, . . . , vn );
```

reads values for each variable in the variable list. Execution of this statement is different from reading values for variables from a nontext file. First, for a Pascal file of type `text` the `read` statement enters a string of characters representing the value of a variable. The Pascal system then converts this string representation of the variable into internal machine form. This differs from nontext files, in which the values for variables are always stored in machine form. Second, several different values can be read from a single file component of a text file, separated by one or more blanks. For nontext files we can only do this by using a file of records. Third, the command `readln` can only be employed with text files. When executed, it terminates the entry of values from a file component. Further execution of `read` or `readln` commands will read values from the next file component of the text file.

There are two special functions exclusively for use with text files, `eoln` and `Page`. The `Boolean` function `eoln(Pascal_File)` returns the value *true* if the end-of-line pointer is pointing to the end-of-line marker. If not, the value returned is *false*. The second function is `Page(Pascal_File)`, for paging an output file. Execution of this procedure causes the output file to skip to the top of a new page when the output file is displayed or printed.

The program titled `End_of_Line` reads several lines of text from the keyboard, character by character. Pressing the Return key allows a new line of text to be entered. It also results in the program writing to a new file component of the output file, provided that the next character pressed is not ~. This example uses the function `eoln` to test for an end-of-line marker from the standard input file. Although it is not possible on input from the keyboard to apply the function `eof`, this function is used to test for the end-of-file of the text file called `Input_File`.

```
program End_of_Line(input, output);
{ Purpose:  This program shows a simple example of a Pascal file }
{           of type text. Both the functions eoln and eof are }
{           used for detecting end-of-line or end-of-file. }
   const
      Endmarker = '~';
   var
      Input_File, Output_File : text;
      A : char;
      S : string;
begin
{ Open text file for output. }
   rewrite(Output_File, ' Text File');
{ Establish header for output file. }
   writeln(' Enter message: ');
   writeln(Output_File, ' Your message: ');
   writeln(Output_File);
   write(Output_File, ' ');
{ Adjust the first file component for aligning the left margin }
{ in the display of lines. }
   repeat
```

```
      repeat
      { Read the next character. }
         read(A);
      { Check for an end marker. }
         if A <> Endmarker then
         { Write a character to file component. }
            write(Output_File, A);
      until (eoln(Input)) or (A = Endmarker);
      { Terminate line to file component. }
         writeln(Output_File);
   until A = Endmarker;
{ Terminate input line from the terminal. }
   readln;
{ Close text file for output. }
   close(Output_File);
{ Open text file for reading. }
   reset(Input_File, ' Text File');
   repeat
   { Displaying text lines. }
      readln(Input_File, S);
      writeln(S);
   until eof(Input_File);
   close(Input_File);
end.
```

Note that the first part of this program (where a nested **repeat-until** statement exists) deals with what we referred to in Chapter 9 as the *lazy-input* problem for entering data from the keyboard. Although we can detect that an end-of-line marker has been entered by employing the function `eoln`, this requires at least one character to have been entered from the keyboard. For standard input there can always be another `read` or `readln` command that can find its data from the same input line, so it is impossible for the Pascal system to adjust the current file pointer and check if there is a next file component or to test if it has reached the end of the file.

Before leaving this section, recall that text files can be interchanged between a word processor (such as WORD or MacWrite) and Macintosh Pascal. Therefore, it is possible to create MacWrite or WORD files that can be read as text files by Macintosh and THINK Pascal. It is also possible to create text files with Pascal that can be read as character files by WORD or MacWrite.

10.10 REFERENCING DEVICES ON THE MACINTOSH AS FILE DEVICES

On the Macintosh computer there are basically four devices that can be referenced through the commands `open`, `reset`, and `rewrite`. These include one or two of the disk drives, a printer, and a modem. When referencing disk drives, the proper disk title must be used along with the physical file name if the file we are referencing is not stored on the Macintosh Pascal System diskette. The disk title is given by the volume name followed by the folder name followed by the file name. The syntax for referencing a file follows:

```
Volume_Name : Folder_Name : Physical_File_Name
```

The proper title when referencing the printer as an output file device is

```
printer:
```

Because the printer is a write-only device, the title `printer:` can only be used as a physical file name with the command `rewrite`. A syntax error occurs if it used with the commands `open` or `reset`.

A modem can act as both an input and output file device. When using a modem, the title associated with the physical file name is

```
modem:
```

Macintosh and THINK Pascal set the baud rate at 300 whenever the modem is being treated as a file device. Pascal files being linked with the physical files `printer:` or `modem:` must be text files. Using any other file types will result in an error. For both `printer:` and `modem:`, file names appearing after the colon are ignored.

The following is a short program titled `Using_Printer_As_File`. As you see from the program listing, the printer is treated as a text file. In this mode of operation, the characters printed are in a standard font, with the program having no control over the size and type of font. The option **Font Control** from the menu option **Windows** has no effect on the characters that are printed.

```
program Using_Printer_As_File(input, output);
{ Purpose:  This program uses the printer as an output text }
{           file. }
   var
      Input_File : text;
      Output_File : text;
      Line_of_Text : string;
begin
{ Open a test file for storing text lines. }
   rewrite(Input_File, ' Test Text File');
{ Store the following lines of text. }
   begin
      writeln(Input_File, ' ****** MESSAGE TO USER ******');
      writeln(Input_File);
      writeln(Input_File, ' This is a test using the printer as');
      writeln(Input_File, ' an output file. The text that you ');
      writeln(Input_File, ' presently see has initially been');
      writeln(Input_File, ' stored in a text file called "Test');
      writeln(Input_File, ' Text File." After it is stored, the');
      writeln(Input_File, ' file pointer is reset and the ');
      writeln(Input_File, ' printer opened as an output file.');
      writeln(Input_File, ' Each line of text is read from the ');
      writeln(Input_File, ' physical file "Test Text File" and');
      writeln(Input_File, ' written to the printer until the ');
      writeln(Input_File, ' program has reached the end-of-');
      writeln(Input_File, ' file.');
      writeln(Input_File);
      writeln(Input_File, '********END OF MESSAGE *********');
   end;
```

```
{ Reset the the file pointer for Input_File and open the file }
{ Output_File. }
   reset(Input_File);
   rewrite(Output_File, 'printer:');
{ Print the contents of Input_File to the printer. }
   Page(Output_File);
   while not eof(Input_File) do
      begin
         readln(Input_File, Line_of_Text);
         writeln(Output_File, Line_of_Text);
      end;
{ Close both the input and output files. }
   close(Input_File);
   close(Output_File);
end.
```

For both Macintosh and THINK Pascal the Text window is treated as a write-only file having the physical file name `TextWindow`; its predefined logical file name is `Output`. The keyboard is treated as a read-only file having the physical file name `Keyboard`; its predefined logical file name is `Input`.

10.11 AN APPLICATION: A SIMPLE DATABASE SYSTEM

In this section we discuss the development and implementation of a simple database system. The purpose of this system is to store names and addresses, using the file and window techniques of Macintosh Pascal. Let us assume that this system will have the following options:

1. Create a new file.
2. Open an existing file.
3. Insert a new record of information.
4. Delete an old record of information.
5. Correct a record of information.
6. Display a record of information.
7. Display all records of information.
8. Clear the screen and exit from the program.

All files are assumed to be composed of unsorted records stored sequentially, with each record containing five fields: full name, street address, city, state, and zip code. For the first two options, we use two special Macintosh Pascal functions: `NewFileName` and `OldFileName`. When executed, both `NewFileName( Prt_Message )` and `OldFileName(Prt_Message)` display a dialog box containing the `Prt_Message`. Figure 10.4 shows examples of both dialog boxes.

In the execution of `NewFileName`, entering a name from the keyboard results in the function returning this string as a value for the function. For `OldFileName` the dialog box allows the user to choose an existing file by using the mouse. In this case the value returned is the name of the file chosen from within a window in the dialog box. Though both functions return `string`-type values, neither has any effect on opening, resetting, or rewriting a file. It is the programmer's responsibility to code for opening a physical file.

For the third option, information is to be entered from the keyboard and then appended, as a new record, to the end of an existing sequential file. At this point we assume that commands such as `seek` and `eof` are sufficient for reaching the end of the file. The fourth option requires a temporary file for storing records while a search is performed to find the record to be deleted. This procedure borrows some of its concepts from the merge-sort algorithm discussed in Section 10.4. For the fifth and sixth options, we apply the concepts of the linear search algorithm for locating a record for updating and displaying. By again using the commands `open` and `seek`, we can apply a simple set of steps to update a record in a sequential file. The seventh option requires that the user be given a choice for listing all of the records to the Drawing window or to the printer.

We will require that Options 3 through 7 be executed only if a new file has been created using Option 1 or if an existing physical file has been found by using Option 2.

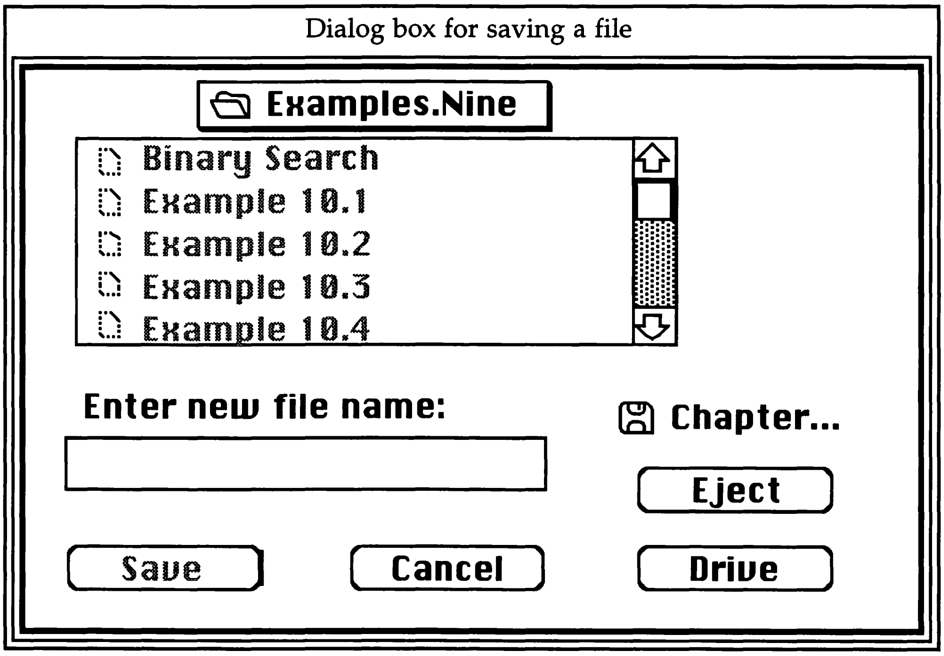


Figure 10.4 Examples of dialog boxes using the routines `OldFileName` and `NewFileName`.

Dialog box for opening a file

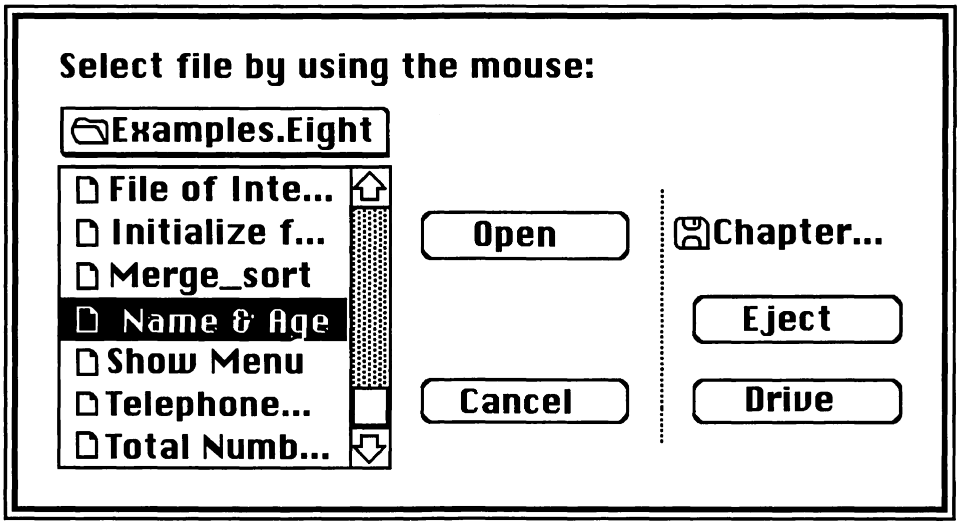


Figure 10.4 (Continued)

Figure 10.5 is an initial view of the module hierarchy. Notice that separate modules exist for initializing data, presenting the menu, creating a new file name, finding an existing file, appending a new record, deleting an old record, updating an existing record, displaying a single record, and displaying all the records. The module `Initialization` initializes several variables: a file flag indicating if a file has been created or located, a table called `Message` used for storing a list of 11 prompts for display by the module `Menu`, a list of records, and a table defining the boundaries of 11 different rectangles.

The steps in the main module are as follows:

1. Initialize a table of messages with prompts and a table of regions defining the boundaries of rectangles.
2. Present the menu to the user for selecting a choice.
3. From the choice made in Step 2, select one of the eight options to be executed.
4. Repeat Steps 2 through 4 until the choice is to exit from the system.

The following is a refinement of our solution, with Steps 2 through 4 represented by a **`repeat-until`** construct, and with Step 3 replaced by a **`case`** statement :

```
begin
{ Initialize the variable File_Flag. }
   Initialize( File_Flag, Message, Box );
   repeat
   { Show menu and return choice. }
      Menu( Choice, Message, Box );
      case Choice of
         1:  Create_Newfile( File_Name, File_Flag );
         2 : Select_Oldfile( File_Name, File_Flag );
```

```
        3:  Append_Record( File_Name, File_Flag );
        4:  Delete_Record( File_Name, File_Flag );
        5:  Update_Record( File_Name, File_Flag );
        6:  Display_Record( File_Name, File_Flag );
        7:  List_Records( File_Name, File_Flag, Message, Box );
        8:  {Clear the screen. }
    end; { case }
  until Choice = 8;
end.
```

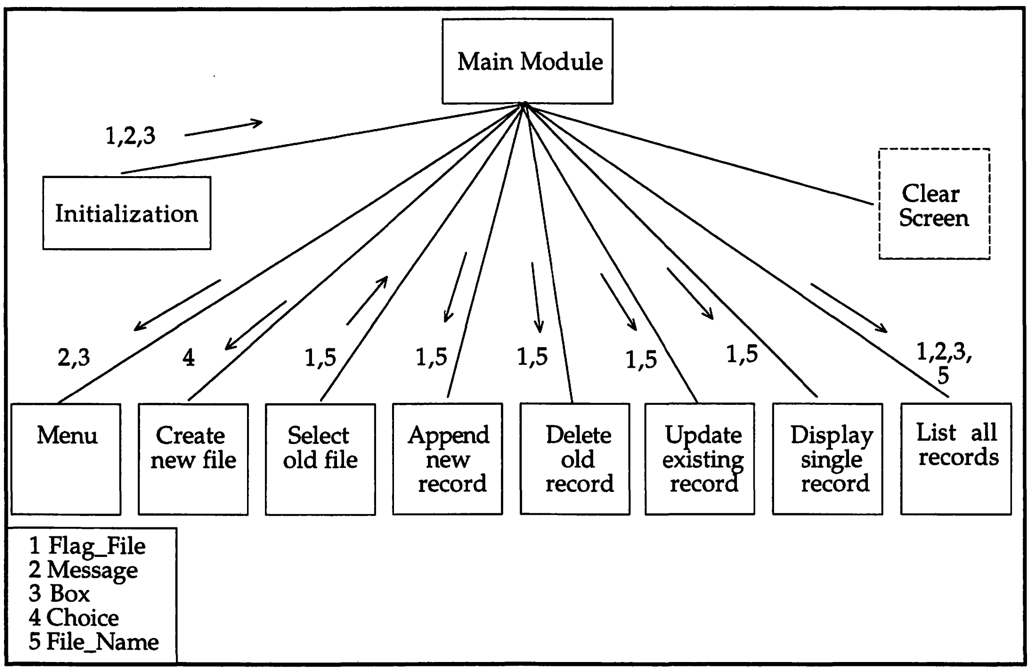


Figure 10.5 Initial view of the module hierarchy of Name_Address_System.

As you can see, our main module requires only five variables: File_Name, representing the physical file; File_Flag, of type Boolean ; Choice, of type integer; Message, containing an array of prompts; and Box, an array of predefined rectangles. In addition to these variables there are also four global user-defined types called Person, Table, Squares, and List. These user-defined types have the following definitions:

```
type
  Person = record
              Full_Name: string[40];
              Address: string[30];
              City: string[20];
              State: string[15];
              Zip: string[12]
           end;
```

```
   Table = array[1..11] of string;
   Squares = array[1..11] of Rect;
   List = file of Person;
```

When entering Options 3 through 7, the physical file being accessed will be opened and remain open only for the length of the particular option. This is in keeping with the philosophy that a file should remain open only as long as it is needed.

At this point the main module, along with dummy modules representing the nine modules at the second level of Figure 10.4, are written in Macintosh Pascal and tested. Because the module Menu must return a value for the variable Choice, this procedure can initially be written either to prompt for a value using the Text window or to assign a value with a simple assignment statement. As the modules are developed and refined, we can integrate them into the present system and test them. This is a simple top-down approach for both the integration and the testing of modules. Module definitions for Initialize, Create_Newfile, and Select_Oldfile follow:

```
procedure Initialize;
{ Purpose:  This procedure requires three formal variable }
{           parameters: Fileopen, a Boolean type; Message, of }
{           user-type Table; and Box, of user-type Squares. }
begin
   Fileopen <-- false;
{ Assign messages to the array Message. }
{ Define rectangles for options in Menu and Listing records. }
end; { Initialize }
```

```
procedure Create_Newfile;
{ Purpose:  This procedure requires two formal variable }
{           parameters: Physical_File, of type string, and }
{           Fileopen, of type Boolean. }
   var
      Pascal_File : List;
begin
{ Prompt user for creating new file. }
   Physical_File <-- NewFileName(' Enter new file name: ');
{ Check if the user has typed a file name in the dialog box. }
   if Physical_File <> '' then
      begin
         Fileopen <-- true
      { Establish the file on the diskette. }
         open( Pascal_File, Physical_File );
         close( Pascal_File )
      end
   else
      Fileopen <-- false
end; { Create_Newfile }
```

```
procedure Select_Oldfile;
{ Purpose:  This procedure requires two formal variable }
{           parameters: Physical_File, of type string, and }
{            Fileopen, of type Boolean. }
begin
{ Prompt user for opening an existing file. }
   Physical_File <-- OldFileName(' Select file by using mouse: ');
{ Check if user has selected a file name from the dialog box. }
   if Physical_File <> ''  then
      Fileopen <-- true
   else
      Fileopen  <--  false
end; { Select_Oldfile }
```

The Menu module presents the eight options to the user by displaying them to the Drawing window. Figure 10.6 shows the format of the display. The user chooses an option by moving the cursor to one of the small squares and pressing the mouse button.

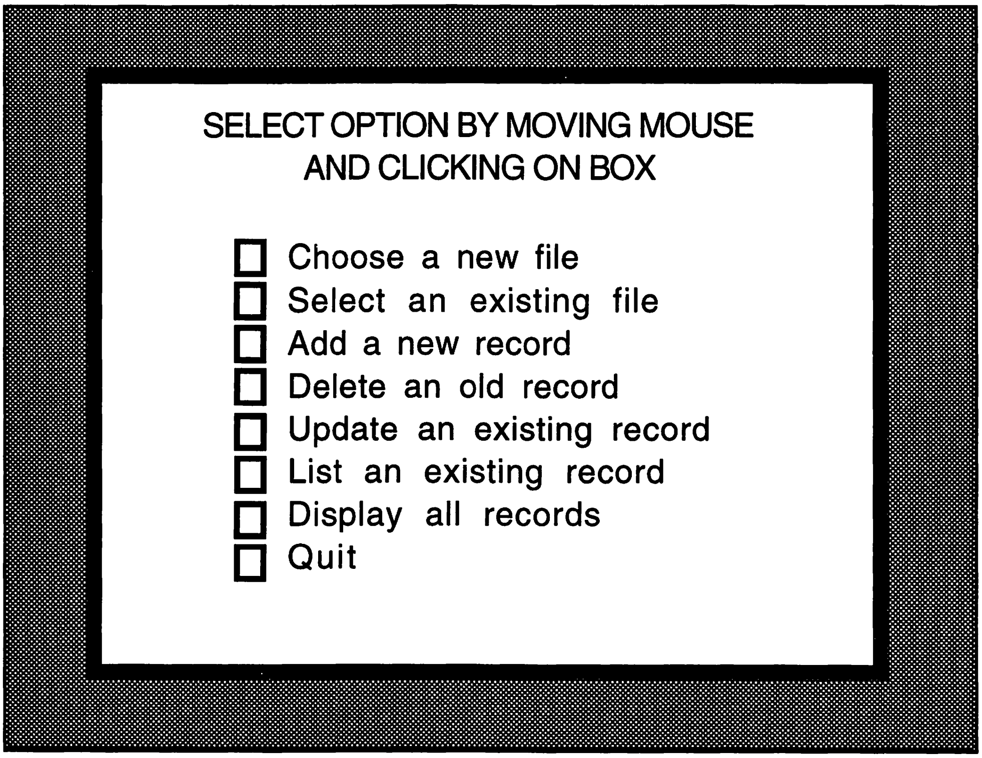


Figure 10.6 Window displaying the menu of Name_Address_System.

The steps for achieving these results are as follows:

1. Initialize the Drawing window, and then display the center window for options.
2. Title the top of the window.

3. Set PenSize, and present the eight options to the user.
4. Continue checking if the mouse lies within one of the boxes and if the mouse button has been pressed.

The following is a refinement of our solution for the module Menu:

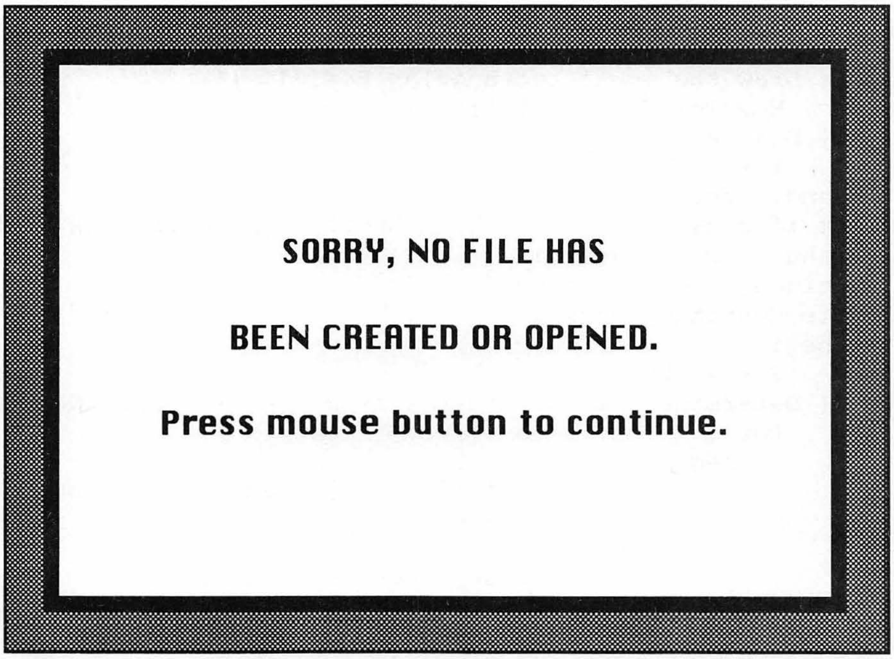
```
procedure Menu;
{ This procedure requires three formal parameters: Choice, an
integer variable for returning the menu choice; Message, a value
parameter of user-type Table; and Box, a value parameter of user-
type Squares. Local variables for this module include X, Y, J,
Continue_Check, and a mouse point. }
 begin
{ Initialize the Drawing window. }
   Show_Drawing_Window;
{ Draw center rectangle. }
   Draw_Centerbox;
{ Present options to user. }
   MoveTo( 110, 70);
{ Display message shown in the window for menu. }
   PenSize( 2, 2 ); X <-- 130; Y <-- 120;
   for J <-- 1 to 8 do
      begin
         MoveTo( X, Y );
      { Draw the small rectangular Box[J]. }
         MoveTo( X + 20, Y );
      { Draw Message[J]. }
         Y <-- Y + 20
      end; {for-loop}
{ Check if mouse button has been pressed and cursor is within one
   of the  eight boxes. }
   Continue_Check <-- true;
   while Continue_Check do
      begin
         Y <-- 120;
      { Determine the coordinates from the procedure GetMouse. }
         for J <-- to 8 do
            begin
               if { Point is in Box[J]}  and { mouse button is
                        pressed }
                  begin
                     Continue_Check <-- false;
                     Choice <-- J
                  end; { if-then }
            end;  {for-loop}
      end; { while-do }
end; { Menu }
```

To test if the mouse is located in one of the eight squares, we can conveniently apply the `Boolean` function `PtInRect`. (See Chapter 14 for more information on this function.) This function takes as input a point and checks to see if it lies within the boundary of a defined rectangle. As you see, the module Menu requires the support of two additional modules, Show_Drawing_Window, for hiding all windows and for showing the Drawing window, and Draw_Centerbox, for displaying the center window. Each of the eight squares has been predefined during the execution of the module Initialization and is passed to this module, by means of an array, along with the table of prompts.

The fifth module, called Append_Record, prompts for a new record. After receiving this information, it also appends the new record to the end of a physical file. The major steps in performing this function follow:

1. Check if a physical file has been found or created.
2. If the physical file does not exist, report a "no file" message to the user.
3. If the physical file does exist, prompt the user for name and address.
4. Open the physical file, and set the file pointer to the end-of-file.
5. Assign the new record to the file buffer, and put the new record to the physical file.
6. Close the physical file.

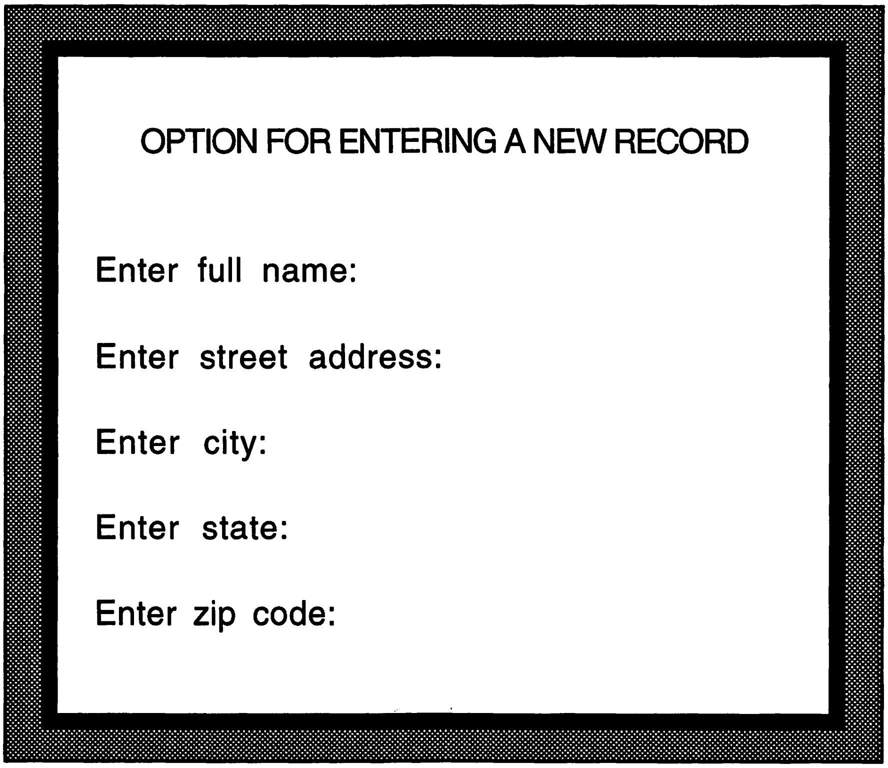
Step 1 checks to see if either Option 1 or 2 has been successfully executed and, if not, displays the message shown in Figure 10.7. If the physical file does exist, Step 3 prompts the user, employing the window format shown in Figure 10.8.


SORRY, NO FILE HAS

BEEN CREATED OR OPENED.

Press mouse button to continue.

Figure 10.7 Window for creating a "no file" message.



OPTION FOR ENTERING A NEW RECORD

Enter full name:

Enter street address:

Enter city:

Enter state:

Enter zip code:

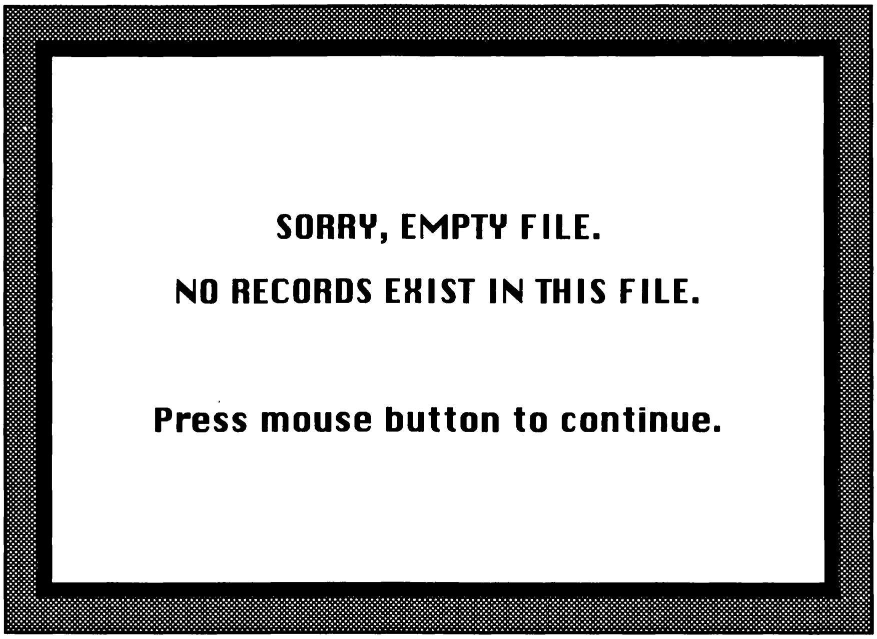
Figure 10.8 Window showing the name-address prompt.

The following is a refinement to the foregoing abstract solution:

```
procedure Append_Record;( Physical_File: string; Flag : Boolean );
{ Purpose:  This procedure uses two formal value parameters: }
{           Physical_File, of type string, and Flag, of type }
{           Boolean. Local variables include Pascal_File, Name, }
{           and Counter. }
begin
   if  Flag  then
{ Check if a file has been opened. }
      begin
      { Prompt user for name and address. }
         Enter_Name_Address( Name );
      { Open physical file for appending record. }
         open( Pascal_File, Physical_File );
      { Continue to search for the end-of-file. }
         Counter <-- 0;
         while not eof( Pascal_File ) do
            begin
               seek( Pascal_File, Counter );
               Counter <-- Counter + 1
            end;
            Pascal_File^ <-- Name;
            put( Pascal_File );
            close( Pascal_File);
         end
      else
      { Report that no file has been created or opened. }
         Display_Nofile_Message;
   end;  { Append_Record }
```

This module requires two supporting modules: `Enter_Name_Address`, for entering a new record from the keyboard, and `Display_Nofile_Message`, for displaying a message that the requested file does not exist. At this point dummy procedures are coded for `Display_Nofile_Message` and `Enter_Name_Address`. In the case of the module `Enter_Name_Address`, one or more assignment statements are sufficient for establishing a new record without explicitly prompting the user. For now, the procedure `Display_Nofile_Message` can only print a message to the Text window. Later these two procedures will be given formal definitions, integrated into the existing system, and tested.

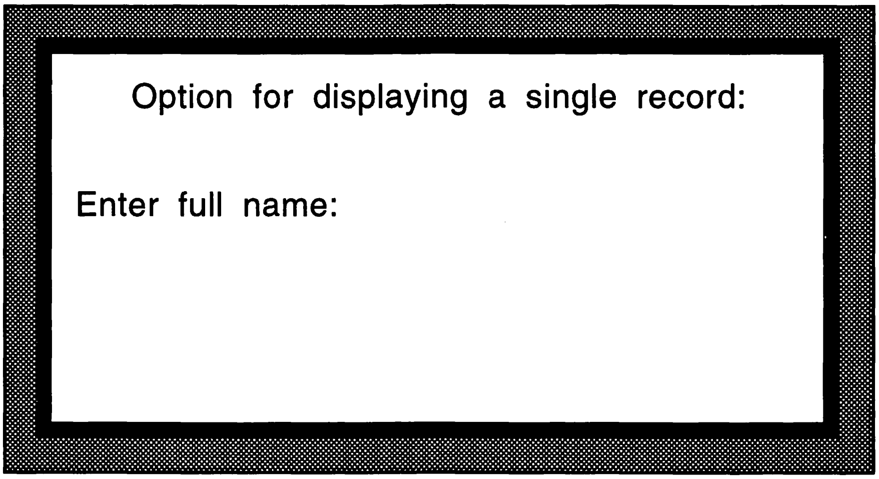
The sixth module, `Delete_Record`, prompts the user for a person's name, and uses it as a key to search for an existing record; if found, this record is deleted from a physical file. Step 4 displays the message shown in Figure 10.9, and Step 5 displays the window format shown in Figure 10.10.


SORRY, EMPTY FILE.

NO RECORDS EXIST IN THIS FILE.

Press mouse button to continue.

Figure 10.9 Window showing the "empty file" message.


Option for displaying a single record:

Enter full name:

Figure 10.10 Window showing the prompt for displaying a single record.

The major steps for performing this function follow:

1. Check if a physical file has been found or created.
2. If the physical file does not exist, report a "no file" message to the user.
3. Open the physical file, and check if it is empty.
4. If the physical file is empty, report an "empty file" message to the user.
5. If the physical file is a nonempty file, display the center window, and prompt the user with a title and a request for a person's full name.
6. Open the temporary file for output.
7. Read a record from the physical file, and compare it with the search key.
8. Write this record to the temporary file if it does not contain the search key.
9. Repeat Steps 7 through 9 until the end of the physical file is reached, or the record to be deleted has been located.
10. If the record is found, read the next record, and write this and any of the remaining records of the physical file to the temporary file.
11. Reset the file pointers for both files; then continue to read each record from the temporary file and write the record to the physical file.
12. Close the physical file.

The following algorithm represents a refined definition of the solution just given.

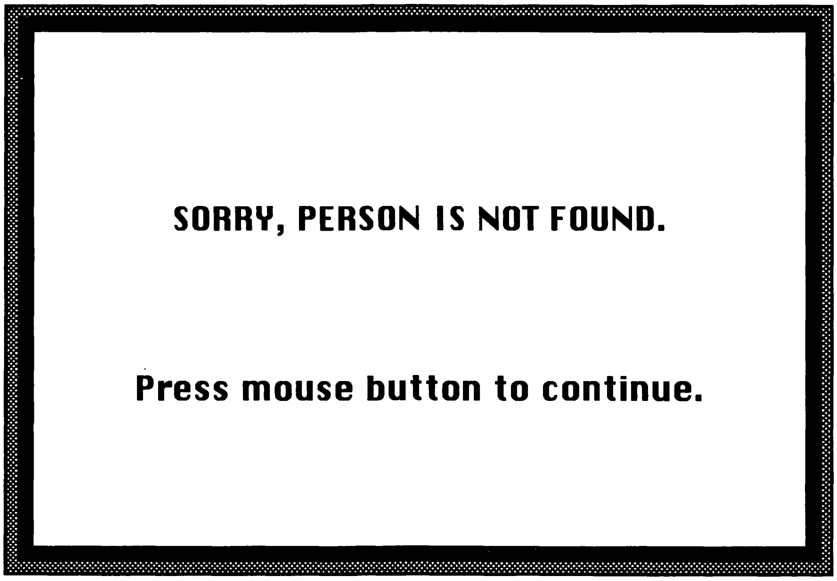
```
procedure Delete_Record;
{ Purpose:  This procedure requires two formal value parameters: }
{           Physical_File, of type string and Flag, of type }
{           Boolean. Local variables include Pascal_File, Name, }
{           Data, and Not_Found. }
begin
{ Check if the file has been opened or created. }
   if Flag then
      begin
      { Open the physical file and check if it is empty. }
         reset( Pascal_File, Physical_File );
         if eof(Pascal_File) then
            begin
               Report_Empty_File;
               close( Pascal_File );
            end
         else
         { Prompt user for the name of the person to search }
         { the records. }
            begin
            { Draw the center box and title top of the window. }
            { Open the temporary file. }
               rewrite( Temporary_File );
               Prompt_for_Name( Name );
               Not_Found <-- true;
               while ( not eof( Pascal_File ) ) and Not_Found do
                  begin
                     read( Pascal_File, Data );
                     { Check if Name is equal to the Full_Name
                        field of Data }
                        if  Name = Data.Full_Name  then
                           Not_Found <-- false
                        else
                           write( Temporary_File, Data )
                  end;
            { Read remaining records from the physical file and }
            { write to temporary file. }
               while not eof( Pascal_File ) do
                  begin
                     read( Pascal_File, Data );
                     write( Temporary_File, Data )
                  end;
            { Copy records from temporary file to Pascal file. }
               rewrite( Pascal_File );
               reset( Temporary_File );
               while not eof( Temporary_File ) do
                  begin
                     read( Temporary_File, Data );
                     write( Pascal_File, Data )
                  end;
            { Close physical file. }
               close( Pascal_File )
            end
         end
      end
   else
      Display_Nofile_Message
end; { Delete_Record }
```

This module requires two supporting modules: Prompt_for_Name, for entering the search key and the person's full name; and Display_Nofile_Message. When testing the module Delete_Record, we write a dummy module for Prompt_ `for_Name` that gives a person's name by means of an assignment statement. `Prompt_for_Name` can be refined later, integrated, and tested with the present system. Notice that even though we explicitly close the physical file represented by `Pascal_File`, any attempt to close the temporary file causes an execution error. Because the temporary file is opened with no physical file link, explicit closing of the file is unnecessary.

The module called `Update_Record` will prompt the user for a person's name, search for a record containing this person's name, prompt the user to correct any of the fields, and then update the existing record. The steps in updating a record follow:

1. Check if a physical file has been found or created.
2. If the physical file does not exist, report a "no file" message to the user.
3. Open the physical file, and check if it is empty.
4. If the physical file is empty, report an "empty file" message to the user.
5. If the physical file is a nonempty file, display the center window, and prompt the user with a title and a request for the person's full name.
6. Read a record from the physical file, and compare it with the search key.
7. Repeat Steps 6 and 7 until the end of the physical file is reached, or the record has been located.
8. If the record is not found, report that the person is not in the file.
9. If the person is found, display the information in the existing record.
10. Prompt the user for corrections.
11. Write the updated record to the physical file.
12. Close the physical file.

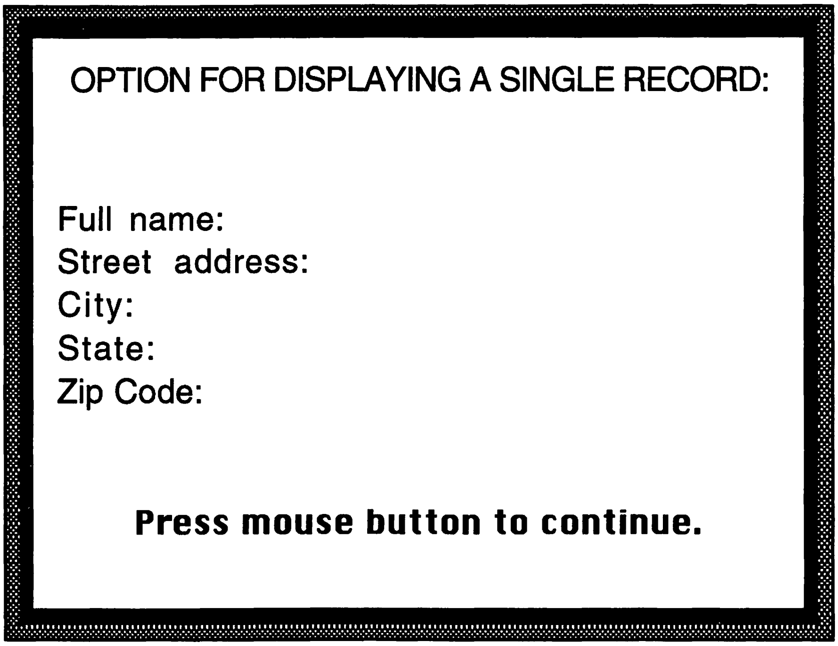
Step 8 displays the window shown in Figure 10.11; Step 9 displays the window of Figure 10.12. Updating the record is requested by the window format shown in Figure 10.13.



SORRY, PERSON IS NOT FOUND.

Press mouse button to continue.

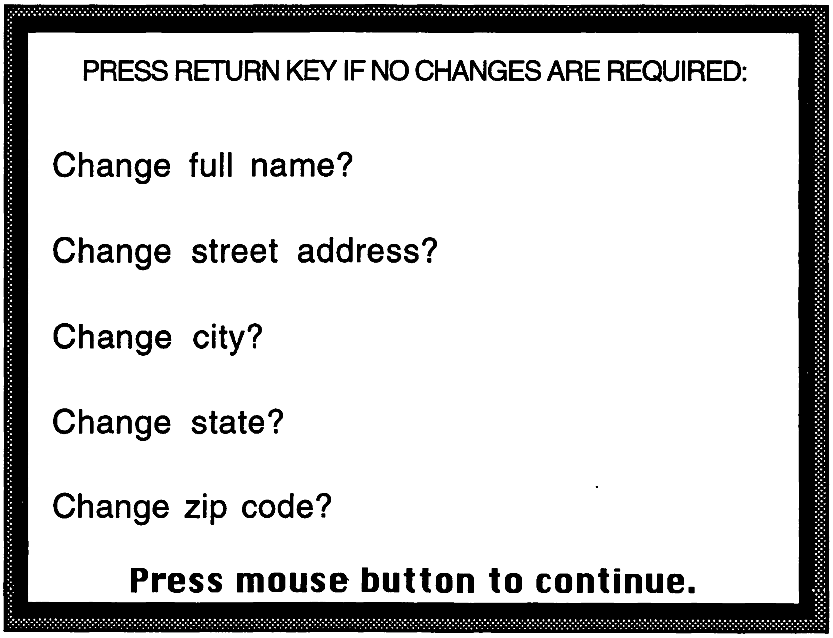
Figure 10.11 Window showing the "person not found" message.


OPTION FOR DISPLAYING A SINGLE RECORD:

Full name:
Street address:
City:
State:
Zip Code:

Press mouse button to continue.

Figure 10.12 Window for displaying a single record.


PRESS RETURN KEY IF NO CHANGES ARE REQUIRED:

Change full name?

Change street address?

Change city?

Change state?

Change zip code?

Press mouse button to continue.

Figure 10.13 Window prompting the user to enter changes for a single record.

A refined solution follows:

```
procedure Update_Record ;
{ Program:  This procedure requires two formal value parameters: }
{           Physical_File, of type string, and Flag, of type }
{           Boolean.  Local variables include Pascal_File, Name, }
{           Data, Counter, and Not_Found. }
begin
{ Check if a file exists or has been created. }
   if Flag then
      begin
      { Open the physical file, using the open command. }
         open( Pascal_File, Physical_File );
      { Check if the physical file is an empty file. }
         if eof( Pascal_File ) then
         { Report empty file and close physical file. }
            begin
               Report_Empty_File;
               close( Pascal_File )
            end
         else
            begin
            { Draw the window in the center of the screen using
               Draw_Centerbox; then present a prompt at the top. }
            { Prompt user for name of person. }
               Prompt_for_Name( Name );
            { Initialize two local variables. }
               Not_Found <-- true;
               Counter <-- 0;
               while ( not eof( Pascal_File ) ) and Not_Found do
                  begin
                  { Read record from the file buffer and test if
                     Full_Name field is equal to the search key. }
                     Data <-- Pascal_File^;
                     if Name = Data.Full_Name then
                        Not_Found <-- true
                     else
                        begin
                        { Seek the next record for matching with }
                        { the search key. }
                           seek( Pascal_File, Counter );
                           Counter <-- Counter + 1
                        end;
                  { Prompt for further information from user if }
                  { the record has been found. }
                     if Not_Found then
                     { Report the record has not been found. }
                        Report_Not_Found
                     else
                        begin
                        { Display present record.}
```

```
                              Display_Name_Address( Data );
                           { Prompt user for corrected information. }
                              Prompt_for_Changes( Data );
                           { Put updated information of record in }
                           { the physical file. }
                              Pascal_File^ <-- Data;
                              put( Pascal_File );
                           end
            end
         close( Pascal_File )
      end
   else
      Display_Nofile_Message
end; { Update_Record }
```

In this module we need the open command for opening the physical file, because the seek command is used to adjust the file pointer. The write command is used to update the existing record. This module requires seven supporting modules: Report_Empty_File, Draw_Centerbox, Prompt_for_Name, Report_Not_Found, Display_Name_Address, Prompt_for_Changes, and Display_Nofile_Message. As before, dummy modules are written and integrated for testing the module Update_Record. These are refined, integrated, and tested at a later time with the present system.

The eighth module, called Display_Record, prompts the user for a person's full name, searches for a record containing that name, and displays the complete record. The major steps in this procedure follow:

1. Check if a physical file has been found or created.
2. If the physical file does not exist, report a "no file" message to the user.
3. Open the physical file, and check if it is empty.
4. If the physical file is empty, report an "empty file'"message to the user.
5. If the physical file is a nonempty file, display the center window, and prompt the user with a title and a request for the person's full name.
6. Read a record from the physical file, and compare it with the search key.
7. Repeat Steps 6 and 7 until the end of the physical file is reached, or the record has been located.
8. If the record is not found, report that person is not in the file.
9. If the person is found, display the information in the existing record.
10. Close the physical file.

Refined steps for displaying a single record of information follow:

```
procedure Display_Record;
{ Purpose:  This procedure requires two formal value parameters: }
{           Physical_File, of type string, and Flag, of type }
{           Boolean. Local variables include Pascal_File, Name, }
{           Data, and Not_Found. }
begin
{ Check if the file exists or has been created. }
   if Flag then
      begin
      { Open the physical file for reading records. }
         reset( Pascal_File, Physical_File );
         if eof( Pascal_File ) then
            begin
            { Report empty file. }
               Report_Empty_File;
               close( Pascal_File )
            end
         else
            begin
            { Draw the center window using Draw_Centerbox and
               display a message at the top of the window.}
            { Prompt user for person's name. }
               Prompt_for_Name( Name );
            { Continue to read records until the file is empty }
            { or the record is found. }
               Not_Found <-- true;
               while ( not( eof( Pascal_File ) ) and Not_Found do
                  begin
                  { Continue to read records and check if key }
                  { matches Full_Name field. }
                     read( Pascal_File, Data );
                     if Name = Data.Full_Name then
                     Not_Found <-- false
                  end; { while-do }
            { If the record is not found, report an appropriate }
            { message; otherwise, report the record to the user. }
               if Not_Found then
                  Report_Not_Found
               else
                  Display_Name_Address( Data );
            { Close the physical file. }
               close( Pascal_File )
        end;
     else
       Display_Nofile_Message
end; { Display_Record }
```

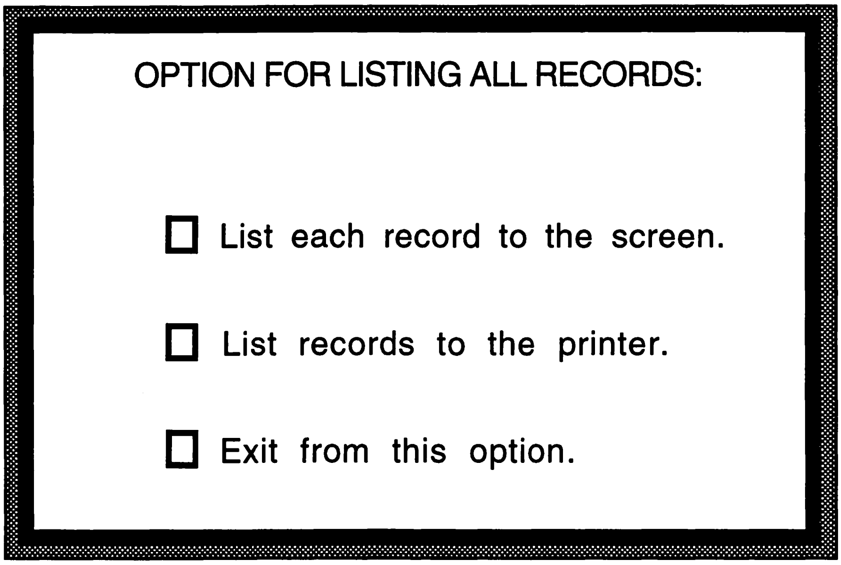
Supporting modules required by `Display_Record` are `Report_Empty_File`, `Draw_Centerbox`, `Prompt_for_Name`, `Report_Not_Found`, `Display_Name_Address`, and `Display_Nofile_Message`.

The last module, called `List_Records`, provides the user with three options: List all the records to the screen, list all the records to the printer, or exit from the option. The steps for listing all records follow:

1. Check if a physical file has been found or created.
2. If the physical file does not exist, report a "no file" message to the user.
3. Open the physical file, and check if it is empty.
4. If the physical file is empty, report an "empty file" message to the user.
5. If the physical file is a nonempty file, prompt the user for a selection of display options.

6. If the option is to display to the screen, continue to read each record from the physical file, displaying the record, until an end-of-file is reached.
7. If the option is to list all records to the printer, open the printer as a text file, and then continue to read records from the physical file while writing the records to the printer until an end-of-file is reached.
8. Close the physical file and, if opened by Step 7, the printer file.

Step 5 provides a window format, shown in Figure 10.14, for choosing the option to display all records. Moving the mouse to one of the three squares and pressing the mouse button selects one of three options.


OPTION FOR LISTING ALL RECORDS:

☐ List each record to the screen.

☐ List records to the printer.

☐ Exit from this option.

Figure 10.14 Window prompting the user for the option of listing all records.

The following is a refinement of the steps for this module.

```
procedure List_Records;
{ Purpose:  This procedure requires four formal value parameters:}
{           Physical_File, of type string; Flag, of type Boolean;}
{           Message, of user-type Table; and Box, of user-type }
{           Squares. Local variables include Pascal_File, }
{           Output_File, Data, and Selection. }
begin
{ Check if the file exists or has been created. }
   if Flag then
      begin
      { Open the physical file and test if it is empty. }
         reset( Pascal_File, Physical_File );
         if eof( Pascal_File ) then
            begin
            { Report an empty file and close the physical file. }
            Report_Empty_File;
            close( Pascal_File )
         end
      else
         begin
         { Prompt user for type of listing. }
            Prompt_for_Listing( Selection, Message, Box );
         { Display records to the window or to the printer. }
            case Selection of
               1: { Display all records to the window. }
                  while not( eof( Pascal_File ) do
                  { Read a record from the Pascal file and }
                  { display to a window using }
                  { Display_Name_Address. };
               2: { Display all records to the printer. }
                  begin
                  { Open the printer as an output file. }
                     rewrite( Output_File, 'Printer:' )
                     while not( eof( Pascal_File ) do
                     { Read a record from the Pascal file }
                     { and write this record to the printer. }
                     { Page and close Output_File. }
                     end;
                        otherwise { exit from this option }
            end { case }
      close( Pascal_File )
   end
else
   Display_Nofile_Message
end; { List_Records }
```

This module requires four supporting modules: Report_Empty_File, Prompt_for_Listing, Display_Name_Address, and Display_Nofile_Message.

Figures 10.15a to 10.15e show the relationships of all the major modules with their supporting modules. Some of the modules have common steps, and further refinement of this design is left as a problem. The complete Pascal source code for Name_Address_System is included in the Program Diskette which is provided with this text. Initially all supporting modules, referred to as *utilities*, are declared with forward declarations followed by the definition of all nine major modules. Following this, all supporting modules, in alphabetic order by procedure name, are defined.

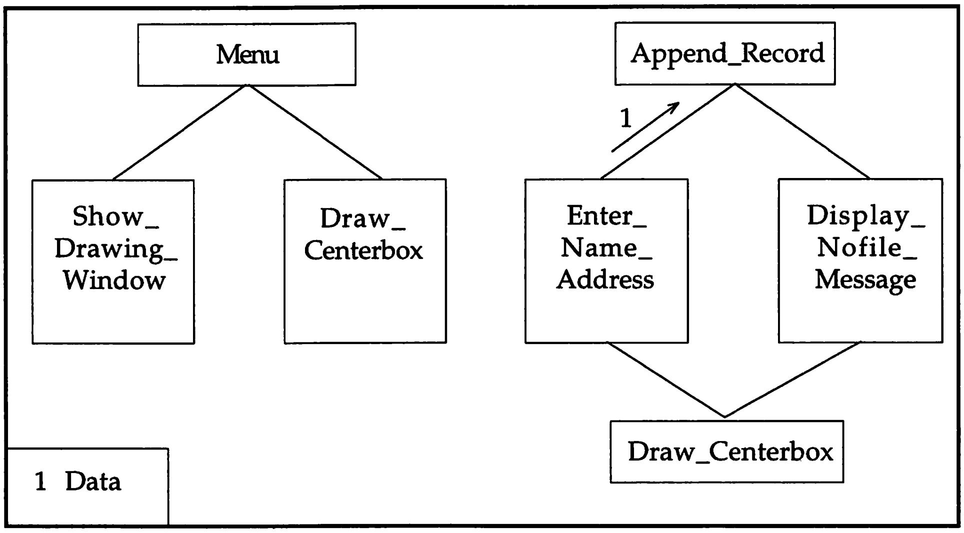


Figure 10.15a. Relationship of major and supporting modules in `Name_Address_System`.

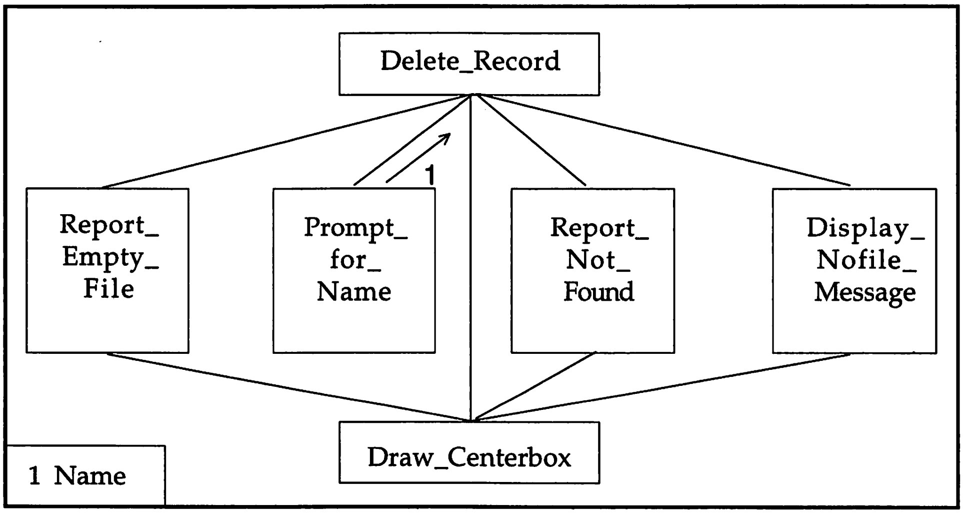


Figure 10.15b. Relationship of major and supporting modules in `Name_Address_System`.

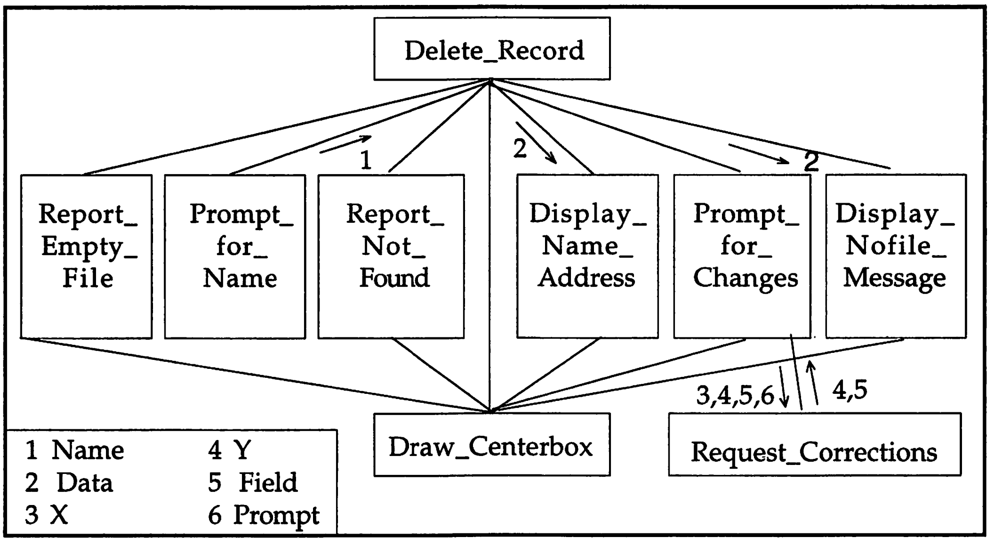


Figure 10.15c. Relationship of major and supporting modules in `Name_Address_System`.

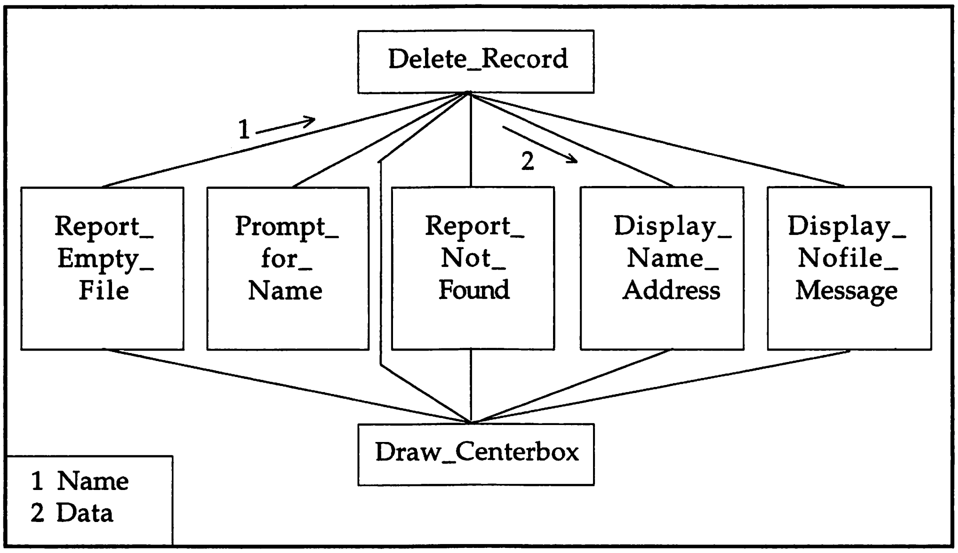


Figure 10.15d. Relationship of major and supporting modules in `Name_Address_System`.

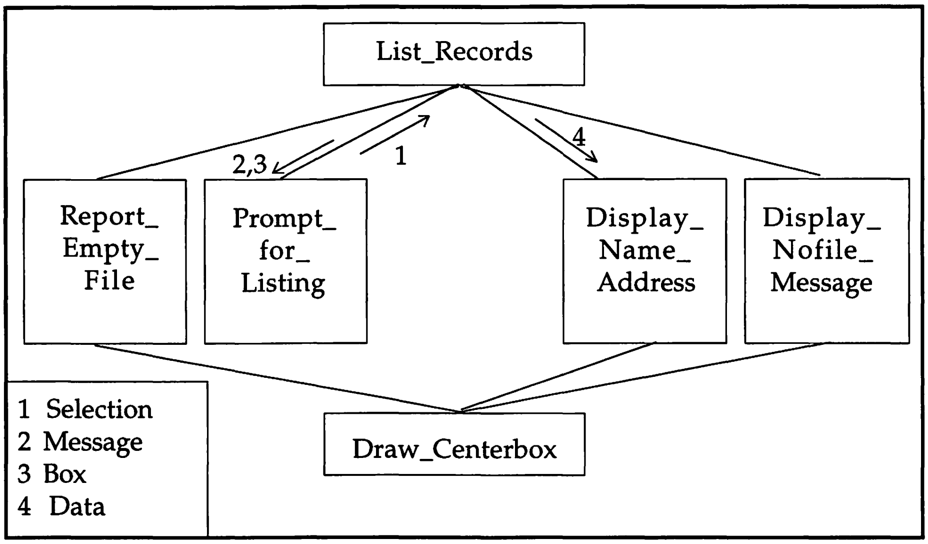


Figure 10.15e. Relationship of major and supporting modules in `Name_Address_System`.

10.12 STANDARD PASCAL VERSUS THINK PASCAL

With respect to files, what makes THINK and Macintosh Pascal different from standard Pascal? First, Macintosh and THINK Pascal allow the procedures `rewrite` and `reset` to have a second parameter for the purpose of associating a physical file name with a Pascal file variable.

In ANSI Pascal, the physical file name takes on the name of the Pascal file variable. The procedures `rewrite` and `reset` can be used to create and open a file, but they have only a single parameter, a Pascal file variable. The names of the Pascal file variables must be explicitly given in the program header where the program parameters `input` and `output` appear.

Second, the procedures `open` and `close` are not supported by standard Pascal. For any Pascal system, the scope of a file and how long it will exist are defined for the length of the program block in which the file is declared. If the file is declared local to a procedure or function, its scope is only the length of the body of the routine and no longer. A file becomes closed once the program ends execution of the program block for which a Pascal file variable has been declared. While these rules apply to THINK and Macintosh Pascal as well as standard Pascal, the procedure `close` gives us the opportunity to close a file if we no longer need it, even though the program has not finished executing the block in which a file has its scope.

The following code for the earlier program called `Reporting_Random_Numbers` is a standard Pascal program, where the name of the physical file `Data_Block` is the same as that of the Pascal file variable. If the physical file exists, it is opened by execution of the `reset` command and remains open for the length of time that the program is in execution. If it does not exist, the program halts execution when it attempts to execute `reset`.

```
program Reporting_Random_Numbers(input, output, Data_Block);
{ Purpose:  This is a standard Pascal program for testing the }
{           opening and reading of a file of real numbers. }
   type
      Number_File = file of real;
   var
      Data_Block : Number_File;
      Counter : integer;
      Number : real;
begin
{ Open the test file and position the pointer at the front of }
{ the file. }
   reset(Data_Block);
{ Read all numbers from the test file. }
   while not eof(Data_Block) do
      begin
      { Read the next number from the test file. }
         read(Data_Block, Number);
      { Display the number just read from the test file and }
      { display its value in the Text window. }
         write(output, Number : 10 : 4, ' ');
      end;
   writeln;
{ Close the test file by terminating execution of this block. }
end.
```

Both Macintosh and THINK Pascal depart from standard Pascal in allowing the name of the Pascal file variable to be listed within the parameter list of the program header. Macintosh Pascal raises a bugs window stating that an "invalid program parameter list has been found" when checking the syntax of the program. THINK Pascal compiles the program but later halts execution when an attempt is made to execute the `reset` command and reports that the "file is not open."

Third, standard Pascal does not define the concept of a random-access file. When such files are allowed, a command such as `seek` can direct a program to a file component without reading the file sequentially. By using an index number, a Pascal program can move either forward or backward through a file without resetting the file pointer with the command `reset`.

Fourth, standard Pascal does not allow `string`-type and enumerated-type values to be read from text files using `read` and `readln` procedures, nor does it allow such values to be written to text files using `write` and `writeln` procedures. While many Pascal compilers allow this extension, it is not defined as part of standard Pascal.

Last, the actual internal representation of a text file in Pascal is implementation-dependent. Although we have viewed a text file as a series of lines composed of characters, with each line terminated by an end-of-line marker, the actual internal representation varies among Pascal translators. For example, Macintosh Pascal treats a text file as a collection of lines, with each line being a series of characters terminated by the carriage-return character as an end-of-line marker. In THINK Pascal a text file is a series of file components, and each component is a structure composed of a count specifying the length of the line followed by a character string for the line. No explicit end-of-line character is stored.

What standard Pascal requires and what Macintosh and THINK Pascal follow is that an end-of-line marker in a text file must always be read as a blank space independent of the existence of the marker. This is why a Pascal program written to explicitly test for reading a return character from standard input (the keyboard acts as an input device for a text file) is never successful. If a Pascal program could read the end-of-line marker from a text file, we would not need the `Boolean` function `eoln`, and, in turn, the concept of a text file would not exist, because it would be nothing more than a file of characters. Thus, the concept of a text file as lines of text, with each line composed of its length followed by its character string and with no end-of-line marker stored is a proper model of a text file. Also, note that if a text file for any Pascal translator uses an explicit end-of-line marker, the end-of-line marker can be read by opening and reading the text file as a file of characters.

SUMMARY

Files provide the means for storing data generated during the execution of a program. Although in principle we can discuss the concept of two fundamental types of files, sequential and random, in reality Macintosh/THINK Pascal supports only sequential files. In Pascal, files are represented by a series of file components, each of which is represented by a unique component number. A special object called a *file buffer* serves as a mechanism both for pointing to a current file component and for reading from or writing to a file component. With the `seek` command, we can treat sequential files as if they were random-access files by referencing each file component with a number. The only restriction is that when a Macintosh or THINK Pascal file is being used as a random-access file, the commands `reset` and `rewrite` will not execute. Only `open` and `seek` are allowed as commands to open a file and to locate the position of the next file component. Commands such as `read`, `get`, `write`, and `put` can access the file component for either reading from or writing to the file.

Pascal files are declared through the syntax **`file of`** `data type`. File data types can include `integer`, `real`, `double`, `longint`, `extended`, `computational`, `Boolean`, or `char`, as well as structured types such as `array`, `record`, `string`, `packed array of char`, and `set`. All of these result in file components having a fixed size. Text files differ in that the file components may vary in length.

By using the commands `open`, `rewrite`, or `reset`, we can link a Pascal file with a physical file. The commands `open` and `rewrite` can create a new physical file if the file does not exist; if the physical file exists, it is opened. In either case the file pointer is directed to the beginning of the file. The command `reset` can open and link a Pascal file to an existing physical file; if the physical file does not exist, the command fails. For an existing file, the command `reset` places the file pointer at the beginning of the physical file for input to a program in execution. For an existing file, execution of the command `rewrite` erases all file components and positions the file pointer to the beginning of an empty file.

The `open` command can open either a new file or an existing file; file components can be either read from or written to during the execution of a program, but neither the command `reset` nor `rewrite` is necessary. By using commands such as `write`, `read`, `put`, or `get`, a Pascal file can be either written to or read from during the execution of a program. The commands `readln` and `writeln` apply only to text files. Text files are different from files of characters. They can contain file components of variable length that end with an end-of-line marker. This is different from a file of `char`, where each file component is of a fixed size, one character.

Two `Boolean` functions exist for testing the position within a file. One, `eof`, will test any type of file for an end-of-file marker. If the file pointer is not pointing to the end-of-file, `eof(Pascal_File)` returns a value of *false*. Once the file pointer is pointing to the end of the file, `eof(Pascal_File )` returns a value of *true*. When the value is *true*, the file buffer becomes undefined. The `Boolean` function `eoln` applies only to text files. When executed, it tests to see if the last-executed `read` has entered the last characters of an input line. If the file has reached an end-of-file marker, execution of `eoln` will fail.

REVIEW QUESTIONS

1. Define the term *file*.
2. What are the advantages of using files?
3. What three file types exist in Macintosh Pascal?
4. Briefly explain the abstract model representing the concept of a file.
5. What is the purpose of the file variable and file buffer?
6. Although a file may seem to be a one-dimensional array, how does a file component differ from an element of an array?
7. What is the special marker that terminates a file?
8. What is the only item in an empty file?
9. What three commands can be used to reposition the file pointer?
10. How is a file declared in Pascal?
11. What is meant by the term *sequential file*?
12. What is the purpose of the `open` command?
13. What is the purpose of the `close` command?
14. What is the difference between the `rewrite` and `reset` commands?
15. Are the following two sets of commands equivalent? Explain your answer.

```
open( Outfile, 'Inventory' );
rewrite( Outfile );
```

and

```
rewrite(Outfile, 'Inventory');
```

16. What two commands allow data to be written to a file?
17. What is the action of the `put` command? How does it differ from a `write` command?
18. Why must a formal parameter representing an actual parameter as a file type be a variable formal parameter?
19. Is it true that execution of the `reset` command places the physical file in a read mode?
20. What two commands allow data to be read from a file?
21. What name is given to the keyboard as a standard input file?
22. What name is given to the Text window as a standard output file?
23. What is the action of the `get` command? How does it differ from a `read` command?
24. Is the following declaration allowed in Macintosh Pascal?

```
var
   Pasfile : file of file of integer;
```

25. How can we create a temporary file that is used only during the execution of a program and has no link with any physical file?
26. What happens if you attempt to close a temporary file?
27. What is meant by the term *random file*?
28. In a Macintosh Pascal program, can a sequential file be accessed like a random file? Explain your answer.
29. What file commands can be used with random files?
30. What is the purpose of the `seek` command? What actions occur when the `seek` command is executed?
31. What command must be executed before executing a `seek` command?
32. What is the purpose of the `filepos` command?
33. How can the `seek` and `put` commands be used to update a file component?
34. How can the special function `eof` be used when accessing a file?
35. What is meant by a *text file*?
36. What is meant by the term *fixed-sized file components* versus *variable-sized file components* ?
37. Why is a text file not equivalent to a **`file of`** `char` ?
38. Whereas `writeln` and `readln` only apply to text files, can we use the commands `read` and `write` to access text files? Can you think of any examples to verify your answer?
39. Is a text file equivalent to a file of `string`? Can you think of a test that might verify your answer?
40. How can the `eoln` function be used with text files?
41. Can the `eof` function be used with all types of files?
42. Can the `eoln` function be used with nontext files?
43. How should a file stored on a particular diskette and within a folder be referenced?
44. What command can be used to reference the printer as an output file?
45. The logical file name for displaying to the text window is `Output`. What is the corresponding physical file name?
46. The logical file name for entering data from the keyboard is `Input`. What is the corresponding physical file name?
47. What is the purpose of the function `NewFileName`?
48. What is the purpose of the function `OldFileName`?
49. Is the following declaration correct?

```
var
   Pasfile : file of text;
```

50. What data types can be associated with the following declaration?

```
var
   Pasfile : file of data type;
```

51. What is wrong with the following code?

```
rewrite(Pasfile);
while not eof(Pasfile) do
```

```
   begin
      read( Pasfile, Number );
   end;
```

52. What is wrong with the following code?

```
reset(Pasfile);
while not eof(Pasfile) do
   begin
      write( Pasfile, Number );
   end;
```

53. What is the effect of executing the following code?

```
Pasfile^ := Value;
put(Pasfile);
seek(filepos(Pasfile) - 1);
Value := Pasfile^;
```

54. Execution of the statement `Page(Output)` or `Page` clears the Text window. If `Pasfile` is a text file, what occurs when the statement `Page(Pasfile)` is executed?
55. Can the commands `get` and `put` be used to input from and output to a text file instead of the commands `read`, `readln`, `write`, and `writeln`? Write a short program to test your ideas.

PROGRAMMING EXERCISES

Although not all programming exercises require you to write an algorithm, you may better understand the problem and what is required if you first write an algorithm and trace it by hand with several examples before attempting to write a Pascal program.

1. Write a program that will take a two-dimensional array of real numbers and store them in a sequential file called `Real Numbers`. Once the file has been closed, reopen it and have your program read the real numbers to verify if they have been stored properly.

2. Write a program that will create real numbers randomly and store them in a two-dimensional array called `A`. Then have your program write the numbers in array `A` to a file named `Row Vectors`, one row at a time. You will have to declare the Pascal file as a one-dimensional array, that is, as an array of `real`.

3. Write a program that will read one-dimensional arrays of real numbers stored in the file called `Row Vectors` into a two-dimensional `real` array called `B` one row at a time. Display the values of array `B`, and see if they match the random numbers generated in Exercise 2, both for value and position.

4. Using the `seek` command, write an algorithm employing the bubble sort algorithm to sort directly the file components of a sequential file without reading any file components to internal storage. *Hint* : As you saw in this chapter, you

can treat each file component as an element in a table. Test your algorithm by creating at random a file of integer numbers, sorting them, and then displaying the contents of the file.

5. Repeat Exercise 4 using the Shellsort algorithm.

6. Repeat Exercise 4 using the quicksort algorithm.

7. Write a function called `File_Size` that will count the number of file components in a sequential file. Assume that the sequential file is a nontext file.

8. Write a function called `Verify_Files`. This function takes two sequential files and checks if the corresponding file components have identical values. The value returned by this function is the `Boolean` value *true* if the files are identical, and *false* if the function fails. Assume that both files are nontext files.

9. Complete the following procedure for appending one file to another:

```
procedure Append( var File1, File2 : file-type ; var Flag :
Boolean);
```

This procedure will append a physical file linked with `File2` to a physical file linked with `File1`; in short, `File1<-- File1 + File2`. Assume that both `File1` and `File2` have been opened and are not text files. The formal parameter `Flag` is set to *true* if `Append` successfully completes execution. The following steps are required for this routine:

(a) Establish `File1` for a read mode. Report an error if this fails, and terminate execution of this routine.
(b) Establish `File2` for a read mode. Report an error if this fails, and terminate execution of this routine.
(c) Open a dummy file for temporary file storage. The dummy file must be established in a write mode.
(d) Read each file component from `File1`, and write each to the dummy file.
(e) Read each file component from `File2`, and write each to the dummy file.
(f) Establish `File1` for a write mode.
(g) Establish the dummy file for a read mode.
(h) Read each file component from the dummy file, and write each to `File1`.
(i) Close the dummy file if it is linked with an explicit physical file.

This procedure must be able to append an empty file to a nonempty file and a nonempty file to an empty file.

10. Write a procedure called `Append_Text_Files` for appending two text files. Follow the steps defined in Exercise 9.

11. Complete the following procedure for truncating a nontext file:

```
procedure Truncate( var Filevar : file-type;
   Current_Component_Position : integer; var Flag : Boolean );
```

The purpose of this routine is to truncate the physical file linked with the file variable from the current file component given by `Current_Component_Position` to the end of the file. Assume that the file variable has been opened. Normal termination of this routine leaves the file variable in a write mode. The formal parameter `Flag` is set to *true* if `Truncate` successfully completes execution.The following steps are required for this routine:

(a) Establish the file variable for a read mode. Report an error if this step fails, and terminate execution of this routine.
(b) Check if `Current_Component_Position` is less than or equal to zero. If so, report an error, and terminate execution of this routine.
(c) Open the dummy file for temporary file storage. The dummy file must be established in a write mode.
(d) Read each file component from the file variable, and write each to the dummy file until `Current_Component_Position` has been reached. If `Current_Component_Position` exceeds the end-of-file position, close the dummy file if necessary, report an error, and terminate execution of this routine.
(e) Establish the file variable for a write mode.
(f) Establish the dummy file for a read mode.
(g) Read each file component from the dummy file, and write each to the file variable.
(h) Close the dummy file if it is linked with an explicit physical file.

12. Write a program that allows the user to enter several paragraphs, storing each line of a paragraph as a line of text in a Pascal file of type `text`. Then verify that the lines have been stored by having the program read them from the text file and display them to the Text window. Be sure to use the `Boolean` functions `eoln` and `eof`.

13. Write a program that counts the number of individual characters, words, and lines of text from the text files created in Exercise 12. Have the program write this information to the Text window.

14. Write a program that can compare two text files and report if they are equivalent or nonequivalent.

15. Modify the name and address system so that all records are stored in alphabetic order by full name. This means that as a new record is added to a file, it must be merged alphabetically with the remaining records. It also requires that if a person's name is changed in an existing record, the old record must be deleted and the updated record merged with the remaining records.

16. Modify the name and address system so that several reports can be displayed to the Text window or printer. These reports should allow the following options:

    (a) Display all records having zip codes within a particular range.
    (b) Display all records having a particular city and state.
    (c) Display all records having a particular last name.

17. Professor Smith has decided to keep a small research file on his Macintosh computer. He has decided that each record entry representing a reference will have the following record format:

```
Reference = record
                Primary_Keyword: string[20];
                Secondary_Keyword: string[20];
                Third_Keyword: string[20];
                Title : string[50];
                Author: string[50];
                Publisher: string[50];
                Publication_Date: string[10];
                Abstract: string;
            end;
```

    Develop a system for Professor Smith with the following options:

    (a) Create a new research file.
    (b) Open an existing research file.
    (c) Close an existing research file.
    (d) Enter new records.
    (e) Update existing records.
    (f) Search a research file with a keyword.
    (g) Search a research file with an author's name.
    (h) Search a research file with a title.
    (i) Display one or more records of information.
    (j) Exit from the system.

18. Develop a way to accept logical search equations from input over the three keywords from Exercise 17. For example, is there a way to accept logical search equations having the following search expressions?

```
(Primary keyword) and/or (Secondary keyword) and/or (Third
   keyword)
```

19. The XYZ Warehouse Association has decided to computerize its record-keeping system by using a Pascal program on the Macintosh. You have been asked by the company president to develop and implement an inventory system for keeping records on parts. Each part is required to have the following record format:

```
Part =    record
             Part_ID : string[18];
             Description: string[50];
```

```
    Entry_Record_Date: DateTimeRec;
    Quantity_in_Stock: longint;
    Last_Access_Date: DateTimeRec;
    Back_Order_Date: DateTimeRec;
    Price_per_Part: real
end;
```

Part_ID must have the format Pddd-ccc-ddcc/19dd, where d represents a digit 0 through 9, and c represents an alphabetic letter. The sequential file must be declared as a file of Part, with each record being stored alphabetically by Part_ID. The system must have the following capabilities:

(a) Open an existing inventory file for purposes of access.
(b) Create a new inventory file.
(c) Close an existing inventory file.
(d) Add one or more records to an existing inventory file.
(e) Remove a record from an existing inventory file.
(f) Update a record from an existing inventory file.
(g) Display one or more records from an existing inventory file.
(h) Exit from the system.

20. Write a program that will read text lines entered from the keyboard and direct each line to both the Text window and the printer. Be sure to establish the printer as an output file. Your program must ask if the user wants to view lines of text as they are being printed. Pick a special character from the keyboard to terminate input when you have entered the last line of text.

21. For the file system discussed in Section 10.11, modify the design for merging a single record into the name and address file using the natural merging algorithm from Section 10.8.

22. Write a program for the Macintosh Pascal translator that can determine the type of character used as an end-of-line marker in text files. If you have access to THINK Pascal as well, see what occurs when you compile and execute this program.

Chapter 11

Manipulation of Strings

OBJECTIVES

After completing Chapter 11, you will know the following:

1. The characteristics of the `string` data type in Macintosh and THINK Pascal.
2. Standard character-string functions and procedures available in Macintosh and THINK Pascal.
3. Pattern matching and object-string replacement and how to implement these actions in a Macintosh Pascal or THINK Pascal program.
4. How to convert numeric data into character strings and character strings into numeric data.
5. How to use string functions and user-developed procedures to create a Pascal `Print Using` procedure.

11.1 STRING TYPES IN MACINTOSH PASCAL

Computers were originally designed to handle numeric computations. Over time, however, there has been an increasing demand on computers to manipulate and store non-numeric data. For example, word processing, one of the major applications of the microcomputer, is based on the manipulation of nonnumeric data. The purpose of this chapter is to demonstrate how to apply Macintosh Pascal and THINK Pascal string procedures and functions for the purpose of pattern matching and object-string replacement. We will review the `string` data type as well as some basic concepts involving character manipulation.

As we saw in Chapter 3, both Macintosh Pascal and THINK Pascal support a special data type called *`string`*. Identifiers declared as type `string` are capable of storing a sequence of characters representing the physical characteristics of a character string. As stated in Section 3.5.4, an identifier is declared to be of type `string` by the following syntactical form:

```
identifier : string[ size-attribute ];
```

Two attributes are associated with a `string` type: a dynamic length specifying the actual length of a sequence of characters assigned during execution, and a static size attribute specifying the maximum limit on the length of a string. The static size attribute is an integer in the range 1 to 255. If not explicitly declared, it assumes a default value of 255. Do not confuse string size with string length. During program execution, the Pascal software allocates for a `string` type based on the following format: 1 byte for the current value of the string length followed by a consecutive set of byte locations equal to the static size attribute. For example, the declaration

```
var
   Name  :  string[29];
```

allocates 30 bytes of storage. The first byte stores the value of the current string length, (29) and the remaining 29 bytes are for the storage of characters. Only the current value of the string length is stored, not the value of the static size attribute.

A character string can be viewed as a data object composed of a sequence of characters packed into a one-dimensional structure called an *array*. Each character occupies a byte of storage space; each byte is viewed as ordered by number from left to right. In the diagram shown in Figure 11.1, you see the declaration of a variable called `Str` and the assignment statement

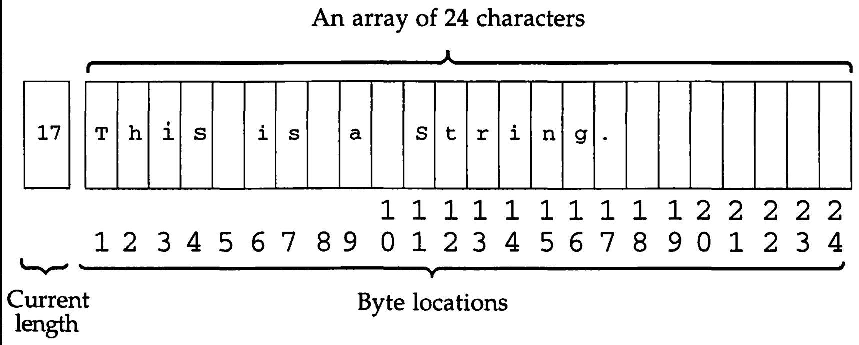
```
Str := 'This is a string.';
```

Using the model of a one-dimensional array to represent the storage of a string shows that individual bytes are ordered and can be addressed individually. For example,

`Str[4] = 's'` while `Str[14] = 'i'`.

Because the current length of the string is 17, the value of `Str[19]` cannot be specified because its index value, 19, is out of bounds. Strings longer than 17, however, can be assigned to `Str`, provided they do not exceed the static size limit of 24. In Macintosh Pascal, only the `Boolean` operators =, <> , < , > , <=, and >= can be used in writing string expressions. No other operators are allowed. When these `Boolean` operators are applied to string expressions, the results are a lexicographic ordering, according to the Macintosh character set. Any two `string` types can be compared, because all string values are type-compatible, independent of their current length. A `char` type can also be compared with a `string` type; the `char` type is treated like a string having a current length of 1. Whenever a relational expression involving two string expressions is evaluated, the result is based on the ordering relationship between the character values in corresponding positions of each of the two strings. If two strings have unequal lengths but similar leading characters, each character of the longer string that fails to correspond to a character in the shorter string is said to be ordered lexically higher. For example, the two strings `'adding'` and `'addition'` have four leading characters that are the same; `'addi'`. The string `'addition'` is ordered greater than `'adding'` because the fifth character from the left in `'adding'` is ordered less than the fifth character in `'addition'`. Two strings are lexically *equal* if (1) they are equal in current string length and (2) each contains identical characters in corresponding character positions. For example, the two strings 'A' and 'A ' are unequal, because the first is of length 1, and the second is of length 2. The second string is the character A followed by a single space.

```
{ Following is a small segment of a Pascal Program. }

program ExampleOfaString;
  var
     Str: string[24];
begin
     Str := 'This is a String.';
{ Below is a model representing the storage of the }
{ string variable called Str. }
```



```
 { Following are some facts about the string Str: }
   Str[4] = 's'
   Str[10] = ''  { blank }
   Str[14] = 'i'
   Length(Str) = 17
   Str[19] = {Unspecified; the index 19 is out }
             { of bounds because the current length }
             { of Str is 17. }
   Static size of Str = 24 bytes
             { 25 if you include the byte location}
             { for the current length. }
```

Figure 11.1 Representation of a character string as a data object in Pascal.

A space is different from a special string called a *null string*. A space represents a blank character and, when displayed, has a length of 1. A null string is a string having no length; that is, its current string length is always zero. Null strings are often useful in writing functions and procedures because it is easy to test for a null string. A null string is always equal to another null string and is always lexically lower than any non-null string.

Unfortunately, the `string` type in Macintosh and THINK Pascal lives in a state of limbo. At times it takes on the appearance of a structured type; in other instances we can treat it as a simple data type. In Macintosh and THINK Pascal it is neither a structured type nor a simple data type. In Chapter 9 we saw how to use a packed array of characters to represent a character string.

11.2 BASIC STRING PROCEDURES AND FUNCTIONS

Macintosh and THINK Pascal share a rich set of string procedures and functions, listed in Figure 11.2. Except for two of the routines, `Delete` and `Insert`, all are functions and return either `longint` (or `integer`) values or `strings`. Applications using many of these string procedures and functions are given in this section and in Sections 11.3 and 11.5.

Name	**Format**	**Result Type**	**Purpose**
Length	Length(Str)	`longint` `integer`[a]	A *function* that returns the current length of the string Str.
Pos	Pos(SubStr, Str)	`longint` `integer`[a]	A *function* that returns the position of substring SubStrin of string Str. A zero is returned if the substring is not located.
Concat	Concat(Str_1, Str_2,..., Str_n)	`string`	A *function* that concatenates the string variables Str_1 through Str_n contained within the parameter list. The resultant string may not be greater than 255.

Name	**Format**	**Result Type**	**Purpose**
Copy	Copy(Str, Index, Count)	`string`	A *function* that returns a substring from string Str beginning at the position specified by Index and extending for a specified number characters given by Count.
Delete	Delete(Str, Index, Count)		A *procedure* that changes the variable Str by deleting characters beginning at the position specified by Index and continuing until the specified number of characters given by Count have been deleted.
Omit	Omit(Str, Index, Count)	`string`	A *function* that returns a substring obtained by deleting the copy of the characters in Str beginning at the position specified by Index and extending over a specified number of characters given by Count. The value of the original string Str is not changed.
Insert	Insert(Pat, Source, Index)		A *procedure* that changes the value of the string Source by inserting another string Pat beginning at the specified position Index. The resultant string may not be greater than 255.

Name	Format	Result Type	Purpose
Include	Include(Pat, Source, Index)	`string`	A *function* that returns a string after inserting the string Pat in a copy of Source beginning at the position specified by Index. The resultant string may not be greater than 255. The values of Pat and Source are unaffected.

[a] THINK Pascal
Note: `Str`, `Source`, `SubStr`, and `Pat` are `string` type; `Index` and `Count` are of `integer` type.

Figure 11.2 Macintosh and THINK Pascal string procedures and functions.

The function `Length` returns a `longint` (or `integer` in THINK Pascal) value for the current length of the string `Str`. For example, the following Pascal program, called `Reverse_Characters`, displays a character string, reversing the order in which it was originally typed:

```
program Reverse_Characters;
{ This program displays the characters of an input string }
{ in reverse order. }

   var
      String_Length, Counter : integer;
      Input_String : string;

begin
   ShowText;
{ Prompt user to type a short string of characters. }
   write(' Type a short string of characters: ');
   readln( Input_String );
{ Compute current string length of the input string. }
   String_Length := Length(Input_String);
{ Show the characters of the input string in reverse order. }
   for Counter := String_Length downto 1 do
      write( Input_String[Counter] );
   writeln;
end.
```

The function `Pos` (for *position*) searches for a substring given by `SubStr`, starting at the left of a source string, `Str`. The type of value returned is a `longint (integer` in THINK Pascal) indicating, from the left, the position where the substring is located within the source string, Str. For example, the value of Position after execution of the following three statements will be 4, since the substring 'si' begins at the fourth position from the left in the source string 'Mississippi':

```
Source := 'Mississippi';
Substring := 'si';
Position := Pos(Substring, Source);
```

If the substring is not contained within the source, the value returned is zero.

The function Concat provides the only means in Macintosh Pascal for joining (concatenating) one or more substrings. It returns as a value a string type having a size attribute of 255. The strings are concatenated from left to right in the order in which they are given. For example, the following statements, when executed, yield the string 'THINK Pascal is ideal for string manipulation ':

```
String1 := 'THINK Pascal ';
String2 := 'is ideal for ';
String3 := 'string manipulation ';
Result := Concat(String1, String2, String3);
```

Notice that the strings can be concatenated from right to left by reversing the order of the parameter list when calling on the function Concat. The following statement yields the result 'string manipulation is ideal for THINK Pascal ':

```
Result := Concat(String3, String2, String1);
```

The function Concat will fail if the string returned by joining the substrings exceeds 255 characters.

The function Copy copies a fixed number of characters from a source string, starting at a position given by Index and extending for a fixed number of characters given by Count. For example, the following procedure, Balance_String, will add to the right of Input_String a sequence of zero or more asterisks. Output_String will always be 15 characters in length.

```
procedure Balance_String( Input_String : string;
   var Output_String : string );

{ Purpose:  This procedure balances the right side of an input }
{           string with asterisks. The formal parameter }
{           Input_String is truncated to 15 characters from the }
{           left. }

   const
      Asterisks = '***************';
   var
      Appended_Asterisks : string[15];

begin
{ Copy only the first 15 characters of Input_String. }
   Input_String := Copy(Input_String,1,15);
{ Form a string of zero or more asterisks. }
```

```
   Appended_Asterisks := Copy( Asterisks, 1,15 -
                                 Length(Input_String) );
{ Append to the right of Input_String zero or more asterisks. }
   Output_String := Concat( Input_String, Appended_Asterisks);
end;
```

Special conditions for the values of parameters `Index` and `Count` can affect the value returned by the function `Copy`:

1. If `Count` is negative, the value returned by `Copy` is a null string.
2. If `Index` is less than 1, or if `(Index + Count) > Length (Source)`, or both, character positions outside the range from 1 to `Length(Source)` are referenced and copied as null characters along with those characters that lie within range.

As an example of null characters being copied, execution of the expression `Copy('Mississippi', -4, 7 )` will return a string value of `'Mi'`. As Figure 11.3 illustrates, this comes from the function `Copy` using an initial index position –4 with the seven characters to the right being copied from that position. Five of these are null characters located at character positions –4, –3, –2, –1, and 0. The remaining two character positions to be copied are from the beginning of the word `Mississippi`.

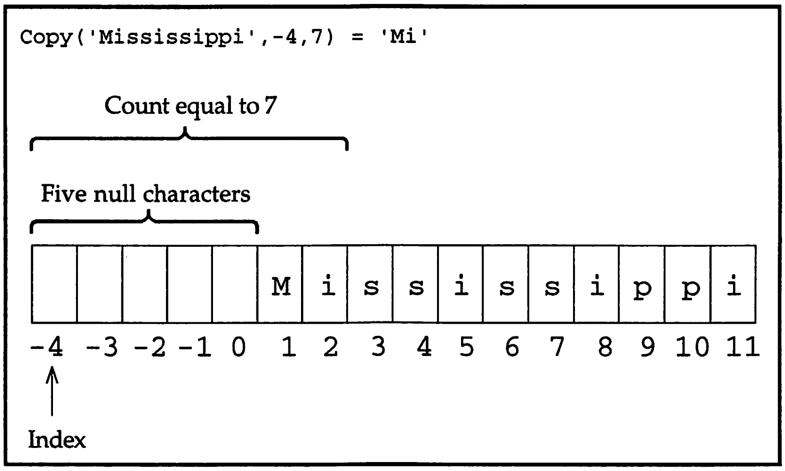


Figure 11.3 Why the value of `Copy('Mississippi',-4,7)` is `'Mi'`.

As an additional example of the use of the function `Copy`, consider defining a new function called `Copy_Right`. Its purpose is to copy a substring from a source string starting at an index position referenced from the right end rather than the left end of the source string. Rather than copying the characters to the right of the index position, this new function must copy the characters to the left of the index position. As a further illustration of this function, consider the relationship between the left and right index positions shown in Figure 11.4. To write the steps necessary to complete `Copy_Right`,we need only two functions: `Length` and `Copy`. `Length` is used in an expression for converting the value of the right index into a left index position for the function Copy. Copy is used to copy a sequence of characters starting at Position_Index and extending for a specified number of positions to the right.

Here is a listing for this function:

```
function Copy_Right( Source : string; Right_Index, Count :
                         integer ) : string;

{ Purpose:  This function copies characters, starting at a }
{           position from the right given by Right_Index, and }
{           copies an extended number of characters to the left }
{           given by Count.}
   var
      Position_Index : integer;

begin
   Position_Index := Length(Source)-(Right_Index + Count - 1) + 1;
   Copy_Right := Copy( Source, Position_Index, Count );
end;
```

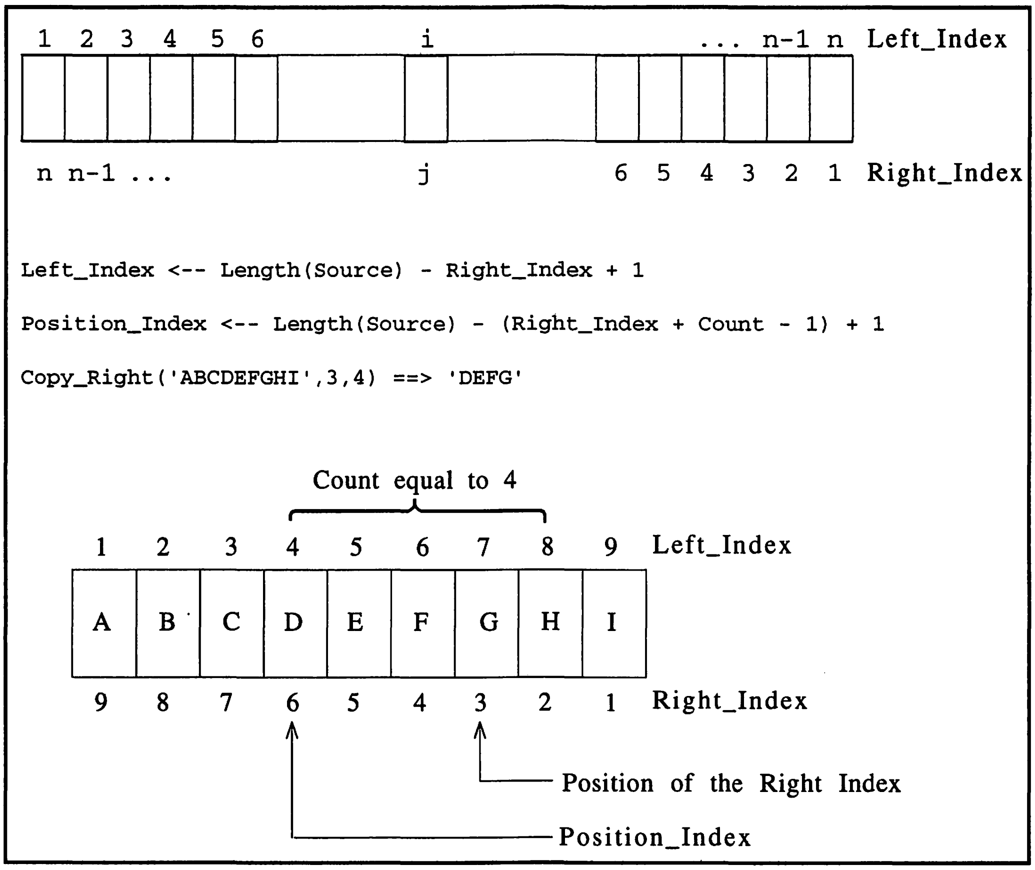
This function has properties similar to Copy. For example, if Count is negative, the function Copy_Right returns a null string. If the value of Index is less than 1, character positions outside the range of

```
Length(Source) - Right_Index + 1 to Length(Source)
```

are copied as null characters in addition to those characters that do lie within the range.

The procedure Delete deletes characters, starting at a position given by Index and continues to character position Index + Count. For example, the following list of statements removes from the source string all patterns of 'si':

```
Source := 'Mississippi';
Pattern := 'si';
Position := Pos(Pattern, Source);
while Position > 0 do
   begin
      Delete(Source, Position, 2);
      Position := Pos(Pattern, Source)
   end;
```



1 2 3 4 5 6 i ... n-1 n Left_Index
n n-1 ... j 6 5 4 3 2 1 Right_Index
Left_Index <-- Length(Source) - Right_Index + 1
Position_Index <-- Length(Source) - (Right_Index + Count - 1) + 1
Copy_Right('ABCDEFGHI',3,4) ==> 'DEFG'
Count equal to 4
1 2 3 4 5 6 7 8 9 Left_Index
A B C D E F G H I
9 8 7 6 5 4 3 2 1 Right_Index
Position of the Right Index
Position_Index

Figure 11.4 Character positions with reference to the right of a character string.

If the source string is to be left unchanged during the action of deleting characters, we can use the function `Omit` to perform the same steps as the procedure `Delete`. The following is a modification of the earlier statements that leaves the source string unchanged:

```
Source := 'Mississippi';
Pattern := 'si';
Substring := Source;
Position := Pos(Pattern, Substring);
while Position > 0 do
   begin
      Substring := Omit(Substring, Position, 2);
      Position := Pos(Pattern, Substring)
   end;
```

Special conditions for the values of the parameters `Index` and `Count` can affect the values returned by the procedure `Delete` and the function `Omit`:

If `Index` is less than 1, or if (`Index + Count`) > `Length(Source)`, or both, character positions outside the range of 1 to `Length(Source)` are referenced. Only those characters that lie within range are deleted.

The procedure `Insert` performs the opposite steps of `Delete` and `Omit`. In executing the procedure `Insert`, the source string is changed by inserting the substring of `Pat` at the position `Index` of `Source`. The following statements provide an example of `Insert` that replaces each occurrence of the pattern `'si'` with `'--'` in the source string:

```
Source := 'Mississippi';
Pattern := 'si';
Dashes := '--';
Position := Pos(Pattern, Source);
while Position > 0 do
   begin
      Delete( Source, Position, 2);
      Insert( Dashes, Source, Position);
      Position := Pos(Pattern, Source);
   end;
```

The function `Include` performs the same steps as `Insert`, except that it leaves the source unchanged. Special conditions for values of the parameter `Index` can affect the values returned by the procedure `Insert` or the function `Include`:

1. If `Index` is less than 1, the string `Pat` is appended to the left of the source.
2. If `Index > Length(Source)`, the string `Pat` is appended to the right of the source.

In either case `Insert` and `Include` will successfully execute *only* if the resulting string has a string length less than or equal to 255. Consider an example that applies several of these string functions, using a new procedure designed to replace all occurrences of double dashes in a string with a single dash. Any double dashes that occur while a double dash is being replaced must also be replaced with a single dash. The steps for executing this new procedure are quite simple:

1. Locate the first set of double dashes, using the function `Pos`.
2. If double dashes exist, perform the following steps:
   (a) Delete the first set of double dashes from the left.
   (b) Insert a single dash where the double dashes were deleted.
   (c) Locate the next set of double dashes from the left and return to step(a) to repeat this process.

The procedure for executing these steps is as follows:

```
procedure Replace_Double_Dashes( var Source : string );
{ Purpose:  This procedure replaces double dashes in the source }
{           string with single dashes. }
   const
```

```
      Dashes = '--';
      Dash = '-';
   var
      Position_Double_Dashes : integer;
begin
{ Locate the first set of double dashes. }
   Position_Double_Dashes := Pos(Dashes, Source);
   while Position_Double_Dashes > 0 do
      begin
      { Delete the first set of double dashes from the left. }
         Delete( Source, Position_Double_Dashes,2 );
      { Insert single dash where the double dashes were deleted. }
         Insert( Dash, Source, Position_Double_Dashes );
      { Determine the next position of double dashes. }
         Position_Double_Dashes := Pos( Dashes, Source );
      end;
end;
```

To avoid iterative executions with a loop, this procedure can be written recursively. In writing a recursive procedure one must first consider the trivial case; that is, the action that stops us from having to replace double dashes with a single dash when the source string has no double dashes. If it does have double dashes, the following steps are performed:

1. Delete the first set of double dashes from the left of the source string.
2. Insert a single dash in the source string where the double dashes were deleted.
3. Copy the sequence of characters from the left of the source string up to but not including the position from where the double dashes were deleted, and assign this to a string called `Front`. `Front` has no double dashes and therefore requires no further examination.
4. Copy the sequence of characters from where the double dashes were replaced to the end of the source string, and assign this to `Rear`.
5. Execute these steps recursively on the string called `Rear`.
6. After removing the double dashes from `Rear`, concatenate `Front` with `Rear`, and assign to `Source`.

Here is our revised procedure:

```
procedure Replace_Double_Dashes_Revised( var Source : string );
{ Purpose:  This procedure replaces double dashes in the source }
{           string with single dashes. }
   const
      Dashes = '--';
      Dash = '-';
   var
      Front, Rear : string;
      Position_Double_Dashes : integer;
begin
```

```
{ Locate the first set of double dashes. }
   Position_Double_Dashes := Pos(Dashes, Source);
   if  Position_Double_Dashes > 0  then
      begin
      { Delete the first set of double dashes. }
         Delete( Source, Position_Double_Dashes,2 );
      { Insert a single dash where the double dashes }
      { were deleted. }
         Insert( Dash, Source, Position_Double_Dashes );
      { Copy front substring having no double dashes. }
         Front := Copy(Source, 1, Position_Double_Dashes - 1 );
      { Copy rear substring, which may still contain double }
      { dashes. }
         Rear := Copy( Source, Position_Double_Dashes,
               Length(Source) - Position_Double_Dashes + 1);
      { Replace double dashes in the rear substring. }
         Replace_Double_Dashes_Revised( Rear );
      { Concatenate the two substrings Front and Rear. }
         Source := Concat(Front, Rear)
      end;
end;
```

Notice that the formal parameter Source is of variable type, because the value of Source must change when either the procedure Replace_Double_Dashes or Replace_Double_Dashes_Revised is executed. If Source were a value parameter, it would leave the original string unchanged.

11.3 PATTERN MATCHING AND OBJECT-STRING REPLACEMENT

This section introduces the concept of pattern matching and replacement. For simplicity, we discuss only three basic string functions: Length, Copy, and Concat. (With these, all other string procedures and functions can be defined.) Assume that we are given a string represented by an identifier called First_String and another string called Second_String. How can we take the value of Second_String as a pattern and search through the value of First_String as an object to see if the pattern is contained within the object? For example, suppose you have declared and defined two strings as follows:

```
. . .
First_String, Second_String : string[80];
. . .
First_String := 'These are the times that try mens souls';
{ Note: The correct possessive, men's, was avoided to prevent }
{ a syntax error caused by a single quote. }
Second_String := 'times';
. . .
```

The problem we are considering is how to determine if the pattern given by `'times'` is contained in the object, `'These are the times that try mens souls'`. The following steps provide a solution:

1. Assume there exists a pointer P that initially points to the leftmost character of the string `First_String`. That is, P is initially assigned to byte 1. In the example, P points to the character T in `First_String`. In addition, we create a variable called `Remainder`, which is initially given the value of `Length(First_String)`.

2. Compare `Length(First_String)`, or whatever portion of it remains, with `Length(Second_String)`. If `Length(Second_String) > Remainder`, it is impossible to have a match, and execution of the algorithm must end. If `(Remainder ≥ Length (Second_String))` is *true*, continue execution of Steps 3 through 5.

3. Starting at position P, use the function `Copy` to extract the first `Length(Second_String)` characters from `First_String`. Assign this substring to the variable `Dummy`. Note that the function `Copy` leaves the value of `First_String` unchanged.

4. Compare `Dummy` and `Second_String` to see if they are equal. If so, the value of the pointer P indicates the position at which the matching string begins, and execution can end. Otherwise, continue the search by executing the following step.

5. Increment P, and decrement `Remainder` by 1. Continue searching `First_String` by returning to Step 2.

Let us refine these steps in the following procedure, `String_Match`. In this procedure, `Object` and `Pattern` represent two formal parameters of value type, and `Position` is of variable type. If the pattern is contained within the object, the value returned through `Position` represents where the pattern begins in the object string. The remaining variables `Dummy`, `Remainder`, `P`, and `Continue_Search` are local.

```
procedure String_Match ;
begin
{ Initialize special variables before beginning search
   of the pattern. }
      Continue_Search <-- true;
      P <-- 1;
      Position <-- 0;
      Remainder <-- Length(Object);
      repeat {searching for pattern }
      { Check if the length of the pattern exceeds the
         remaining length of the object.}
         if Length(Pattern) > Remainder then
                     Continue_Search <-- false
         else
            begin
```

```
                Dummy<--Copy(Object,P,Length(Pattern));
             { Check if the substring Dummy is equal to
                the pattern. }
                if Dummy = Pattern then
                   begin
                      Position <-- P;
                      Continue_Search <-- false
                   end
                else
                   begin
                      P <-- P + 1;
                      Remainder <-- Remainder - 1
                   end
             end;
      until not Continue_Search
end;   { String_Match }
```

The Macintosh Pascal program Match_Strings demonstrates this basic concept of pattern matching.

```
program Match_Strings;
{ Purpose:  This program accepts an object string and a pattern }
{           string from the keyboard and determines if the }
{           pattern is within the object string. }
   uses     {Remove the uses clause for THINK Pascal }
      QuickDraw1;
   type
      Str = string[80];
   var
      First_String, Second_String : Str;
      Answer : string[1];
      Position_Pattern : integer;
{ ********************************************************* }

   procedure Text_Window;
   { This procedure sets the boundaries for the Text window. }
      var
         Window : Rect;
   begin
      HideAll;
      SetRect(Window, 0, 40, 510, 340);
      SetTextRect(Window);
      ShowText;
   end;
{ ********************************************************* }

   procedure Prompt_and_Enter( Message : string;
                               var Input_String : string);
   { This procedure prompts the user with a message and accepts }
   { an input string typed at the keyboard. }
   begin
```

```
      writeln(Message);
      readln(Input_String);
      writeln
   end;
{ ******************************************************* }

   procedure String_Match (Object, Pattern : Str;
                                 var Position : integer);
   { This procedure locates the position of a pattern within an }
   { object string. }
      var
         Remainder, P : integer;
         Dummy : Str;
         Continue_Search : Boolean;
   begin
   { Initialize special variables before beginning search of }
   { the pattern. }
      Continue_Search := true;
      P := 1;
      Position := 0;
      Remainder := Length(Object);
      repeat {searching for the pattern }
      { Check if the length of the pattern exceeds the }
      { remaining length of the object.}
         if Length(Pattern) > Remainder then
            Continue_Search := false
         else
            begin
               Dummy := Copy(Object, P, Length(Pattern));
               { Check if substring Dummy is equal to pattern. }
                  if Dummy = Pattern then
                     begin
                        Position := P;
                        Continue_Search := false
                     end
                  else
                     begin
                        P := P + 1;
                        Remainder := Remainder - 1
                     end
            end;
      until not Continue_Search
   end;
{ ******************************************************* }

begin { Body of the main program. }
{ Hide all windows and open the Text window.  }
   Text_Window;
   repeat
   { Enter the object and pattern from the keyboard. }
      Page;
      Prompt_and_Enter(' Enter the object string: ',
                         First_String);
      Prompt_and_Enter(' Enter the pattern string: ',
                         Second_String);
   { Search First_String for the pattern given by Second_String. }
      String_Match(First_String, Second_String, Position_Pattern);
   { Report on the success of the search for the object string. }
      writeln(' Pattern:  ', Second_String);
      writeln('   Object: ', First_String);
      if Position_Pattern > 0 then
         writeln(' Pattern begins at position ',
                     Position_Pattern : 2)
      else
         writeln(' Pattern does not exist in object string. ');
      writeln;
   { Prompt user to continue. }
      Prompt_and_Enter(' Compare other strings ( Y / N ) ? ',
                         Answer);
   until (Answer = 'N') or (Answer = 'n')
end.
```

To execute this program under THINK Pascal, you must make two changes. First, you must remove the **uses** clause. Second, you must use a different identifier for `Object`. The reason for the latter change is simple: **`Object`** is a reserved word in THINK Pascal. Try changing all occurrences of the identifier to `ObjectStr`. How can we use the procedure `String_Match` to perform the basic operation of pattern matching and replacement? The following Macintosh Pascal procedure, called `Replacement`, demonstrates this concept.

```
procedure Replacement (var Object : Str; Pattern, Substring :
                            Str);
{ Purpose:  This procedure replaces the pattern in the object }
{           string with a substring. }
   var
      Left_String, Remaining_String : Str;
      Position : integer;

begin
{ Determine if the pattern is within the object string. }
   String_Match(Object, Pattern, Position);
   if Position > 0 then
      begin
      { Remove the left substring in Object just before }
      { the pattern. }
         Left_String := Copy(Object, 1, Position - 1);
      { Concatenate Substring to the right of Left_String. }
         Left_String := Concat(Left_String, Substring);
      { Determine the position of the remaining right substring }
      { in Object. }
         Position := Position + Length(Pattern);
      { Copy the remaining string in Object. }
         Remaining_String := Copy(Object, Position,
                                  Length(Object) - Position + 1);
      { Assign to Object Left_String concatenated on the right }
      { with Remaining_String. }
         Object := Concat(Left_String, Remaining_String)
      end
end;
```

This procedure first calls on String_Match to see if the pattern exists in the object string. If so, the value of Position is greater than zero, and the procedure Replacement proceeds to extract the substring left of the pattern in the object. Next, it concatenates this part with the string being substituted, called Substring. The remainder of the substring to the right of the pattern is then copied from the object and concatenated on the right of the variable Left_String. To execute this procedure under THINK Pascal, you must change the identifier Object.

Through the application of only the three string functions Length, Copy, and Concat, the actions of procedures String_Match and Replacement represent the definitions of the function Pos and the procedures Delete and Insert. Using these added routines reduces the executable steps in the procedures String_Match and Replacement. As the following listings show, the body of the procedure String_Match is reduced to one executable statement, and the body of Replacement requires only three executable statements.

```
procedure String_Match( Object, Pattern : Str; var Position :
integer);

{ Purpose:  This procedure locates the position of a pattern }
{           within an object string. }

begin
   Position := Pos(Pattern, Object)
end;

procedure Replacement(var Object : Str; Pattern, Substring :
                          Str);
{ Purpose:  This procedure replaces the pattern in the object }
{           string  with a substring. }
   var
   Position : integer;

begin
{ Determine if the pattern is within the object string. }
   String_Match(Object, Pattern, Position);
{ Delete the pattern from the object string. }
   Delete(Object, Position, Length(Pattern));
{ Insert the replacement string. }
   Insert(Substring, Object, Position)
end;
```

11.4 SOME MISCELLANEOUS STRING ROUTINES FOR MACINTOSH AND THINK PASCAL

Macintosh Pascal supports two miscellaneous string routines for converting numeric data into strings and string data into numeric values. The first, called `StringOf`, is a function that can convert a sequence of numeric and nonnumeric data into a single string of characters. It takes as actual parameters a list of expressions that are equivalent to the parameters used in a `write` or `writeln` statement. The value returned is a `string` type containing the string representations for the values of the parameters. Here is the syntax required to call the function `StringOf`:

identifier := StringOf(parm_1, parm_2, parm_3, . . . , parm_n);

The expressions parm_1, . . . , parm_n can contain the specifications of field width as well as a specification for the number of decimal places. Each expression must either be a `char` type, `integer` type (`integer`, `subrange` of an `integer`, or `longint`), `real` type (`real`, `double`, `extended`, or `computational`), `string` type, `enumerated` type (`Boolean` or a `subrange` of an `enumerated` type), or of type `packed array of char` (discussed in Chapter 9).

The procedure `ReadString` reverses the action of `StringOf`. This procedure, which is similar to `read`, accepts its text from a string parameter instead of from the keyboard. Use the following syntax to call the procedure `ReadString`:

ReadString(Source_String, var_1, var_2, . . . , var_n);

Here `Source_String` is a `string` type, while the variables var_1, var_2, . . . , var_n must be declared as one of the following: `char` type, `integer` type (`integer`, `subrange` of an `integer`, or `longint`), `real` type (`real`, `double`, `extended`, or `computational`), `enumerated` type (`Boolean` or a `subrange` of an `enumerated` type), or `string` type. When executed, the procedure `ReadString` reads text from `Source_String` in the same way as a `read` statement. The substrings read from `Source_String` are converted into their proper internal machine formats and assigned to their corresponding variables. An execution error occurs if `ReadString` tries to read characters beyond the end of the source string. The following short program shows the results of executing `StringOf` and `ReadString`.

```
program Miscellaneous_Routines;
   uses    {Remove the uses clause for THINK Pascal }
      QuickDraw1;
   type
      Days = (Monday, Tuesday, Wednesday);
   var
      Receiving_String : string;
      Source_Str : string[20];
      Number : integer;
      Value : real;
      Character : char;
      WeekDay : Days;
```

```
begin
{ Show only the Text window. }
   HideAll;
   ShowText;
{ Initialize the variables declared above. }
   Source_Str := ' Test of StringOf ';
   Number := 12345;
   Value := 3.142356;
   Character := '@';
   WeekDay := Monday;
{ Assign a value using StringOf to Receiving_String. }
   Receiving_String := StringOf(Number:5, Value :10:5,
                               Character, WeekDay, Source_Str);
   writeln;
   writeln(' Characters in Receiving_String: ');
   writeln(Receiving_String);
   writeln;
{ Reverse the process of StringOf by using ReadString. }
   ReadString(Receiving_String, Number, Value, Character,
                  WeekDay, Source_Str);
{ Display each of the five different values. }
   writeln(' Results from executing procedure ReadString: ');
   writeln(' Number: ', Number);
   writeln(' Value: ', Value : 10 : 5);
   writeln(' Character: ', Character);
   writeln(' WeekDay: ', WeekDay);
   writeln(' Source_Str: ', Source_Str);
end.
```

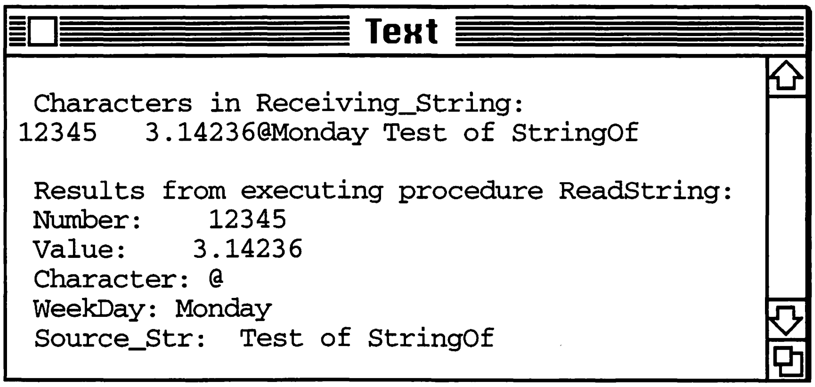
Figure 11.5 shows the output from the program `Miscellaneous_Routines`. Be careful when using `string` types with `ReadString`. Although the expression

```
Receiving_String := StringOf(Source_Str, Number:5, Value :10:5,
                             Character, WeekDay);
```

will execute successfully, the statement

```
ReadString( Receiving_String, Source_Str, Number, Value,
            Character, WeekDay);
```

will cause an error during execution, indicating that we are attempting to read a sequence of characters for `Source_Str` that is larger than the static limit of 20. Simply said, `ReadString` attempts to read all of the characters of `Receiving_String` for `Source_String`.



```
 Characters in Receiving_String:
12345   3.14236@Monday Test of StringOf

 Results from executing procedure ReadString:
 Number:    12345
 Value:    3.14236
 Character: @
 WeekDay: Monday
 Source_Str:  Test of StringOf
```

Figure 11.5 Sample output from a program using the function `StringOf` and the procedure `ReadString`.

11.5 EXAMPLE: EMULATING A `PRINT USING` STATEMENT

The `Print Using` statement allows control over the appearance and location of numeric information as it is displayed to the Text window. This is important if we need to display the information in a right-justified format, that is, if the information needs to appear on the right side of a print field rather than on the left side. Another situation that calls for a `Print Using` statement is aligning a column of numbers. This is especially important in assuring that the decimal points in a column of numbers will align properly when the numbers are displayed.

The `Print Using` statement relies on a user format specified by placing symbols in what is called an *edit field*. For example, if the format `##,###.##` is placed in the edit field and the number to be formatted is 56123.456, the result should be `56,123.46`. If the number to be formatted is 234122, the result should be `234,122.00`.

Macintosh Pascal does not provide a `Print Using` command, but the following pages demonstrate how to write such a procedure. The procedure, called `Print_Using`, allows the user the option of displaying data using the following formats:

1. A fixed number of digits, where the symbol # is used to specify a digit position. For example, the format #### will result in the string 34.3 being displayed as ' 34'.

2. Fixing the location of the decimal point by inserting a period in the edit field. For example, ###.# will result in the string −67.83 being displayed as '– 67.8'.

3. Inclusion of the dollar sign in the display. For example, the format $###.## will cause the string 204.576 to be displayed as '$204.58'.

4. Inclusion of commas in the displayed number. For example, the format ##,###.## will cause the string 1234.56 to be displayed as ' 1,234.56', and the string 12.3 will be displayed as '   12.30'.

This `Print_Using` procedure assumes the following rules regarding the specification of the edit field:

1. The number of spaces reserved for digits in the edit field will be specified by the amount symbol, #.

2. The number of digits on each side of the decimal point will be determined by the placement of a period between two amount symbols in the edit field. The appearance of more than one period in the edit field will produce the error message "Too many periods."

3. Any edit field beginning with the symbol $ will cause the dollar sign to be displayed before the first digit in the number. A dollar sign in any other position in the edit field will result in the error message "Improper use of the dollar sign."

4. A comma placed in the edit field will result in the digits being displayed with a comma located before every third digit to the left of the decimal. If there is no digit to the left of the comma, the comma will not be displayed.

5. Any other characters showing in the edit field will produce the error message "Illegal character in the edit field."

6. If the number to be displayed is larger than the edit field, the error message "The number exceeds the edit field" will be displayed.

The key elements in the algorithm for the `Print_Using` procedure follow:

```
Algorithm Print_Using;
{ This algorithm will begin by measuring the length of the edit
  field. If this length is greater than zero, the algorithm
  checks for any improper characters within the edit field. As
  the edit field is being examined character by character, two
  counters are employed: Leftcount and Rightcount. Leftcount
  represents the number of left digit positions and Rightcount
  represents the number of right digit positions. Additional
  markers such as Markcomma, Markdollar, and Periodcount are
  employed  to determine when a comma, dollar sign, or period is
  encountered. If any error should appear, the variable Error is
  set to true. }

{ Initialize the flags, counters, and a temporary variable called
  Num. }

Error <-- false;
Markcomma  <-- false;
Negative <-- false;
Markdollar <-- false
Leftcount  <-- 0;
Rightcount  <--  0;
```

```
Periodcount <-- 0
Num <-- Number

{ Check if Number is negative; if so, set flag Negative and
  initialize Num as the absolute value of Number. }

if (Number < 0) then
   begin
      Negative <-- true;
      Num <-- abs(Number);
   end; {if}

{ Measure the length of the edit field and check if the length is
  greater than zero. }

Len <-- Length(Edit);
if (Len > 0) then
   begin
   { Initialize two additional counters.  }
      Count <-- 0;
      Pointer <-- 1;
   { Examine each character of the edit field for commas, dollar
      signs, and periods, and count each decimal place as the edit
      field is scanned. }

      while (Pointer <= Len) do
         begin
            Character <-- Copy(Edit, Pointer, 1);
         { Check the character picked from the edit field. }
            case ( Character ) of
               "$"  :  { If this is the leftmost character,  then
                          set Markdollar flag; else set the Error
                          flag, since the character is not in its
                          proper place, and display error message
                          1. }

                  if (Pointer = 1) then
                     Markdollar <-- true;
                  else
                     begin
                        Error <-- true;
                        write ({ error message 1})
                     end {if};

               "#"  :  { Count this digit position. }
                  Count <-- Count + 1;

               "."  : { Increment the current value of Count and
                        reset Count to keep track of digit
                        positions to the right of the decimal
                        point. }
                  Periodcount <-- Periodcount + 1;
                  if (Periodcount > 1) then
                     begin
                        Error <-- true;
                        write( { error message 2 })
                     end
                  else
                     begin
                        Leftcount <-- Count;
                        Count <-- 0
                     end; { endif }

               "," : { Set the comma flag. }
                  Markcomma <-- true;

               otherwise  { There is an illegal character in
                              the edit field. Set error flag
                              and report error message 3. }
                  begin
                     Error <-- true;
                     write( {error message 3 } )
                  end
            end { case };

      { Increment character pointer for the next character in
            the edit field. }
         Pointer <-- Pointer + 1;
      end { while };

   { If Periodcount is nonzero, then Count represents only right
      digit positions; else this counter represents only left
      digit positions. }
      if (Periodcount > 0) then
         Rightcount <-- Count;
      else
         Leftcount <-- Count
   { endif }
end { then clause for Len > 0 }

else  { Edit field is empty. Set error flag and report error
         message 4. }
   begin
      Error <-- true;
      write ( { error message 4 } )
   end;
{ endif }

{ Next convert the number to a string and check to see if it is
   larger than the edit field; if so, set the error flag and
   display error message 5. With the digits to the right of the
   decimal point being shifted to the left of the decimal point,
   convert the number from a real to an integer.  In the program a
   function called Power is used to raise 10 to the appropriate
   power. }

   Num2 <-- trunc(( Num * $10^{Rightcount}$ ) + 0.5)) ;

{ Now convert the integer Num2 to a string; the function
   Remove_Blanks removes leading blanks from the string. }
   Numstring <-- Remove_Blanks(Num2);

{ Check if the number of digits to the left of the decimal point,
   including space for a negative sign,  exceeds the number of
   digits allowed in the edit field. }
   Len1 <-- Length(Numstring);

{ Len1 represents the number of digits in Numstring. }
   Len2 <-- Len1 + ord(Negative);

{ Len2 is the value of Len1 plus one for the digit
   position of a minus sign. }
   if (Len2 > Rightcount + Leftcount) then
      begin
         Error <-- true;
         write( { error message 5 } )
      end; { endif }

{ If an error exists at this point,  display %  followed by the
   value of the number;  else complete the remainder of the steps
   in this  algorithm. }
   if ( Error ) then
      write(  '%', Number )
   else  { continue with this algorithm }
      begin
         {  If no error exists in the edit field, the  steps
            necessary to produce the format representation of
            Number can now be executed.  The string Numstring is
            divided  into two substrings:  Leftfield represents
            the digits to the left of the decimal point and, if
            necessary,  a dollar sign contains the digits to the
            right of the decimal point.  If a decimal point is
            required,  it will be inserted between these two
            fields. }
        { Extract from the number called Num2 the leftmost and
            rightmost digits adjacent to the  decimal point.
            Compose the field to the right of the decimal point if
            Rightcount > 0. }
         Leftvalue <-- trunc( Num2 / ( $10^{Rightcount}$ ));
         Rightvalue <-- trunc( Num2 - ( Leftvalue *
                                        $10^{Rightcount}$) + 0.5 );
         { Compose the string Rightfield and check if it needs to
            be padded with zeros to the left of this string. }
        if (Rightcount > 0) then
           begin
              Rightfield <-- Remove_Blanks(Rightvalue);
              Number_Rightdigits <-- Length(Rightfield);
              if (Rightcount > Number_Rightdigits) then
                 for Pointer := 1 to (Rightcount -
                                 Number_Rightdigits) do
                    Rightfield <-- Concat('0', Rightfield);
              { endif }
           end  { then clause }
        else { None of the edit field is reserved for the right
                 of the decimal. Set Rightfield equal to a null
                 string. }
           Rightfield <-- '';
        { endif }

{ Initialize Leftfield as a null string  and the number of digits
   in the left field as 0. }
   Leftfield <-- '';
   Number_Leftdigits <-- 0;

{ If Leftcount is positive, then check if commas are required to
   the left of the decimal point.  Two temporary strings store
   the string representation of Leftvalue and the length of the
   temporary string variable. }
   if ( Leftcount > 0 ) then
      begin
         Tempfield <-- Remove_Blanks(Leftvalue);
         Tempvalue <-- Tempfield;
         Number_Leftdigits <-- Length(Tempfield);
      end { endif }

{ Check if commas are required and, if so,  insert them. }
   if ( Markcomma ) then
      begin
         Y <-- Number_Leftdigits;
         while Y < 3 do
            begin
            { Extract the three rightmost digits in Tempfield. }
               Threechar <-- Copy(Tempfield, Y-2, 3);
            { Compose Leftfield by inserting a comma before
               Threechar. }
               Leftfield <-- Concat(',', Threechar, Leftfield);
               Y <-- Y - 3
            { Extract the remaining left digits from Tempstring. }
               Tempfield <-- Copy(Tempstring,1,Y);
            end; { while }

      { Compose the remainder of the string. }
         Leftfield <-- Concat(Tempfield, Leftfield)
      end
   else { the comma flag is not set }
      Leftfield <-- Tempfield;
   { endif }

{ Check if a decimal point is required and,  if so, insert a
   period  and compose the entire string,  including Leftfield and
   Rightfield. }
      Finalstring <-- Leftfield;
      if (Periodcount > 0) then
         Finalstring <-- Concat(Finalstring, '.', Rightfield);
      { endif }

{ Check if a negative sign is required and,  if so, insert the
   character '-' to the left of Finalstring. }
   if ( Negative ) then
      Finalstring <-- Concat('-', Finalstring);
   { endif }

{ A special case occurs if the value of Number is zero. In this
   case check if a decimal point is required and,  if so,
   initialize Finalstring as '0.'  Then determine the number of
   zeros required to the right of the decimal point. If no decimal
   point is required,  initialize Finalstring as '0.'. }
   if (Number = 0) then
      if (Periodcount > 0) then
         begin
            Finalstring <-- '0.';
            for Y := 1 to Rightcount do
               Finalstring <-- Concat(Finalstring, '0');
         end
      else  { the edit field does not contain a decimal point }
         Finalstring <-- '0';
      { endif }
   { endif }

{ Check if a dollar sign is required and,  if so, insert '$'
   to left of  Finalstring. }
   if ( Markdollar ) then
      Finalstring <-- Concat('$', Finalstring);
   { endif }

{ Determine the difference between the size of the left side of
   the edit field and the left side of Finalstring, and insert any
   necessary blank spaces to the left of Finalstring. }
   Y <-- (Leftcount - Number_Leftdigits) - ord(Negative)
   while Y > 0 do
      begin
         Finalstring <-- Concat(' ',Finalstring);
         Y <-- Y - 1
      end; { while }
```

```
{ Display the formatted number. }
   write(  'The formatted number is ', Finalstring );
end; { else clause if Error is false }
end. { algorithm }
```

The procedure `Print_Using` is given here in a program called `Print_Using_Emulation`. This program allows you to observe the effects of the `Print_Using` procedure with different edit fields and different numbers.

This program requires a special function called `Remove_Blanks` for removing the blanks when the value of `Number` is converted into a string by the Macintosh Pascal function `StringOf`.

```
program Print_Using_Emulation;
{ Purpose:  Emulation of a Print Using command. }
{           Note: The SANE library is used, since we are using }
{           the variable Number as an extended type. }
   uses
      QuickDraw1, SANE;
   type
      Str = string[80];
   var
      Edit, Answer : Str;
      Number : extended;

{ *********************************************************** }
   procedure Text_Window;
   { Set the boundaries and show the Text window. }
      var
            Window_Rect : Rect;
   begin
         HideAll;
         SetRect(Window_Rect, 0,40,510,340);
         SetTextRect(Window_Rect);
         ShowText;
   end;
{ *********************************************************** }
   procedure Print_Using (Edit : Str; Number : extended);
   { Purpose:  Formats a number according to the style entered }
   {           into the edit field. }

      var
            Num, X : extended;
            Error, Markcomma, Markdollar, Negative : Boolean;
            Count, Leftcount, Len, Number_Leftdigits,
                     Number_Rightdigits, Periodcount : integer;
            Pointer, Rightcount, Y : integer;
            Character : char;
            Leftvalue, Len1, Len2, Num2, Rightvalue : longint;
            Leftfield, Finalstring, Numstring, Rightfield,
                     Tempfield, Tempstring, Threechar : Str;
```

```
{Two internal functions are declared:Power and Remove_Blanks.}
{   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
   function Power (X : extended; Y : integer) : extended;
   { Purpose: This function raises X to a power Y. }
      var
         Product : extended;
         J : integer;
   begin {Power}
      Product := 1.0;
      for J := 1 to Y do
         Product := Product * X;
      Power := Product
   end; {Power}
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
   function Remove_Blanks (Number : longint) : Str;
   { Purpose:  A custom string function that eliminates }
   {           leading blanks in a string. These blanks are }
   {           inserted when an integer value is }
   {           converted into a string using the library }
   {           function  StringOf. }
      var
         Counter, Len : integer;
         Tempstring : Str;
   begin
      Tempstring := StringOf(Number);
      Len := Length(Tempstring);
      Counter := 1;
      while (Tempstring[Counter]=' ') and (Counter < Len) do
         Counter := Counter + 1;
         Remove_Blanks := Copy(Tempstring, Counter, Len -
                                 Counter + 1)
   end;
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  }

begin { Body of procedure Print_Using. }
{ Initialize flags }
   Error := false;          { Error flag. }
   Markcomma := false;      { Comma flag. }
   Negative := false;       { Negative number flag. }
   Markdollar := false;     { Dollar flag. }

{ Initialize counters }
   Leftcount := 0;       { Count characters left of decimal. }
   Rightcount := 0;      { Count characters right of decimal. }
   Periodcount := 0;     { Count of periods encountered. }

{ Initialize temporary number variable. }
   Num := Number;

{ Check if number is negative.  If so, set the sign flag }
{ and set the temporary variable Num to the absolute value }
{ of the number. }
   if  Number < 0  then
      begin
         Negative := True; { set negative flag }
         Num := abs(Number);
      end; {if}

{ Check if length of edit field is greater than zero. }
   Len := Length(Edit);
   if  Len > 0 then
      begin
      { Initialize additional counters. }
         Count := 0;        { count digit positions }
         Pointer := 1;      { point to the current character }
                            { in the string }

      { Examine edit field for commas, dollar signs, periods }
      { and count places. If other characters are }
      { encountered or if the period or dollar sign }
      { is used improperly, set the error flag. }
         while  Pointer <= Len  do
            begin
               Character := copy(Edit, Pointer, 1);
            { select next character }
               case Character of
                  '$' : { If $ is the leftmost character, }
                        { set the dollar flag; else }
                        { there is an error, since the }
                        { character is out of place. }
                        if Pointer = 1 then
                           Markdollar := true
                        else
                           begin
                              Error := true;
                              writeln('Error1: Improper use
                                          of dollar sign.')
                           end; {else clause}
                  '#' : { Count the digit positions. }
                        Count := Count + 1;
                  '.' : { Check if the period has already }
                        { been encountered. }
                        begin {period check}
                           Periodcount := Periodcount + 1;
                           if  Periodcount > 1  then
                              begin
                                 Error := true;
                                 writeln('Error2: Too many
                                                periods.');
                              end {then clause}
                           else
                              begin
```

```
                                    Leftcount := Count;
                              {Count of digits left of period. }
                                    Count := 0
                              { Reset Count to keep track of }
                              { digits to right of period. }
                                 end;   {else clause}
                           end; {period check}
                     ',' :    { Check for commas. }
                           Markcomma := true;
                           otherwise
                           { illegal character in edit field }
                              begin
                                 Error := true;
                                 writeln('Error3: Illegal
                                       character in edit field.');
                              end { otherwise clause }
                  end; { endcase }
                  Pointer := Pointer + 1;
                  { Move pointer to next character. }
               end; {while loop}
            if  Periodcount > 0  then
            { If a period was encountered, Rightcount = Count. }
               Rightcount := Count
            else
            { No period was encountered, so Leftcount = Count. }
               Leftcount := Count;
            end   { then clause }
      else
   { The edit field is empty, since Len < 0,  so set error flag. }

         begin
            Error := true;
            writeln('Error4 : The edit field is empty.');
         end; { else clause }

{ Compose the number string and check its length against  }
{ that allowed in the edit field. }

{ Convert the number to an integer, including the digits to }
{ the right of the decimal point. }
      Num2 := trunc((Num * Power(10, Rightcount)) + 0.5);

{  Convert the integer to a string called Numstring. }
      Numstring := Remove_Blanks(Num2);

{ Check if the number of digits to the left of the decimal }
{ point, including space for a negative sign, exceeds the number }
{ of digits allowed in the edit field. }
      Len1 := Length(Numstring);
{ Len1 represents the number of digits in Numstring. }
      Len2 := Length(Numstring) + ord(Negative);
```

```
{ Len2 is the value of Len1 plus one for the digit position of }
{ a minus sign. }
      if  Len2 > Rightcount + Leftcount  then
         begin
            Error := true;  { set error flag }
            writeln('Error5 : The number exceeds the edit
                     field.');
         end;

{ If an error exists at this point, print % + the number and }
{ skip the  remainder of this procedure.  The error message }
{ will already have been printed. }
      if  Error  then
         begin
         { The error flag has been set. }
            writeln;
            writeln('%', Number : 1 : 5);
            writeln;
         end {then clause}
      else  { There is no error, so continue. }
         begin
{ Compose the entire number string, which will consist of the }
{ digits to the left of the decimal point, the dollar sign if }
{ needed, any necessary commas,  the decimal point if required, }
{  and any digits to the right of the decimal point. }

{ Extract from Num2 the leftmost and rightmost digits adjacent }
{ to the decimal point.}

            Leftvalue := trunc(Num2 / Power(10, Rightcount));
            Rightvalue := trunc(Num2 - Leftvalue * (Power(10,
                                     Rightcount)) + 0.5);

{ Determine Rightfield and check if it needs to be padded }
{ with zero on the left of the string. }
            if  Rightcount > 0  then     { Rightcount  greater }
                                         { than 0. }
               begin
                  Rightfield := Remove_Blanks(Rightvalue);
                  Number_Rightdigits := Length(Rightfield);
                  if  Rightcount > Number_Rightdigits  then
                     begin
                        for Pointer := 1 to (Rightcount -
                                         Number_Rightdigits) do
                           Rightfield := Concat('0', Rightfield);
                     end; {if}
               end {then clause}
            else    { Rightcount not greater than 0 }
               Rightfield := '';

      {  Initialize Leftfield and Number_Leftdigits. }
```

```
            Leftfield := '';
            Number_Leftdigits := 0;

{ If Leftcount > 0, create temporary variables Tempfield and }
{ Tempstring; determine the number of digits left of decimal. }
            if  Leftcount > 0  then
               begin
{ Take the actual number of digits to the left of the decimal }
{ and assign to Tempfield. }
                  Tempfield := Remove_Blanks(Leftvalue);
                  Tempstring := Tempfield;
                  Number_Leftdigits := Length(Tempfield);
               end;  { then clause }

      { If commas are required, insert them in the left field. }
            if  Markcomma  then
               begin
                  Y := Number_Leftdigits;
{ Create a counter equal to the number of  left digits. }
                  while  Y > 3  do
                     begin
{ Extract the three rightmost digits in Tempfield. }
                        Threechar := Copy(Tempfield, Y - 2, 3);
                        Leftfield := Concat(',', Threechar,
                                            Leftfield);
                        Y := Y - 3;
{ Extract remaining leftmost digits from Tempstring. }
                        Tempfield := Copy(Tempstring, 1, Y);
                     end;

{ Concatenate remaining digits with Leftfield. }
                     Leftfield := Concat(Tempfield, Leftfield);
               end  { then clause }
            else  { Comma flag is not set. }
               Leftfield := Tempfield;

{ Insert a period if required and put fields together. }
            Finalstring := Leftfield;
            if  Periodcount > 0  then
               Finalstring := Concat(Finalstring, '.',
                                            Rightfield);
            {endif}
   { If number is negative, insert a minus sign. }
            if   Negative  then
               Finalstring := Concat('-', Finalstring);
            { endif }
   { Now consider the special case of Number = 0. }
            if  Number = 0  then
               if  Periodcount > 0  then
               { If there is a decimal point, initialize }
               { Finalstring. }
```

```
                  begin
                     Finalstring := '0.';
                     for Y := 1 to Rightcount do
                        Finalstring := Concat(Finalstring, '0');
                  end { then clause }
               { endif }
            else { If there is no decimal point. }
                  Finalstring := '0';

      {  Insert the dollar sign if required. }
            if  Markdollar  then
               Finalstring := Concat('$', Finalstring);
         { endif }

{ Insert the necessary number of blank spaces to the left of }
{ Finalstring. Include a space for the sign if it is needed. }
            Y := (Leftcount - Number_Leftdigits) - ord(Negative);
            while  Y > 0  do
               begin
                  Finalstring := Concat(' ', Finalstring);
                  Y := Y - 1;
               end;

      {  Display the formatted number. }
            writeln('The formatted number is ', Finalstring);
      end;  { else clause if Error is false }
   end;

{ ************************************************************* }
begin { Body of the main program. }
{ Open Text window. }
   Text_Window;
{ Enter edit field from keyboard. }
   writeln('Enter edit field');
   readln(Edit);
{ Enter a number to be formatted. }
   repeat
         writeln('Enter number: ');
         readln(Number);
         Print_Using(Edit, Number);
         writeln;
         write('Do you wish to enter another number?');
         readln('Type YES or NO:  ');
         readln(Answer);
   until  (Answer = 'No') or (Answer = 'no') or (Answer = 'NO');
end.
```

SUMMARY

In this chapter we discussed the Macintosh Pascal string routines `Length`, `Pos`, `Concat`, `Copy`, `Delete`, `Omit`, `Insert`, and `Include`, and considered sample programs that implemented many of these procedures and functions. The procedure called `String_Match` illustrated the basic concept of pattern matching, given a source string and a pattern. A procedure called `Replacement` showed how a substring can be substituted for a pattern located within a source string. Finally, the program called `Print_Using_Emulation`, using a procedure called `Print_Using`, showed how to use strings representing numeric values to format numbers for display.

This chapter also considered two miscellaneous but important string routines for Macintosh Pascal. The function called `StringOf` can convert the values of variables represented in internal machine form to characters concatenated into a single string function. The second procedure, called `ReadString`, can reverse the process of the function `StringOf`.

REVIEW QUESTIONS

1. Define the declaration of a string called `Sentence` having at most 80 characters.
2. What are the two attributes associated with a `string` type?
3. How much storage does the following string declaration require?

```
Line_of_Text : string[47] ?
```

4. If `Line_of_Text` is assigned the string 'having to program in Pascal', what Pascal code will display each individual character?
5. Can the string

```
' THIS IS ANOTHER EXAMPLE OF A LINE OF TEXT TO BE ASSIGNED TO A
    STRING VARIABLE. '
```

be assigned to `Line_of_Text`? Explain your answer.

6. What operators are allowed with `string` variables?
7. Which of the two following strings are ordered greater: `'drawing'` or `'drawn'`?
8. What action is taken by the following conditional statement?

```
Name := 'Jones';
Match := ' Jones ';
if Name = Match then
   writeln(' Dear Mr./Mrs./Ms. ', Name );
```

9. What actions are taken by the following conditional statement?

```
First_Name := 'Jones';
Second_Name := 'Jonies';

if First_Name > Second_Name then
   List_Name := First_Name
```

```
else
   List_Name := Second_Name;
```

10. What other types of data can be compared with a `string` type?
11. Is a blank space equivalent to a null string?
12. How can a null string be explicitly represented in Pascal?
13. List the string procedures supported by Macintosh Pascal.
14. List the string functions supported by Macintosh Pascal.
15. What is the purpose of the string function `Pos`?
16. What is the purpose of the procedure `Delete`?
17. What is the purpose of the procedure `Insert`?
18. Write the necessary code, using the function `Pos` and the procedures `Delete` and `Insert`, to remove a substring from a source and replace it with another substring.
19. What limitations exist when using the procedure `Insert`?
20. What are the differences between the procedure `Insert` and the function `Include`?
21. What is the purpose of the function `Copy`?
22. What is the purpose of the function `Omit`?
23. Can the function `Copy` be used to concatenate strings?
24. Assume that `A, B, C, D, E, F`, and `G` are all declared as type `string`. Given the following assignments, what is the result of executing the `Concat` function?

```
A := '.';
B := 'is ';
C := 'beginning';
D := 'This ';
E := 'only ';
F := 'the ';
G := Concat(D, B, E, F, C, A);
```

25. Can the procedure `String_Match` in Section 11.3 be used to find the position of a null string?
26. Can the procedure `Replacement` in Section 11.3 be used to replace a null string with a nonnull string?
27. What is the purpose of the Macintosh Pascal string function `StringOf`?
28. What is the purpose of the Macintosh Pascal string procedure `ReadString`?
29. What are the advantages of having the two routines `StringOf` and `ReadString`?

PROGRAMMING EXERCISES

Not all programming exercises require you to write an algorithm, but you may better understand the problem and what is required if you first write an algorithm and trace it by hand with several examples before writing the Pascal program.

1. Write a program for testing the two procedures `String_Match` and `Replacement`.

2. Write a procedure called `Abbreviate_Name` that takes a string representing a person's full name in the format *last name, first name middle name* and returns the following string: *first character of first name. first character of middle name. last name*. For example, consider `Source_Name` to have the value `'Macintoshes, John Abner'`. The procedure `Abbreviate_Name(Source_Name, New_Name)` would return for New_Name the value

   `'J. A. Macintoshes'`.

3. The cost of sending a telegram from Macinville to Apple Village is $2.15 for the first 20 words or less, plus 15 cents for each additional word beyond 20. Develop an algorithm that accepts a line of text, counts the individual words, and computes the total cost based upon the number of words. A word is any string of characters having a leading blank and ending with either a blank or a punctuation mark. After you have tested this algorithm, convert your algorithm into a Macintosh Pascal program, and execute it.

4. Write a program that prompts the user for a single line of text. This program is required to take the characters from the string and display full words to the Text window. A word is defined as any group of characters starting with an alphabetic letter that continues over all characters, ending when a punctuation character is encountered. A punctuation character is defined as one of the following: !, @, #, $, ^, &, *, (,), =, +, {, }. [,], :, :, ", ', <, >. ?, /, |, \, ~, `, comma, period, blank.

5. Modify the program in Exercise 4 so that it can count each of the individual punctuation marks, with the option of reporting these total counts after the full words have been reported.

6. Assume that a cipher is to be used for coding words stored within a database. This will require two procedures: `Plaintext_to_Code` and `Code_to_Plaintext`. Here is a list of the plaintext alphabet versus the cipher alphabet:

```
Plaintext: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Cipher   : Z V G I N J H R S U O Q B K X L P W T M E A Y C D F
```

For example, plaintext `A` would be replaced by `Z`, plaintext `B` by `V`, and so on. Develop and implement a program that offers the user two options:

(a) Enter plaintext from the keyboard, encipher this text, and display the enciphered code to the Text window (provide encoding).

(b) Enter ciphertext from a file, and display the deciphered code to the Text window (provide decoding).

7. Write a function called `Pad` that requires three formal parameters: `Str`, representing a string; `Fill`, representing one or more filler characters to pad onto the right of parameter `Str`; and `Size`, representing the number of times the string is to be extended with fill characters. This function will return a string not exceeding 255 characters. For example,

```
Source := 'This is a string';
New_String := Pad( Source, '*/', 4 );
```

would result in `New_String` having the value `'This is a string*/*/*/*/'`.

8. Write a program that will search an object string with a pattern and produce a list of positions where the pattern begins in the object string. Extend this program with a procedure that replaces all instances of the pattern in the object string with a replacement string.

9. Write a function called `Reverse_Characters` that will completely reverse the characters of a string. For example, the string `'A line of text as a test.'` would have the value `'.tset a sa txet fo enil A'`.

10. Write a function called `Reverse_Words` that will reverse the order of the words within a string. For example, the string `'This is another line of text.'` would have the value `'text. of line another is This'`.

11. Write individual functions for each of the following operations:

```
Convert_Integer_String( Integer_Number )
Convert_Real_String( Real_Number )
Convert_String_Integer( Numeric_String )
Convert_String_Real( Numeric_String )
```

12. Modify the `Print_Using` procedure so that when a dollar sign is required, it will be printed on the leftmost side of the edit field, with blanks inserted as needed between it and the number.

13. Assume that a line of text to be entered from the keyboard is limited to a maximum of 80 characters. Develop and implement a program that will display a line of text to the Text window having one the following options:

(a) Line of text is to be displayed right-justified. (Line is positioned to the far right with leading blanks to the left.)
(b) Line of text is to be displayed center-justified. (Line is positioned in the center with equal blanks to the left and right of the text line .)

(c) Line of text is to be displayed left-justified. (Line of text is positioned to the left of the display line.)

14. The Zeta-Data Flea Company uses a string as an inventory number. This number is represented by the format

```
dddddddddccccccccccccdd/dd/dd$dddd.dd#ddddd
```

The first 9 digits represent an item number, the next 12 characters represent a brief description of the item, followed by the date of entry into the company's catalog, followed by the unit price, followed by the number of these items remaining in inventory. For example, consider the following coded inventory number:

```
543098766Steel Hammer12/28/87$0015.99#00123
```

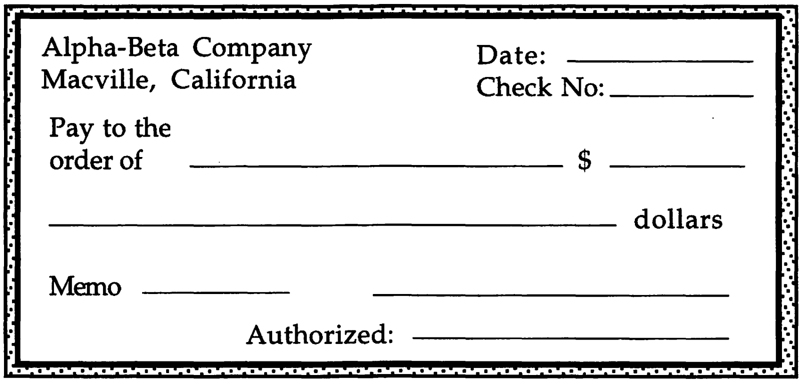
Develop and implement a program that will accept the inventory number as a string, assign the first 9 digits of the inventory number to a `string` variable representing a shorter inventory number, the next 12 characters as a string to a variable storing the description, the date as a string to a variable for storing the date of catalog entry, a `real` variable for storing the unit price, and the last 5 digits to an integer number for storing the quantity of items in stock. Have your program report the item to the Text window, using the following format:

```
Inventory Number   Description   Date of Catalog   Unit Price  Quantity
                                                                in Stock
-------------------------------------------------------------------------
543098766          Steel Hammer    12/28/87          $  15.99      123
```

15. The Alpha-Beta Disk Company has decided to use its Macintosh to print checks. Each check is to be drawn in the Drawing window, as shown in Figure 11.6. The following input is required from the keyboard in response to prompts appearing in the Text window:

(a) Next check number
(b) Date
(c) Person or company to whom the check is to be issued
(d) Amount of the check (entered as a real number)
(e) Person authorized to issue the check

The check-writing program must be able to convert the amount as a real number into an equivalent set of words. For example, if the amount is entered as $125.87, the equivalent words printed on the check must read `ONE HUNDRED TWENTY-FIVE AND 87/100`. The maximum limit on any check is $1000. Develop an algorithm for this problem, and after you have tested it with several examples, implement it as a Macintosh Pascal program. Use your examples to see if the program (algorithm) is functional. Use procedures at various steps to implement the algorithm.



Alpha-Beta Company
Macville, California

Date: ____________
Check No: ____________

Pay to the
order of ______________________ $ __________

____ dollars

Memo __________ ______________________

Authorized: ______________________

Figure 11.6

16. A palindrome is a sequence of letters that reads the same forward and backward. For example, the line *Never odd or even.* reads the same from left to right or from right to left if you ignore spaces, capital letters, and punctuation. Develop an algorithm that can test if a line of text is a palindrome. This algorithm will have the following major set of steps:

    (a) Prompt the user, and accept a line of text from the keyboard.
    (b) Convert all uppercase alphabetic letters to lowercase.
    (c) Compress the line of text by removing all blanks and punctuation characters.
    (d) Reverse the characters in the text line, and compare with the compressed text line.
    (e) Test if both the given text line and the reversed text are palindromes.

    Each step must be implemented by a call to a procedure. Once you have tested your algorithm, transform it into a Macintosh Pascal program, and test it. Try some of the following palindromes as test cases:

```
Evil I did I live.
Able was I ere I saw Elba.
12345 54321
abcdef fedcba
A man, a plan, a canal Panama!
```

17. Suppose you need to display a fixed decimal or floating-point number in a floating-point format, where the precision of the integer and fractional parts can be specified using a series of # characters, and where the explicit letter *E* is given. Using the concepts of the algorithm titled `Print_Using`, develop a new algorithm called `Print_Using_ Float` that can display a real number, given a string of characters as the format for displaying a new value in a string representation. In specifying the format, use the the character ^ to reserve space for the

letter *E*, followed by a plus sign if the exponent is positive or a minus sign if negative. If fewer than four carets (^) are used, the number is not printed in *E* format; instead, carets are printed as a literal. If more than four carets are used in the format, the number is displayed in *E* format but with extra carets trailing the resulting string. Here are some examples of using the `Print_Using_Float` procedure:

Initial Value	Format	Result of `Print_Using_Float`
3	###.##^^^^	300.00E-02
1234	###.##^^^^	123.40E+01
234	###.##^^	234.00^^
-6787.65	####.##^^^^^	-678.77E+01^

Whenever possible, the resulting number should be rounded up. If a value cannot fit the format, the algorithm must default to the standard format. When you have written and successfully tested your algorithm, transform your algorithm into a Pascal program, and test to see that it works.

Chapter 12

Pointers

OBJECTIVES

After completing Chapter 12, you will know the following:

1. The concept of an abstract data type (ADT).
2. The concept of the pointer as a dynamic variable.
3. How to create an object as an abstract data type, using the command `new` and deallocated using the command `dispose`.
4. How to use the pointer to define special data objects: stacks and queues.
5. Application of pointers to the growing and pruning of binary trees.
6. The Macintosh memory routines and the concept of handles.

12.1 THE CONCEPT OF AN ABSTRACT DATA TYPE

We often need to abstract objects representing data as well as the operations associated with those objects when we define the algorithms of an application. This implies defining a data type as conjectural information along with the operations that support the abstraction. The detail of how the data type is to be implemented is not relevant when we are defining the abstract type. What is important is the concept that the data object represents and the operations that it can perform.

For example, we have accepted the `string` type as a useful structure for storing character strings of varying lengths. We have accepted basic operations of reading, writing, concatenating, and decomposing objects of type `string`. We accept these operations even though we do not understand the internal structure of a `string` type. Although we may have some preconceived ideas on how a `string` type may be implemented, it is for the most part accepted as an abstract data type when programming in Pascal. Our ability to apply a `string` type is purely as an abstract data type. In computer science, it is common for data structures to be classified into categories such as linear, hierarchical, and network. If we understand the basic operations on these types of classifications, we can define and implement abstract data objects. Initially, the detail of how an abstract data type is to be implemented is secondary to understanding its definition and basic operations.

In the context of this discussion an abstract data type (ADT) represents a special data type composed of two parts. First, it is composed of a statement describing the relationships of its own components and its base type. Second, it is composed of a statement describing the actions (operations) that can be performed on elements of this type. Initially, the base type of its components is often treated as some vague basic type, and the operations are generally classified as constructors, destructors, modifiers, selectors, and iterators.

A *constructor* is an operation for creating an object and, if necessary, initializing its state. For example, giving a data object an initial value is an example of a constructor. A state represents the condition in which an object exists, characterized by values associated with the properties of the object. A *destructor* is an operation that frees the state of an object by either nulling the value of the object by destroying it. A *modifier* is an operator that alters the state of an object, amd a *selector* is an operation that can access the state of an object. For example, reassigning a value to part of a data object is a modifier, whereas simply reading this value is a selector. A selector is not allowed to alter the state of an object. Last, an *iterator* allows all or part of an object to be accessed in some defined order. We may also need to consider whether to view an abstract data type as being bounded or unbounded. Is it limited in size, or can it continue to grow without any bound? Are there exceptions (problems) that may emerge during the lifetime of the object?

As an example consider the definition of an abstract data type called a *list.* A list is a finite sequence of elements of an arbitrary type `T` together with a collection of operations. These operators are listed according to the following five categories:

Constructor:
- Create the structure representing a list.
- Initialize a list as being empty.

Destructor:
- Destroy the structure representing a list.

Modifier:
- Insert a new element of type `T` at some position within a list.
- Replace an element of type `T` within a list with that of another element of type `T`.
- Delete an element of type `T` from a list.

Selector:
- Determine the length of a list.
- Determine if a list is full.
- Determine if a list is empty.

Iterator:
- View all elements of a list.
- Search a list for an element that has a state associated with that of type `T`.

Notice that each operation in the definition is highly cohesive (highly functional); that is, each operation performs a unique action. Although it might be convenient to combine one or more operations, this can make the definition of an abstract data type less exact and can lead to an ambiguous implementation. In addition, two exceptions are implied by two of the selector operations. One exception is when the list is empty and the other is when the list is full. Is the list a bounded or unbounded structure? The definition is clear: the list is bounded. Why? First, it is defined as a finite sequence of elements. Second, being finite and having a selector to test if the list is full, it must be implemented as a bounded structure to satisfy its own definition.

In the sections that follow, you will see how to use Macintosh and THINK Pascal to implement both bounded and unbounded abstract data types. By encapsulating the definition of an abstract data type within a program **unit**, the information of the abstract type is hidden. Only the programmer need see documentation on the definition of an abstract data type, including the rules for implementing its operators.

12.2 POINTERS AND DYNAMIC VARIABLES

Pointers allow us to write Pascal programs in which structures for data objects can be dynamically allocated and deallocated during execution. In actuality, pointers reference physical memory locations during the execution of a program. Through the execution of the command `new`, storage is allocated, and an identifier is made to point to the object. This same object can be deallocated storage through the execution of the command `dispose`. We define a pointer as a simple data object capable of pointing to a physical location in memory that contains storage for another data object. As an example, consider the following two declarations:

```
var
   First_Number : ^integer;
   Second_Number : ^real;
```

When declaring a pointer type, the character ^ precedes the base type. In the example, the variable `First_Number` is a pointer to an `integer` type, and the variable `Second_Number` is a pointer to a `real` type. It is convenient at times to declare pointers through a programmer-defined type. For example, the following statements declare the variables `First_Number` and `Second_Number` in terms of the programmer-defined types `Integer_Number` and `Real_Number`, respectively. `Integer_Number` is associated with a pointer to an `integer` type, and `Real_Number` is associated with a pointer to a `real` type. Through this indirect level of reference, both variables remain pointer types.

```
type
   Integer_Number = ^integer;
   Real_Number = ^real;
var
   First_Number : Integer_Number;
   Second_Number : Real_Number;
```

Pointer types are different from the other types discussed in Chapter 3. First, a pointer variable can directly reference a memory location storing an object. When dealing directly with the memory address, only the name of the pointer variable is used. A pointer variable has the value **nil** when a program begins execution. From the level of Pascal **nil** represents an address in memory compatible with all pointer types. It also is a memory location containing no value of any data type. Second, storage is only allocated for the object being pointed to by the pointer variable when the command `new` is executed. For example, attempts to reference the variables `First_Number` or `Second_Number` produce an error message at the time of execution unless storage has been explicitly allocated. Although each pointer variable exists when the program begins execution, objects pointed to by the variables do not exist until the command `new` is executed. As a procedure, the command new uses the following format:

```
new( Pointer_Variable );
```

Each time the command new is executed, storage is allocated for a new dynamic variable by creating an object having a base type associated with the pointer variable, and setting the pointer variable to point to this new object. For example, the command `new(First_Number)` allocates storage for an object capable of storing an integer number with `First_Number` pointing to this object. Figure 12.1 shows a graphic representation of the execution of the command `new`, involving the pointer variables `First_Number` and `Second_Number`.

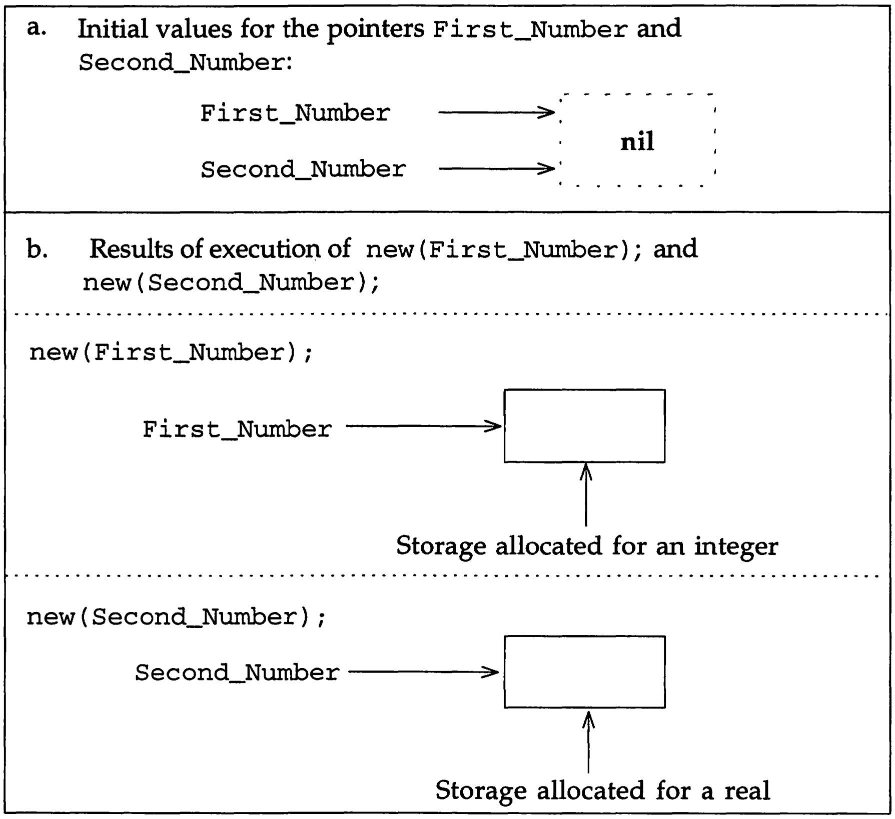

a. Initial values for the pointers First_Number and Second_Number:
First_Number
Second_Number
nil
b. Results of execution of new(First_Number); and new(Second_Number);
new(First_Number);
First_Number
Storage allocated for an integer
new(Second_Number);
Second_Number
Storage allocated for a real

Figure 12.1 A graphic view of the execution of the procedure `new`.

A third difference characterizing a pointer type involves referencing the value of the object pointed to by the pointer variable. To reference a value associated with a pointer variable, use the pointer variable name followed by the character ^. For example, after executing the command `new(First_Number)`, the statement `read(First_Number^)` reads an integer value from the keyboard and then assigns this to the allocated storage area pointed to by `First_Number`. Keep in mind that the variable `First_Number` represents the pointer, which points to an address in memory capable of storing an integer value. The object `First_Number^` accesses the integer value referenced by the pointer `First_Number`.

Before terminating execution, the program frees memory used for the storage of the dynamic variable by executing the command `dispose`. The purpose of this command is to free the storage pointed to by the pointer variable, but not to destroy the pointer variable itself. The following represents the syntax for this command:

```
dispose( Pointer_Variable );
```

Any attempt to dispose of an object that has no defined storage or of a pointer variable that has a value of **`nil`** causes an error to be reported at the time of execution. The following short program, `Pointers`, demonstrates the concepts `new` and `dispose`.

```
program Pointers(input, output);
{ Purpose:  This program demonstrates the use of the commands }
{           new and dispose. }

   type
      Integer_Number = ^integer;
      Real_Number = ^real;
   var
      First_Number : Integer_Number;
      Second_Number, Third_Number : Real_Number;

begin
{ Allocate storage for the first and second numbers. }
   new(First_Number);
   new(Second_Number);
{ Enter values for the first and second numbers. }
   write(' Enter an integer number: ');
   readln(First_Number^);
   write(' Enter a real number: ');
   readln(Second_Number^);
{ Allocate storage for the third number.  }
   new(Third_Number);
{ Assign to the third number the sum of the values of the first }
{ and second numbers. }
   Third_Number^ := First_Number^ + Second_Number^ ;
{ Display the value of each number. }
   writeln(' First number: ', First_Number^ );
   writeln(' Second number: ',  Second_Number^: 9: 3 );
   writeln(' Third number: ', Third_Number^: 9: 3 );
{ Deallocate storage for each number. }
   dispose(First_Number);
   dispose(Second_Number);
   dispose(Third_Number);
end.
```

It will be common in this chapter, when referring to an expression such as First_Number^, to allude to it as the "value of the object to which First_Number is pointing."

As a second example, consider the following program, titled Pointer_to_Employee. This short program allocates storage to a dynamic record structure represented by the pointer variable Employee. During execution of this program, values are assigned to a field of the record pointed to by Employee. Before disposing of storage for the object Employee, the program displays the same information to the Text window.

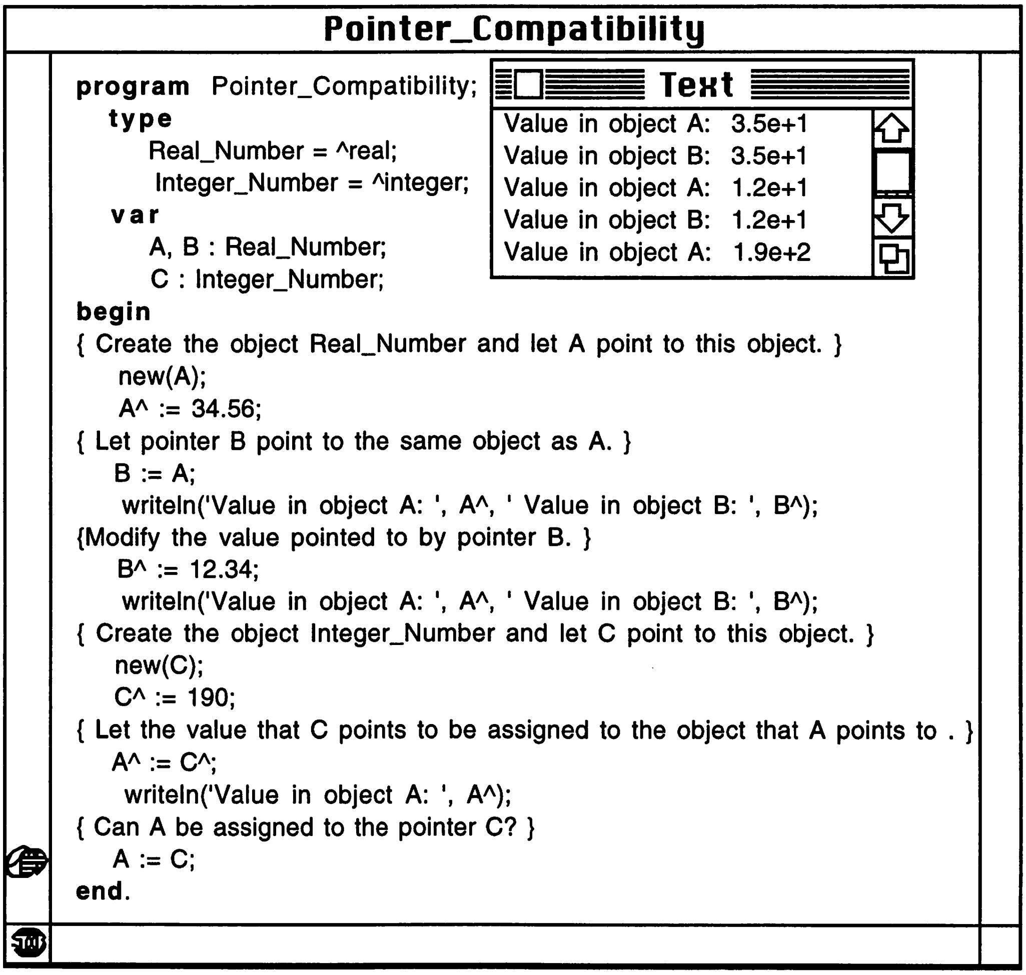
```
program Pointer_to_Employee(input, output);
{ Purpose:  This program reads data from the keyboard and stores }
{           this information in a record being pointed to by the }
{           variable Employee. }
   type
      Profile =   record
                     ID_Number : string;
                     Name : string;
                     Wage : real
                  end;
      Folder = ^Profile;
   var
      Employee : Folder;
begin
{ Create storage for an employee record. }
   new(Employee);
{ Enter employee data from the keyboard. }
   with Employee^ do
      begin
         write('Enter full name: ');
         readln(Name);
         write(' Enter identification number: ');
         readln(ID_Number);
         write('Enter hourly wage: ');
         readln(Wage)
      end;
   writeln;
{ Display employee data to the Text window. }
   with Employee^ do
      begin
         writeln(Name);
         writeln(ID_Number);
         writeln('$', Wage : 5 : 2)
      end;
   dispose(Employee);
end.
```

The variable Employee is represented by a programmer-defined type called Folder. Folder is in turn represented by a pointer type pointing to a record called Profile. Although Employee is a programmer-defined type, Folder, it is, by definition, a pointer type of type Folder. This example makes use of the **with** clause for accessing fields of the record object pointed to by the pointer Employee. Individual fields of the record could be accessed through references such as Employee^.Name, Employee^.ID_Number, and Employee^.Wage. It will be common in this chapter, when referencing an expression such as Employee^.Name, to refer to this as the "name field" of the object to which Employee is pointing.

A fourth difference of the pointer type is in the location where objects pointed to are stored in RAM. During execution of a Pascal program, storage for all static objects is physically placed on a system stack referred to as a *run-time stack.* A stack is itself a special data structure in which data is stored in last-in, first-out order. That is, the last object pushed onto the top of the stack is the first object to be "popped off." The area allocated to the stack must be contiguous, with space being allocated or deallocated at the top, never at the middle or end. When a procedure or function is called, and before the routine begins execution, the environment of the routine, represented by all formal parameters, local constants, and local variables, is pushed onto the top of the run-time stack. Thus the environment at the top of the run-time stack always represents the routine presently in execution. During execution access to all variables is through references to memory locations on the run-time stack. Once a routine has completed execution, the environment at the top of the stack is popped off, leaving the the current executing environment as the environment now at the top of the stack. Whereas pointer variables are stored on a stack, dynamic objects created by executing the command new are stored in an auxiliary storage area called a *heap.*

Heap storage is different from stack storage in that it allows dynamic objects to be allocated and deallocated storage from a free storage pool as requested by the program. Whereas the memory size of an environment is estimated at translation time, memory needed for dynamic objects is not known until the program is executed. Pushing dynamic objects onto the run-time stack instead of allocating storage in a heap could make management of the stack difficult. For example, though it may be necessary to pop the current environment, allocation of a dynamic object pointed to by an actual parameter would need to remain. This would lead to holes in the stack and violate the concept of contiguous storage allocation.

Pointer variables must be type-compatible. Let us examine the screen dump of the program Pointer_Compatibility, shown in Figure 12.2. Note that both A and B are declared as pointers that can point to a real number. In this program an object of type Real_Number is created and pointed to by A by executing new(A). By means of the assignment statement B := A, pointer B takes on the value of pointer A. That is, pointer B does not make a copy of this object; it points to the same object as pointer A. If the value to which A points is changed, the value of the object to which B points will change as well, because it is the same as that for pointer A. Changing the value of the object to which B points changes the value of the object to which A points because both A and B now point to the same object. Notice that the statement A^ := C^ takes a copy of the value to which C is pointing and assigns this to the object to which A is pointing. This is different from the assignment statement A := C. This latter statement fails to execute because pointers A and C are not type-compatible.



Pointer_Compatibility

```
program Pointer_Compatibility;
  type
     Real_Number = ^real;
      Integer_Number = ^integer;
  var
     A, B : Real_Number;
     C : Integer_Number;
begin
{ Create the object Real_Number and let A point to this object. }
   new(A);
   A^ := 34.56;
{ Let pointer B point to the same object as A. }
   B := A;
    writeln('Value in object A: ', A^, ' Value in object B: ', B^);
{Modify the value pointed to by pointer B. }
   B^ := 12.34;
    writeln('Value in object A: ', A^, ' Value in object B: ', B^);
{ Create the object Integer_Number and let C point to this object. }
   new(C);
   C^ := 190;
{ Let the value that C points to be assigned to the object that A points to . }
   A^ := C^;
    writeln('Value in object A: ', A^);
{ Can A be assigned to the pointer C? }
   A := C;
end.
```

```
Value in object A:  3.5e+1
Value in object B:  3.5e+1
Value in object A:  1.2e+1
Value in object B:  1.2e+1
Value in object A:  1.9e+2
```

Figure 12.2 Example of pointer-compatibility problems in Pascal.

Consider the following statements, where it is assumed that the variables A, B, C are pointers of the same compatible type:

```
new(A);
A^  := 120;
B := A;
C := A;
dispose(A);
```

In some Pascal systems you may encounter the problem of dangling pointers (dangling references). A dangling pointer occurs when several pointers are pointing to a single object, and that object is deallocated by execution of the command `dispose`. The remaining pointers are left dangling, because the object being pointed to no longer exists. In the example just given, B and C would be dangling after execution of `dispose(A)`, because the object they had been pointing to has been deallocated from the heap.

Macintosh Pascal avoids this problem by using a reference count technique for all objects stored in the heap. A reference counter is an internal counter associated with each object that counts the number of pointers pointing to the object stored in the storage heap. For example, if A, B, and C are compatible pointers, the following statements would lead to a count of 3 after execution of the statement C := A :

```
new(A);         { reference count of 1 for the created object A }
B := A;         { reference count of 2 for the created object A }
C := A;         { reference count of 3 for the created object A }
dispose(A);     { reference count of 2 for the created object A }
dispose(B);     { reference count of 1 for the created object A }
dispose(C);     { reference count of 0 for the created object A }
```

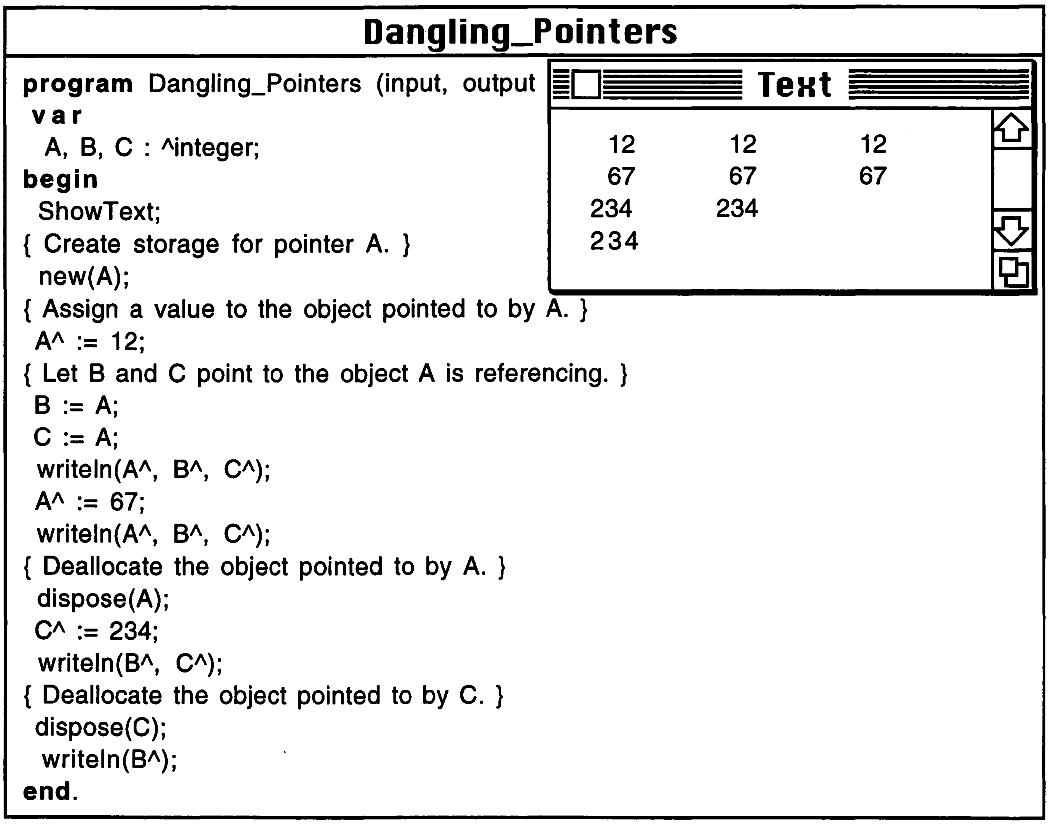
Figure 12.3 shows another example of the problem of dangling pointers in Macintosh Pascal. Storage is first allocated for the object pointed to by A. After assigning a value to A^, the remaining two pointers, B and C, are assigned to point to what A is referencing. Even after disposing of A, we can still access the values of B^ and C^, because the object that A was pointing to still exists. Simply put, its reference count has decreased by 1 but remains greater than zero. Only when the reference count for an object becomes zero is storage for the object deallocated from the storage heap. THINK Pascal does not follow this convention. An attempt to execute the program Dangling_Pointers under THINK Pascal will fail. Thus, the THINK Pascal programmer needs to be more careful to avoid creating dangling references when using the dispose procedure.

In some Pascal systems you may encounter a problem of excess garbage being created in heap storage by a program. For example, the following statements result in the modification of pointer A, leaving the initial object generated by the execution of new(A) still in existence. It becomes garbage when the statement A := B is executed, because pointers to this object no longer exist.

```
new(A);
A^ := 120;
new(B);
A := B;   { A now points to where B is pointing, leaving the }
          { initial object of A unreferenced but still in heap }
          { storage. }
```

Macintosh Pascal protects itself from generating garbage in heap storage by using reference counters. Whenever the reference count for an object becomes zero, the storage for that object in the heap is deallocated. For example, execution of the statement A := B above results in the reference count for the initial object pointed to by A becoming zero, and the object is deallocated from memory.

Both THINK and Macintosh Pascal support a special unary operator represented by the character @. This operator can take as an operand a single variable reference, a formal parameter, a procedure identifier, or a function identifier and then compute the address of this operand as its value.



```
program Dangling_Pointers (input, output
 var
  A, B, C : ^integer;
begin
 ShowText;
{ Create storage for pointer A. }
 new(A);
{ Assign a value to the object pointed to by A. }
 A^ := 12;
{ Let B and C point to the object A is referencing. }
 B := A;
 C := A;
 writeln(A^, B^, C^);
 A^ := 67;
 writeln(A^, B^, C^);
{ Deallocate the object pointed to by A. }
 dispose(A);
 C^ := 234;
 writeln(B^, C^);
{ Deallocate the object pointed to by C. }
 dispose(C);
 writeln(B^);
end.
```

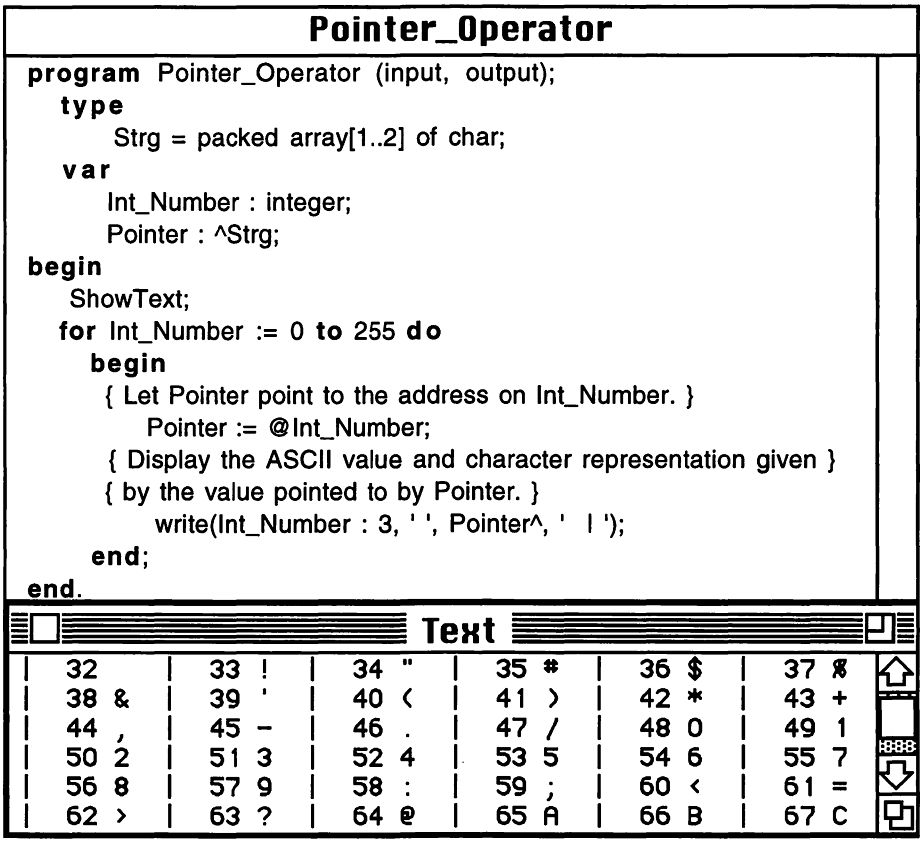
```
  12      12      12
  67      67      67
 234     234
 234
```

Figure 12.3 A simple example showing that Macintosh Pascal does not suffer from the problem of dangling pointers.

Consider the example shown in the screen dump of `Pointer_Operator` and its output, as shown in Figure 12.4. Here the identifier called `Pointer` is a pointer of type `Strg`. Notice that the assignment statement

```
Pointer := @Int_Number;
```

assigns the address of the location of the variable `Int_Number` to `Pointer`. `Pointer` treats the location of `Int_Number` as a two-character string. The loop displays both the ASCII integer value and the equivalent character to the Text window for each value stored in `Int_Number`.



Pointer_Operator

```
program Pointer_Operator (input, output);
  type
     Strg = packed array[1..2] of char;
  var
     Int_Number : integer;
     Pointer : ^Strg;
begin
   ShowText;
  for Int_Number := 0 to 255 do
     begin
      { Let Pointer point to the address on Int_Number. }
          Pointer := @Int_Number;
      { Display the ASCII value and character representation given }
      { by the value pointed to by Pointer. }
           write(Int_Number : 3, ' ', Pointer^, '  | ');
     end;
end.
```

Text

```
|  32    |  33 !  |  34 "  |  35 #  |  36 $  |  37 %
|  38 &  |  39 '  |  40 (  |  41 )  |  42 *  |  43 +
|  44 ,  |  45 -  |  46 .  |  47 /  |  48 0  |  49 1
|  50 2  |  51 3  |  52 4  |  53 5  |  54 6  |  55 7
|  56 8  |  57 9  |  58 :  |  59 ;  |  60 <  |  61 =
|  62 >  |  63 ?  |  64 @  |  65 A  |  66 B  |  67 C
```

Figure 12.4 Screen dump for the program `Pointer_Operator` and the results of its output to the Text window.

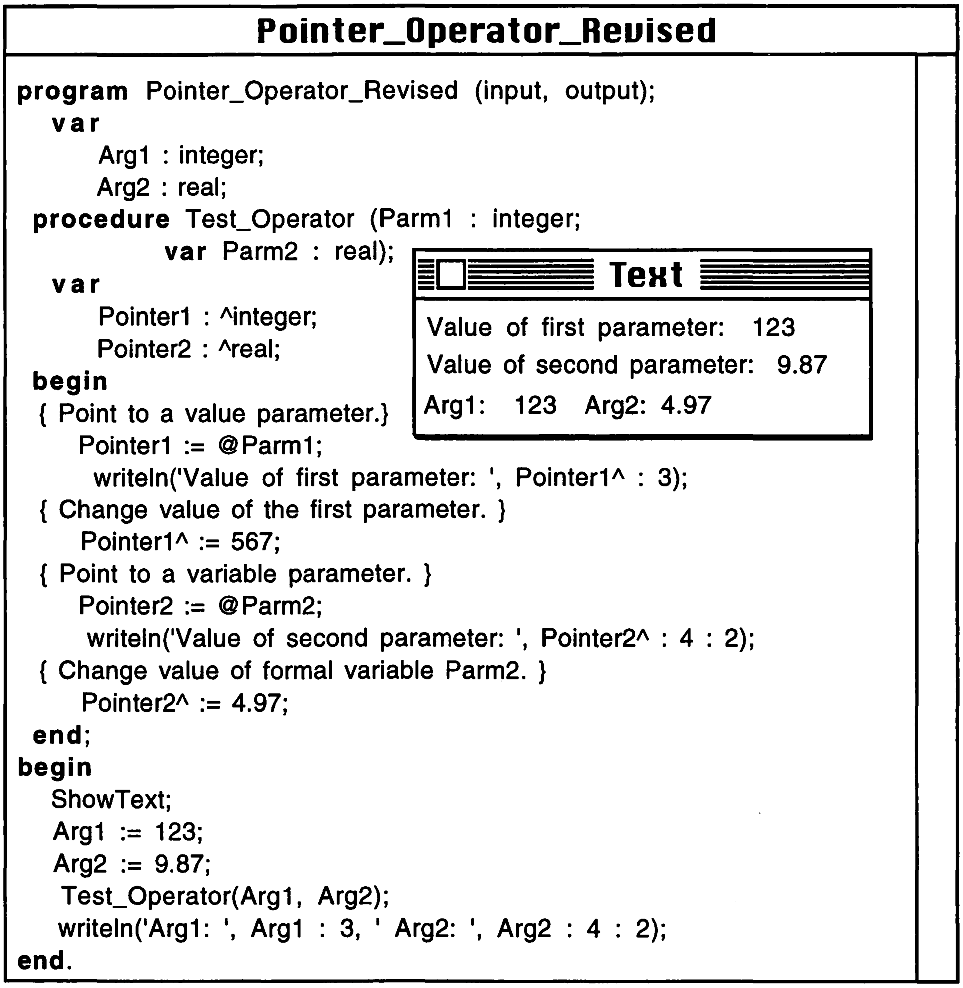
The operator @ can also be applied to value and variable parameters. The Text window for `Pointer_Operator_Revised` in Figure 12.5 shows examples that apply the operator @ to two formal parameters. Notice that the assignment statement

```
Pointer1 := @Parm1;
```

assigns the stack location in memory where the value of the actual parameter `Arg1` has been passed. This location is only in effect as long as the procedure is executing. Assigning a new value to `Pointer1^` changes only the value pointed to by `Pointer;` it has no effect on the value of `Arg1`. This is different for `Parm2`. Here the assignment statement

```
Pointer2 := @Parm2
```

assigns the memory location of `Arg2` rather than the memory location of `Parm2` on the stack. In this case a new value assigned to `Pointer2^` has an additional effect on `Parm2`: It changes the value of the actual parameter `Arg2`. This example reinforces the concept that a value parameter acts like a local variable with an initial value passed to it by a corresponding actual parameter, whereas a formal variable parameter has the memory address of a corresponding actual parameter.



```
Pointer_Operator_Revised

program Pointer_Operator_Revised (input, output);
  var
     Arg1 : integer;
     Arg2 : real;
 procedure Test_Operator (Parm1 : integer;
            var Parm2 : real);
  var
     Pointer1 : ^integer;
     Pointer2 : ^real;
 begin
 { Point to a value parameter.}
    Pointer1 := @Parm1;
     writeln('Value of first parameter: ', Pointer1^ : 3);
 { Change value of the first parameter. }
    Pointer1^ := 567;
 { Point to a variable parameter. }
    Pointer2 := @Parm2;
     writeln('Value of second parameter: ', Pointer2^ : 4 : 2);
 { Change value of formal variable Parm2. }
    Pointer2^ := 4.97;
 end;
begin
  ShowText;
  Arg1 := 123;
  Arg2 := 9.87;
   Test_Operator(Arg1, Arg2);
   writeln('Arg1: ', Arg1 : 3, ' Arg2: ', Arg2 : 4 : 2);
end.
```

```
Text

Value of first parameter:  123
Value of second parameter:  9.87
Arg1:  123  Arg2: 4.97
```

Figure 12.5 Screen dump for the program `Pointer_Operator_Revised` and its output to the Text window.

Although we can apply the @ operator to procedure and function names, Macintosh Pascal provides no mechanisms for using such pointers, except for passing them to a library procedure.

Both `new` and `dispose` are important to the implementation of abstract data types. `New` acts as a natural constructor operator, because it creates an object having the properties of an abstract data type; `dispose` acts as a natural destructor by eliminating an object.

12.3 SPECIAL DATA STRUCTURES: LINKED LISTS, STACKS, AND QUEUES

Let us consider a modification of the program Pointer_to_Employee. The core of the modification is an abstract data type called a *list* for the purpose of accepting and holding employee information as it is typed at the keyboard. This action requires a constructor for creating a new element for the list, a modifier for adding a new element representing information on an employee to the front of the list, and an iterator for viewing information on all of the employees contained within the list.

One solution to this problem is to implement the list as an array of employee records. Although this solution seems simple, it can waste memory if we only need a few records and we choose a large upper bound for the size of the array. In addition, how can we eliminate an employee record from the array when a particular person is no longer employed? To resolve this problem, we could use a second list to hold index positions of all array elements representing persons no longer employed. When we need to add a new employee to the list, the second list would be checked to determine the next free array element. This seems to solve this minor problem, but we must now store two arrays, both having the same number of elements. Another problem occurs if the employee list is filled and we need to add another employee. This would require re-editing the source program and retranslation to have a larger array for storing employees.

A second solution is to write the contents of each employee record directly to a sequential file. This could be viewed as an unbounded array, where an employee could always be merged into the file. Searching for an employee record, however, can require prolonged execution times when appending an employee record either to the beginning or end of a lengthy sequential file. Again, if a person is no longer employed, can we write an efficient algorithm to remove employee information from the sequential file?

A third solution is to keep the list of employees in RAM by linking each employee record by using pointers. With this approach, searching the employee list is generally faster than searching through a sequential file stored on a diskette, and keeping the employee list as a linked list of records reduces execution time for adding and deleting employee records. In practice we would still need a file for backup storage of the employee list.

To build a linked list of employees, consider the following modification to the record type called Profile:

```
type
   Node    = ^Profile;
   Profile =   record
                  Name : string;
                  ID_Number : string;
                  Wage : real;
                  Link : Node
               end;
var
   Employee : Node;
   List_Head : Node;
```

The definitions are recursive, because the type called Node represents a pointer to an object called Profile, with the field Link of type Profile itself being of type Node, that is, a pointer to an object called Profile. Employee is now a pointer type declared as type Node. A new variable called List_Head is added. This special pointer variable is used to point to the beginning of our employee list. Within this special node called List_Head is a link field containing the value **nil** if the employee list is empty. If the employee list is not empty, the link field will contain a pointer pointing to an employee node at the front of the linked list. Figure 12.6 shows an abstraction of the concepts of Node and List_Head.

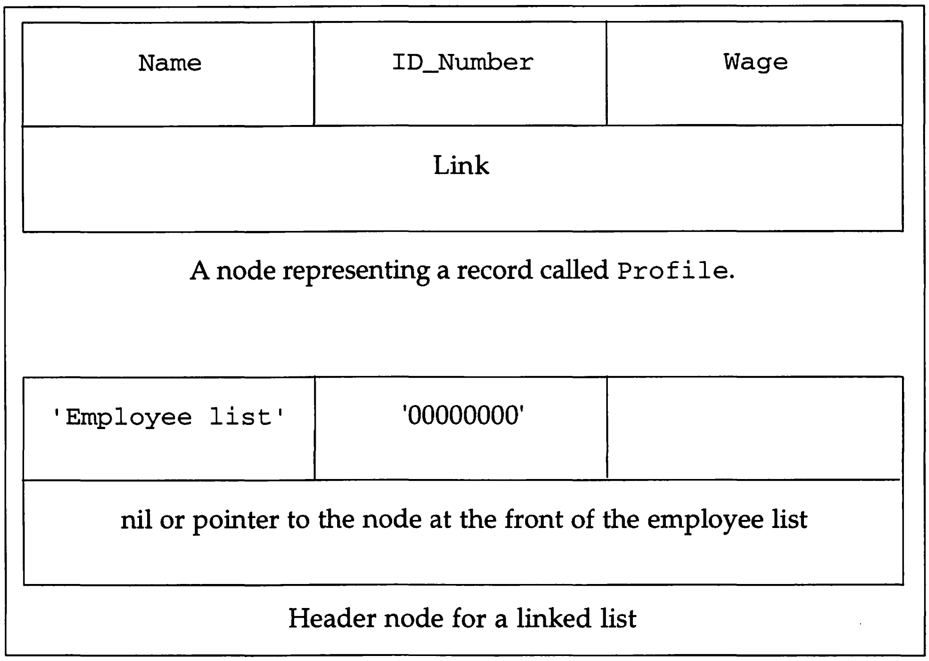

Name
ID_Number
Wage
Link
A node representing a record called Profile.
'Employee list'
'00000000'
nil or pointer to the node at the front of the employee list
Header node for a linked list

Figure 12.6 Abstraction representing the concepts of Node and List_Head.

Assuming that a header node exists, Figure 12.7 shows the steps for modifying the state of a linked list by adding a new node to the beginning of the list. To accomplish this task, a new node is created, with the pointer Employee pointing to this node.

The link field of the new node pointed to by Employee is then assigned the link field of the header node, so that the link field of our new node points to the remainder of the linked list, ensuring that a pointer to the front node of the present employee list is not lost. The link field of the header node referred to as List_Head is now assigned the pointer of Employee, so that the link field of the header node points to the new node Employee. The result is that the new node has been inserted at the beginning of a linked list with the header node pointing to the new node and the link field of Employee pointing to the remainder of the linked list. If the linked list is initially empty, the link field of the header node is initially **nil**. On adding the first node to the linked list, the link field of Employee is assigned the value **nil**. As other nodes are added, the last node of the linked list will always have a link field value that is **nil**, and the link field of the header is always pointing to the front of the linked list. By having the link field of the last node **nil**, we can see where the list terminates. The steps needed for entering employee records from the keyboard and adding them to the linked list of records follow.

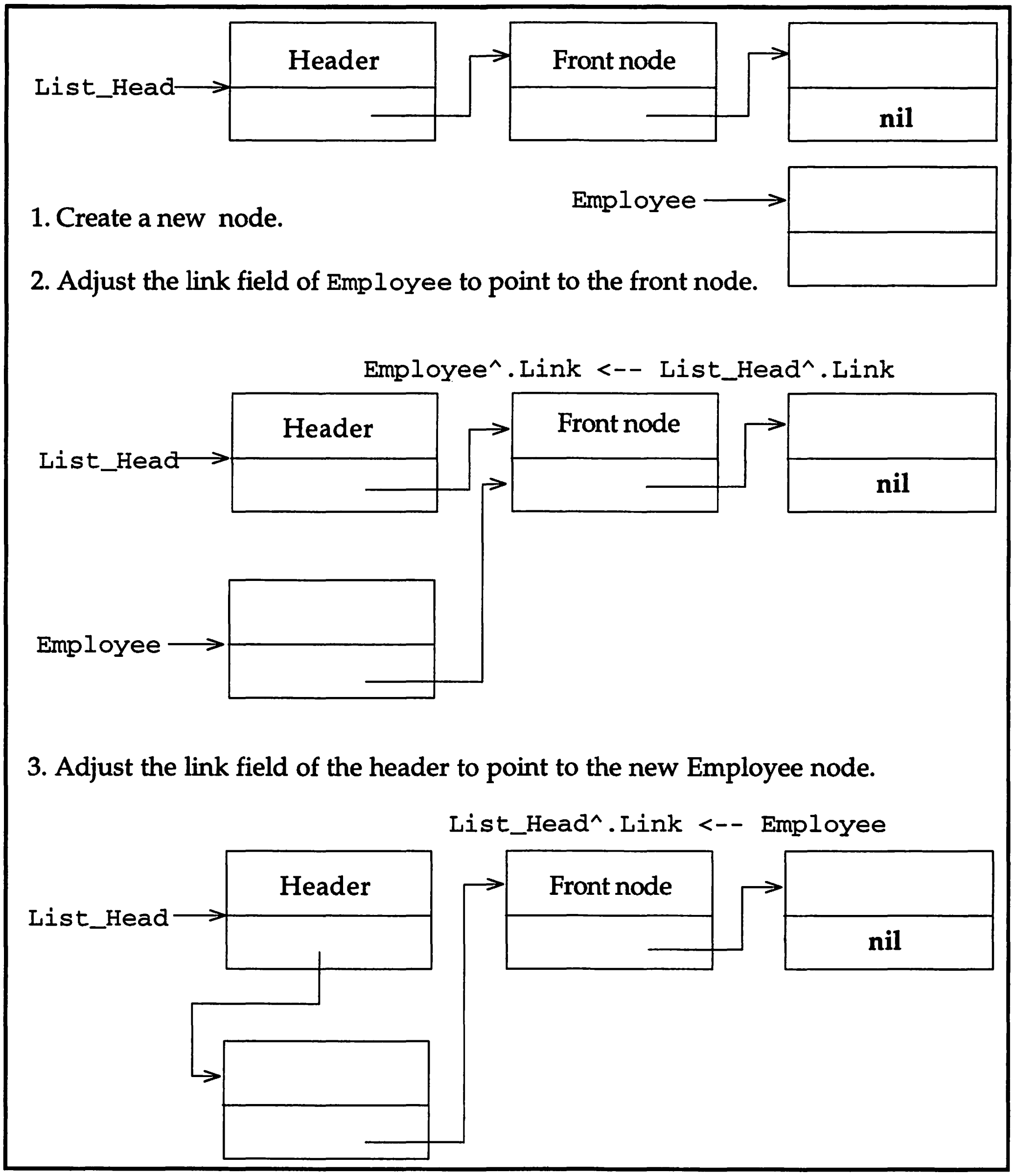

Header
Front node
List_Head
nil
Employee
1. Create a new node.
2. Adjust the link field of Employee to point to the front node.
Employee^.Link <-- List_Head^.Link
3. Adjust the link field of the header to point to the new Employee node.
List_Head^.Link <-- Employee

Figure 12.7 Adding a new node to the beginning of a linked list.

```
begin
{ Initialize the pointer List_Head. }
   Initialize_Header( List_Head );
{ Enter employee records from the keyboard. }
   write( ' Enter another employee? Type Y for yes, N for no: ');
   readln( Response );
   while Response = 'Y' do
```

```
      begin
      { Create new employee record. }
         new( Employee );
      { Enter information into new employee record. }
         Enter_Data( Employee );
      { Link this new node to beginning of the employee list. }
         Employee^.Link := List_Head^.Link;
         List_Head^.Link  := Employee;
      { Prompt user for another record. }
         write('Enter another employee? Type Y for yes, N for
                 no:');
         readln( Response );
      end; { while-do }
{ Report records to the Text window. }
      Report_Records( List_Head );
end.
```

The linked list shown in Figure 12.7 is often referred to as a *singularly linked list*, because the link field of each node points only to its successor node and never to any of its predecessor nodes. For reporting the employee records as an iterative operation, the following steps are required:

1. First, test if the employee list is empty.
2. If empty, report that there are no records to be displayed.
3. If not empty, initialize a temporary pointer by pointing to the front node of the employee list.
4. Report the fields of the present node.
5. Now assign to the temporary pointer the link field of the present node that it is pointing to, and if this value is not **nil**, repeat Steps 4 and 5.

Here is the complete THINK Pascal program.

```
program Employee_List(input, output);
{ Purpose:  This program creates a linked list of records. }
{           Add the following uses clause for Macintosh Pascal:}
{  uses                                                 }
{     QuickDraw1;                                       }
   type
      Node = ^Profile;
      Profile =   record
                     Name : string;
                     ID_Number : string;
                     Wage : real;
                     Link : Node
                  end;
   var
      List_Head, Employee : Node;
      Response : char;
{ ******************************************************** }
   procedure Initialize_Header (var Header : Node);
   {Purpose:   Create a header node for a linked list. }
   begin
      new(Header);
      Header^.Name := 'Employee List';
      Header^.ID_Number := '00000000';
      Header^.Link := nil
   end;
{ ********************************************************* }
   procedure Enter_Data (Person : Node);
   {Purpose:   This routine reads data from the keyboard and }
   {           inserts this data into an object called a node.}
   begin
      writeln;
      with Person^ do
         begin
            write(' Enter full name: ');
            readln(Name);
            write(' Enter ID number: ');
            readln(ID_Number);
            write(' Enter hourly wage: ');
            readln(Wage);
         end;
   end;
{ ********************************************************* }
   procedure Modify_List( Header, Employee : Node);
   {Purpose:   This routine links a node to the beginning of a }
   {           linked list.}
   begin
         Employee^.Link := Header^.Link;
         Header^.Link := Employee;
   end;
{ ********************************************************* }
   procedure Report_Records (Header : Node);
   {Purpose:   Display of content in each node of a linked list. }
      var
         P : node;
   begin
   { Test if the employee list is empty. }
      if Header^.Link = nil then
         begin
            write(' Employee list is empty --' );
            writeln( ' no records presently exist.');
         end
      else
      { Continue to report the contents of each node in the }
      { employee list. }
      begin
      { Let the temporary pointer point to first node in list. }
         P := Header^.Link;
         repeat
            writeln;
            with P^ do {Pointer P points to node for reporting.}
```

```
                 begin
                    writeln(' Name: ', Name);
                    writeln(' ID number: ', ID_Number);
                    writeln(' Hourly wage: $', Wage : 5 : 2);
                 end;
              { Let pointer P point to the next node in the }
              { employee list. }
              P := P^.Link;
        until P = nil;
     end;
  end;
{ ******************************************************* }
begin { Body of the main program. }
{ Show Text window for prompts. }
  ShowText;
{ Initialize the pointer List_Head. }
  Initialize_Header(List_Head);
{ Enter employee records from the keyboard. }
  write(' Enter another employee? Type Y for yes, N for no: ');
  readln(Response);
  writeln;
  while Response = 'Y' do
     begin
     { Create new employee record. }
        new(Employee);
     { Enter information into new employee record. }
        Enter_Data(Employee);
     { Link this new node to beginning of the employee list. }
        Modify_List( List_Head, Employee );
     { Prompt user for another record. }
        writeln;
        write('Enter another employee? Type Y for yes,');
        write(' N for no: ');
        readln(Response);
     end; { while-do }
{ Report records to the Text window. }
  Report_Records(List_Head);
end.
```

The procedure Initialize_Header acts a constructor by creating and initializing the header node, while procedure Report_Records acts an iterator for viewing the content of the linked list. Procedure Modify_List is a two-step routine acting as a modifying operation of the list. Notice that some of the formal parameters in procedure headers are variable types, whereas others are value types even when side effects take place on the objects pointed to by the parameters. Procedure Initialize_Header requires that the formal parameter Header be a variable type, because a header node is created during execution of this procedure; the parameter Header points to this new object in memory, and the address pointed to by the formal parameter is changed during the execution of this procedure. In the procedure Enter_Data, the formal parameter Person is a value type, even though the fields of the node of Person are modified. It is a value type rather than a variable type because the address associated with `Person` remains unchanged.

An additional modification operation for this program is a procedure for deleting an employee record from the list. This requires that we search through the employee list with a key until we either find the node containing the key and mark it for deletion or reach the last node of the linked list. Once the node is found, the link field of the preceding node is adjusted so that it points to the link field for the node being deleted. The employee node is then deleted. Figure 12.8 shows the steps required for disposing of a node from a linked list. Notice that we need two pointers: one points to the node preceding the node to be disposed of, and a second points to the node for deletion. We need two pointers because the link field of each node points only to its successor node. There is no link field pointing to a predecessor node. Because we cannot reference the predecessor node of the node being deleted, it is impossible to modify the link field of the predecessor node by having it point to the successor node of the node being deleted. The following is a refinement of the steps for deleting a node from a singularly linked list.

```
procedure Delete_Record( Header : Node; Key : string );
{Purpose: On finding a node containing the value of Key, the }
{         node is marked by a pointer and deleted from the }
{         linked list.}
   var
      P, Q : Node;
      Found : Boolean;
begin
{ Initialize pointer Q. }
   Q := Header;
   Found := false;
   while ( Q^.Link <> nil) and ( not Found ) do
      begin
      { Let P point to the succeeding node. }
         P := Q^.Link;
      { Check if the succeeding node has the key. }
         if P^.Name = Key then  { delete node pointed to by P }
            begin
               Found := true;
            { Adjust the link field of Q to point to the link }
            { field of P. }
               Q^.Link := P^.Link;
            { Deallocate the node from the employee list. }
               dispose(P)
            end
         else  { let Q point to the succeeding node }
            Q := P
      end;
end;
```

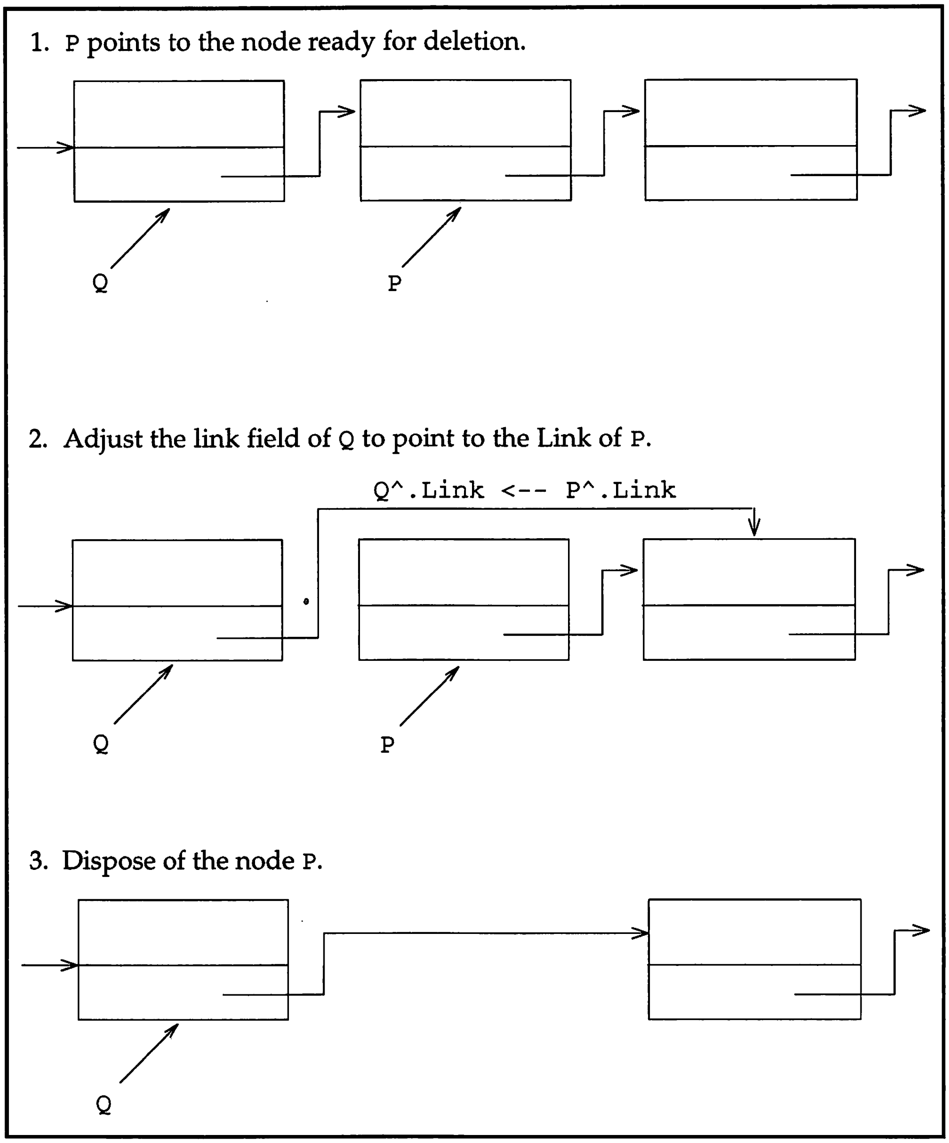

1. P points to the node ready for deletion.
Q
P
2. Adjust the link field of Q to point to the Link of P.
Q^.Link <-- P^.Link
Q
P
3. Dispose of the node P.
Q

Figure 12.8 Disposing of a node from a linked list.

In the procedure `Delete_Record`, the formal parameter `Header` is a value type. Even though the linked list pointed to by `Header` can be modified during execution, the address pointed to by `Header` remains unchanged during the execution of the procedure. It is the nodes within the linked list that may be deleted, not the node representing the header. Furthermore, this procedure conserves memory for heap storage by executing the command `dispose` for any node that is no longer needed.

Another abstract data type of interest is the stack. A stack is a finite sequence of elements all of the same type, having the following operations:

Constructor:
    Create the structure representing a stack.
    Initialize a stack as being empty.
Destructor:
    Destroy the structure representing a stack.
Modifier:
    Insert a new element at one end of the stack, called its *top*.
    Delete an element from the top of the stack.
Selector:
    Determine if a stack is full.
    Determine if a stack is empty.
Iterator:
    View the element at the top of the stack.

The stack is a special structure where data objects are inserted (pushed on) and removed (popped off) at only one end. In this context a stack is referred to as a *last-in, first-out* structure. Figure 12.9 shows a simple model of the stack. Objects are pushed and popped at the top of the stack; the top is determined by a stack pointer. We can emulate stacks by one of two approaches: an array or a linked list.

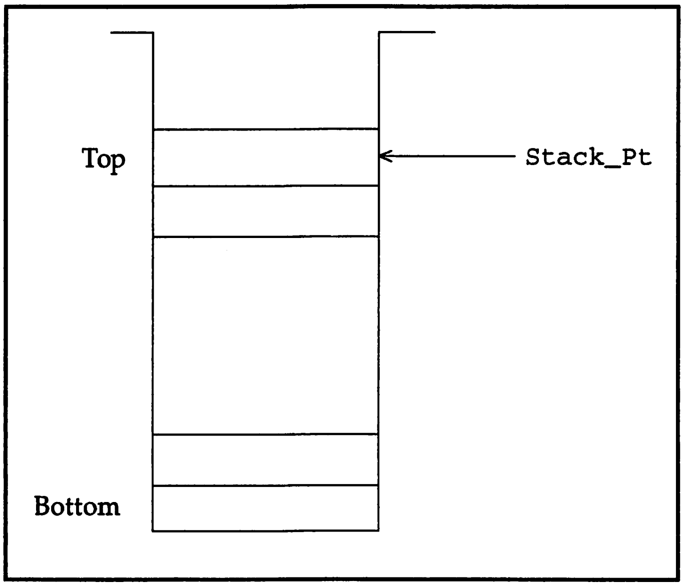


Figure 12.9 A simple model representing a data object called a *stack*.

To emulate a stack using an array, we can use the following programmer-defined type:

```
const
   N = 100;
type
   Stack =  record
               Stack_Pt : 0. . N;
```

```
                Stack_Table : array[1.. N] of Item
             end;
var
   S : Stack;
```

The type `Stack` represents our model of a stack as an array of 100 elements indexed from 1 through *N*. The following assumptions are defined as exceptions:

1. An empty stack is represented by assigning `Stack_Pt` a value of zero.
2. Other than for an empty stack, `Stack_Pt` points to the last item pushed onto the top of the stack.
3. The stack is full when the value of `Stack_Pt` has the value of *N*.

Variables of type `Stack` have storage allocated when the program unit in which they are declared is executed, so the procedure that follows defines both the constructor and destructor operation for initializing the state of a stack as empty.

```
procedure Create_Stack( var S : Stack );
begin
   S.Stack_Pt := 0;
end;
```

Selector operations for testing if a stack is empty or full are defined by the following two `Boolean` functions. These functions have simple definitions, because the stack pointer has either the value zero if the stack is empty or the value of *N* if the stack is full.

```
function Empty_Stack( S : Stack ): Boolean;
begin
   if S.Stack_Pt = 0 then
      Empty_Stack := true
   else
      Empty_Stack := false;
end;
```

```
function Full_Stack( S : Stack ): Boolean;
begin
   if S.Stack_Pt = N then
      Full_Stack := true
   else
      Full_Stack := false;
end;
```

There are two modifying operations for changing the state of the stack. The first modifier, `Push_Stack`, inserts a new element at the top of the stack. In defining this routine, we must first determine if the stack is full. Any attempt to insert an element onto the stack when it is full produces a condition referred to as *stack overflow*. The following represents the routine as a function returning a `Boolean` value of *true* if an element is successfully inserted at the top of a stack and *false* if stack *overflow* occurs.

```
function Push_Stack ( var S : Stack; Element : Item ): Boolean;
begin
{ Test for stack overflow. }
   if  Full_Stack(S) then
      Push_Stack := false
   else
   { Adjust the stack pointer and insert an element at the }
   { top of the stack. }
      begin
         S.Stack_Pt := succ(S.Stack_Pt);
         S.Stack_Table[S.Stack_Pt] := Element;
         Push_Stack := true;
      end;
end;
```

The second modifier, Pop_Stack, allows a new element to be popped from the top of the stack. In defining this routine, we must first determine if the stack is empty. Any attempt to delete an element from a stack when it is empty represents a condition referred to as *stack underflow*. The following represents the routine as a function returning a Boolean value of *true* if an element is successfully deleted from a stack and *false* if stack *underflow* occurs.

```
function Pop_Stack( var S : Stack ): Boolean;
begin
{ Test for stack underflow. }
   if Empty_Stack(S) then
      Pop_Stack := false
   else
   { Delete the top element by adjusting the stack pointer. }
      begin
         S.Stack_Pt :=  pred(S.Stack_Pt);
         Pop_Stack := true;
      end;
end;
```

The iterator operator for viewing the value at the top of the stack also depends upon the stack not being empty. If it is empty, no value can be returned by the operation. The following function defines the steps for retrieving the value at the top of stack.

```
function Retrieve_Element( S : Stack; var Element : Item ):
                              Boolean;
begin
{ Test for stack underflow.}
   if Empty_Stack(S) then
      Retrieve_Element := false
   else
{ Retrieve the top element from stack S.}
      begin
         Element :=  S.Stack_Table[S.Stack_Pt];
         Retrieve_Element := true;
```

```
        end;
end;
```

We use `Boolean` functions for the various stack operations rather than procedures because they give us more control over the next step of the program. Given that the stack type and operations are encapsulated into a unit, a programmer using this unit can decide how the program is to recover in case of a stack overflow or underflow. This allows the avoidance of useless message displays and allows the programmer to dictate the next state of the program.

A second approach to emulating a stack is to use a linked list. There are several reasons why this approach is useful. First, there is no need in principle to test for a full stack, because we can always add an element to the top of the stack. Second, storage is limited to only those elements pushed onto the stack; maximum storage for the stack is whatever is allowed for the heap. Figure 12.10 shows a linked-list representation of a stack.

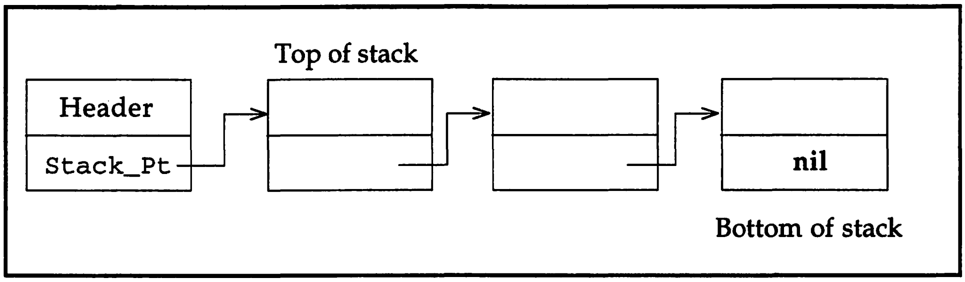


Figure 12.10 A linked list emulating a stack.

The following are the types and modules needed to represent a stack using a linked list. These are again given as `Boolean` functions for the purpose of compatibility with any program using an array as a representation of a stack:

```
type
   Pointer = ^Node;
   Node  =  record
               Value : Item;
               Link : Pointer
            end;
   Stack = Pointer;
var
   S : Stack;
```

```
procedure Create_Stack( var S : Stack );
begin
{ Create a list head for the stack. }
   new( S );
{ Empty stack is represented by a link field of the header being }
{ nil.}
    S^.Link := nil
end;
```

```
function Empty_Stack( S : Stack ): Boolean;
begin
   if S^.Link = nil then
      Empty_Stack := true
   else
      Empty_Stack := false;
end;
```

```
function Full_Stack( S : Stack ): Boolean;
begin
   Full_Stack := false;
end;
```

```
function Push_Stack( S : Stack; Element : Item ): Boolean;
   var
      P : Pointer;
begin
{ Create a new node and link it to the top of the stack. }
   new( P );
   P^.Value := Element;
{ Let the link field of P point to the node given by the link }
{ field of S. }
   P^.Link := S^.Link;
{ Let the header point to the new node pointed to by P. }
   S^.Link := P;
   Push_Stack := true;
end;
```

```
function Pop_Stack( S : Stack ): Boolean;
   var
      P : Pointer;
begin
{ Check for an empty stack. }
   if Empty_Stack(S) then
      Pop_Stack := false
   else
      begin
      { Let pointer P point to the node for deletion.}
         P := S^.Link;
      { Adjust link field of the header with the link field of }
      { pointer P. }
         S^.Link := P^.Link;
      { Deallocate the node pointed to by P. }
         dispose( P );
         Pop_Stack := true;
      end
end;
```

```
function Retrieve_Element( S : Stack; var Element : Item ):
Boolean;
begin
{ Test for stack underflow.}
      if Empty_Stack(S) then
         Retrieve_Element := false
      else
      { Retrieve the value from the top element of the stack.}
         begin
            Element :=  S^.Link^.Value;
            Retrieve_Element := true;
         end;
end;
```

One procedure that is not yet defined is a destructor for abrogating the object representing a stack. This requires that the stack be emptied (deleted of all of its nodes) before disposing of the header node of the stack. By doing this, we avoid garbage when the stack is eliminated. The following routine provides the steps for deallocating a stack:

```
procedure Deallocate_Stack( var S : Stack );
begin
   while Pop_Stack(S) do
      { nothing };
   dispose(S);
end;
```

A queue is a third abstract data type. Data objects of this type have elements that are inserted (pushed on) at one end and deleted (popped off) at the other. A queue is referred to as a *first-in, first-out* structure. Figure 12.11 shows a model of a queue.

As a finite sequence of elements of the same type, a queue, as an abstract data type, has the following operations:

Constructor:
- Create the structure representing a queue.
- Initialize a queue as empty.

Destructor:
- Destroy the structure representing a queue.

Modifier:
- Insert a new element at the rear of the queue.
- Delete an element from the front of the queue.

Selector:
- Determine if a queue is full.
- Determine if a queue is empty.

Iterator:
- View the element at the front of the queue.

When developing algorithms to perform these basic operations, we need to consider the following assumptions and questions:

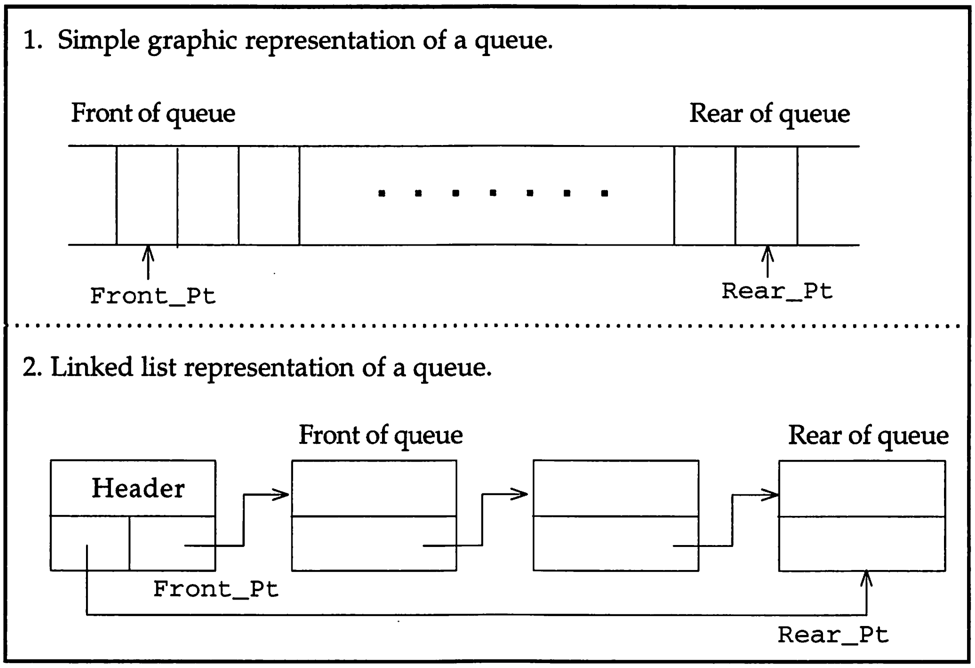


Figure 12.11 Models representing a data object called a queue.

Assumptions

1. The front pointer is assumed to point to the next element to be popped or to the position of the last element that was deleted.
2. The rear pointer points to the next position where an element can be inserted or to the last position where an element was inserted.
3. A queue is empty when the front pointer catches up with the rear pointer.
4. A queue is full when the rear pointer catches up with the front pointer.
5. When an element is pushed, the rear pointer is incremented.
6. When an element is popped, the front pointer is incremented.
7. If an array with *n* elements is used to emulate a queue, the queue should be considered circular, with the $(n-1)$th position of the array next to the 0th position.
8. If a linked list is being used to emulate a queue, the queue header will contain the front and rear pointers, and the nodes representing elements of the queue will need only a link field.

Questions

1. If you are emulating a queue with an array, what is the size of the queue?
2. How is a queue tested for being empty and for being full?

3. How is a queue created or deleted?

The use of an array to emulate a queue is left as programming exercise. What follows is an emulation of a queue by means of a linked list. In using a linked list, we assume that the front pointer points to the next element ready for deletion, and the rear pointer points to the last element that was inserted.

```
type
   Pointer = ^Node;
   Node =   record
               Value : Item;
               Link : Pointer
            end;
   Queue = ^Header;
   Header = record
               Title : string;
               Front_Pt : Pointer;
               Rear_Pt : Pointer
            end;
var
   Q : Queue;
```

In this application a queue is considered empty if both the front and rear pointers are **nil**. Here is a constructor for creating a queue where both the rear and front pointers are **nil**.

```
procedure Create_Queue( var Q : Queue );
begin
   new(Q);
   Q^.Title := 'List head for queue';
   Q^.Front_Pt := nil;
   Q^.Rear_Pt := nil
end;
```

The following are selectors for testing if a queue is empty or full. We are using a linked list to represent a queue, so we assume that the function `Full_Queue` will always be *false*.

```
function Empty_Queue( Q : Queue ): Boolean;
begin
   if Q^.Rear_Pt = nil then
      Empty_Queue := true
   else
      Empty_Queue := false;
end;
```

```
function Full_Queue( Q : Queue ): Boolean;
begin
   Full_Queue := false;
end;
```

Inserting (pushing) a new element onto the queue requires the creation of a new node, adjusting the rear pointer, and, if the queue is initially empty, adjusting the front pointer. Two local pointers are required for adjusting link fields.

```
function Push_Queue( Q : Queue ; Element : Item ): Boolean;
   var
      P, T : Pointer;
begin
{ Create a new node for inserting an element. }
   new( P );
   P^.Value := Element;
   P^.Link := nil;
{ After checking for an empty queue, insert this new node at the }
{ rear of, the queue. }
   if Empty_Queue(Q) then { queue is empty }
      begin
         Q^.Rear_Pt := P;
         Q^.Front_Pt := P
      end
   else
      begin { nonempty queue }
         T := Q^.Rear_Pt;
         T^.Link := P;
         Q^.Rear_Pt := P
      end;
   Push_Queue := true;
end;
```

In deleting an element from the queue, we test the queue to see if it is empty. If not, the value of the element to be deleted is copied, and a test is made to see if the front and rear pointers are pointing to the same node. If they are, both pointers are assigned **nil** values. If they are not, only the front pointer is adjusted.

```
function Pop_Queue( Q : Queue ): Boolean;
   var
      P : Pointer;
begin
{ Test for an empty queue. }
   if Empty_Queue(Q) then { Report queue is empty. }
      Pop_Queue := false
   else
      begin
      { Set pointer to the node for deletion. }
         P := Q^.Front_Pt;
      { Test if both front and rear pointers point to P. }
         if Q^.Front_Pt = Q^.Rear_Pt then
            begin
            { Re-initialize the pointers as an empty queue since }
            { the front pointer has caught the rear pointer. }
            Q^.Front_Pt := nil;
            Q^.Rear_Pt := nil
         end
      else { Adjust the front pointer. }
         Q^.Front_Pt := P^.Link;
   { Deallocate the node pointed to by P. }
      dispose( P );
      Pop_Queue := true;
   end;
end;
```

```
function Retrieve_Element( Q : Queue; var Element : Item ):
                                       Boolean;
begin
{ Test for stack underflow.}
   if Empty_Queue(Q) then
      Retrieve_Element := false
   else
   { Retrieve the value of the front element of the queue.}
      begin
         Element :=  Q^.Front_Pt^.Value;
         Retrieve_Element := true;
      end;
end;
```

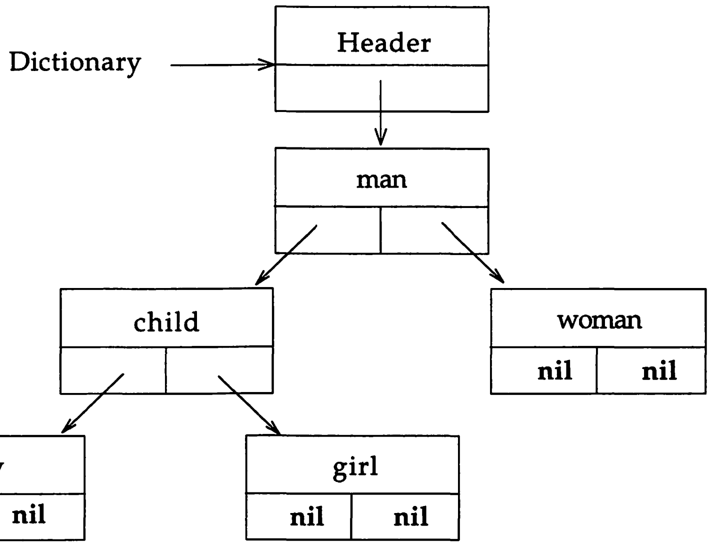
```
procedure Deallocate_Queue( var Q : Queue );
begin
   while Pop_Queue(Q) do
      { nothing };
   dispose(Q);
end;
```

12.4 APPLICATION OF POINTERS: BINARY TREES

There are numerous applications for pointers and linked lists. A common application is the "growing" and "pruning" of binary trees. A binary tree is an abstract data type. An object of this type is either empty or composed of a "root" node acting as a parent, and two binary trees called a *left binary subtree* (*left descendant*) and a *right binary subtree* (*right descendant*).

When a binary tree is implemented as an object, the root and each subtree are represented by a node having left and right link fields. In turn each subtree of a parent node can point to a descendant, a node containing a left link and right link field. When both the left link field and right link field of a node are **nil**, the node is referred to as a *leaf node*. One interesting application of a binary tree is in storing a dictionary. Having an on-line dictionary allows us to check the spelling in a text document.

Our dictionary is represented by a binary tree. Initially, we assume the tree to be empty except for a header node. As Figure 12.12 shows, the first word to be entered into the dictionary establishes the root node. Other words are added as either left or right descendants of the binary tree.



Order in which words are inserted: man, woman, `child`, boy, `girl`.

Left link of node man is the node containing the word `child`.
Right link of the node man is the node containing the word woman.

Left link of node `child` is the node containing the word boy.
Right link of the node `child` is the node containing the word `girl`.

The nodes boy, `girl`, and woman are leaf nodes (terminal nodes) because both the left and right links of each node are nil.

Figure 12.12 Binary tree representing a simple dictionary.

Here is a list of the necessary steps for inserting one or more words into the dictionary, assuming the existence of a pointer `P` initially pointing to the root of the dictionary:

1. If `P` is **`nil`**, create a new node and insert the word into this new node.
2. Otherwise, if the word is alphabetically less than the value field of the node pointed to by `P`, attempt to insert the word on the left side of `P` by passing to `P` the left link of `P` and repeating this algorithm.
3. Otherwise, if the word is alphabetically larger than the value field of the node pointed to by `P`, attempt to insert the word on the right side of `P` by passing to `P` the right link of `P` and repeating this algorithm.
4. Otherwise, the word already exists in the dictionary; there is no need for insertion.

We can now extend this example by creating the following programmer-defined types for our dictionary problem:

```
type
   Pointer = ^Node;
   Node =   record
               Value : string;
               Left_Link : Pointer;
               Right_Link : Pointer
            end;
   Header_Node =  record
                     Title : string;
                     Link : Pointer
                  end;
   Tree = ^Header_Node;
var
   Dictionary : Tree;
```

Initially we need a module for creating the header of the dictionary, given in the following procedure.

```
procedure Create_Dictionary_Header( var Header : Tree );
begin
   new( Header );
   Header^.Title := 'Sample Dictionary';
   Header^.Link  := nil
end;
```

The module for inserting a new word follows the three basic steps just discussed. The procedure for inserting a new word is as follows:

```
procedure Insert_Word( var P : Pointer; Word : string );
begin
{ Test if pointer P is nil. }
   if P = nil then
   { Add word to the dictionary. }
      begin
         new( P );
         with P^ do
            begin
               P^.Value  :=  Word;
               P^.Left_Link  :=  nil;
               P^.Right_Link  :=  nil
            end
      end
   else { Test if word is to be inserted to the left of node P. }
      if Word < P^.Value then
         Insert_Word( P^.Left_Link, Word )
      else { Test if word is to be inserted to right of node P. }
         if Word > P^.Value then
```

```
            Insert_Word( P^.Right_Link, Word)
         else { Word already exists in the dictionary. }
            writeln(' The word', Word, ' is already in the
                         dictionary.' );
end;
```

Notice that this procedure is recursive. It calls on itself to insert the word either to the left or to the right of the root node. Recursion stops when it finds a left link or right link field that is **nil** or when the value of Word is equal to that of P^.Value.

To display the words from our dictionary in alphabetic order, we must perform an *inorder* traversal of the binary tree. Traversing a binary tree is equivalent to walking through the paths of the binary tree. An *inorder* traversal of a binary tree is represented by the following recursive steps:

1. Traverse the left link of the binary tree of the present node in *inorder.*
2. Visit the present node by displaying the contents of the value field.
3. Traverse the right link of the binary tree of the present node in *inorder.*

We have reached the end of a link in a binary tree when either the left link field or the right link field of a node has a pointer that is **nil**. The following procedure refines these steps.

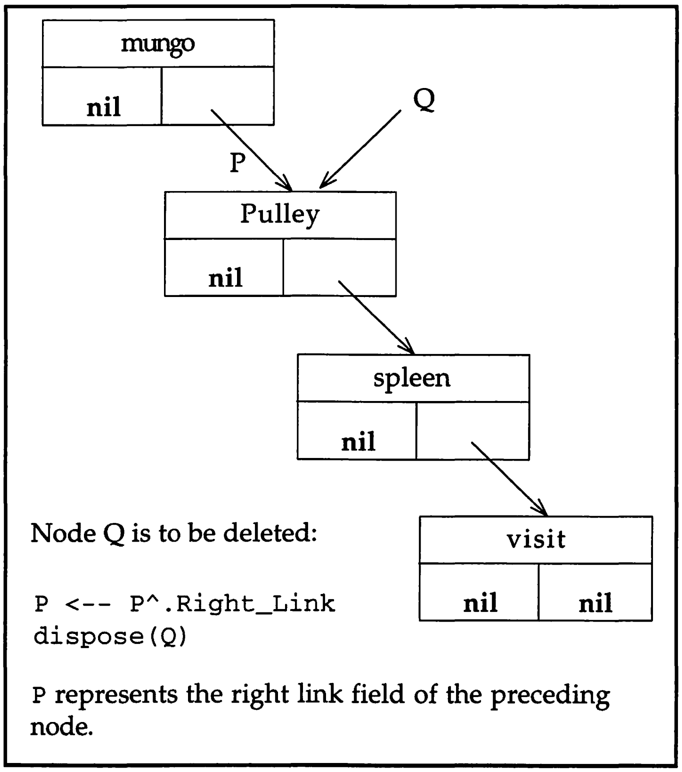
```
procedure Print_Word( P : Pointer );
begin
{ Test if node P is not nil. }
   if P <> nil  then
      begin
      { Traverse in inorder the left link of P.  }
         Print_Word( P^.Left_Link );
      { Visit the node. }
         writeln( P^.Value );
      { Traverse in inorder the right link of P. }
         Print_Word( P^.Right_Link )
      end
   else { reached a link that is nil }
end;
```

Dictionary_Example is a sample program for testing these modules.

```
program Dictionary_Example(input, output);
{ Purpose:  This is a simple dictionary program. Words can be }
{           inserted into the dictionary, and the dictionary is }
{           displayed after each word is inserted. }
   uses
      QuickDraw1;
   type
      Pointer = ^Node;
      Node  =  record
                  Value : string;
                  Left_Link : Pointer;
                  Right_Link : Pointer
               end;
      Header_Node =  record
                        Title  : string;
                        Link : Pointer
                     end;
      Tree = ^Header_Node;
   var
      Dictionary : Tree;
      Word  : string;
      Response : char;
{ ********************************************************* }
   procedure Create_Dictionary_Header  (var Header  :  Tree);
   begin
      new(Header);
      Header^.Title := 'Sample Dictionary';
      Header^.Link  := nil
   end;
{ ********************************************************* }
   procedure Insert_Word  (var P  :  Pointer; Word  :  string);
   begin
   { Test if pointer P is nil. }
      if P = nil then { Add word to the dictionary. }
         begin
            new(P);
            with P^ do
               begin
                  Value := Word;
                  Left_Link  := nil;
                  Right_Link  := nil
               end
         end
      else { Test if the word is to be inserted to the left }
            { of node P. }
         if Word < P^.Value then
            Insert_Word(P^.Left_Link, Word)
         else { Test if the word is to be inserted to the }
               { right of node P. }
            if Word > P^.Value then
               Insert_Word(P^.Right_Link, Word)
            else { Word already exists in the dictionary. }
               begin
                  write(' The word ', Word, ' is already in');
                  writeln(' the dictionary.');
               end;
   end;
{ ********************************************************* }
   procedure Print_Word  (P  :  Pointer);
   begin
   { Test if node P is not nil. }
      if P <> nil then { traverse the binary tree in inorder }
         begin
            Print_Word(P^.Left_Link);
            write(P^.Value, ' ');
            Print_Word(P^.Right_Link)
         end
      else { reached a link that is nil } ;
   end;
{ ********************************************************** }
   procedure Display_Dictionary (Header : Tree);
      var
         P : Pointer;
   begin
      P := Header^.Link;
      if P <> nil then
         Print_Word(P)
      else
         writeln(' Dictionary is empty. ');
      writeln;
   end;
{ ********************************************************** }
begin { Body of the main program. }
{ Show Text window for displaying prompts. }
   ShowText;
{ Create header for dictionary. }
   Create_Dictionary_Header(Dictionary);
{ Display contents of the dictionary. }
   Display_Dictionary(Dictionary);
   repeat
   { Prompt for next word. }
      write(' Enter next word: ');
      readln(Word);
   { Insert new word in dictionary. }
      Insert_Word(Dictionary^.Link, Word);
      writeln;
   { Display contents of dictionary. }
      Display_Dictionary(Dictionary);
      writeln;
   { Prompt user to continue. }
      write(' Enter Y to continue inserting words, N to quit: ');
      readln(Response);
   until Response <> 'Y';
   Display_Dictionary(Dictionary)
end.
```

The program also has a procedure called `Display_Dictionary`. In the body of `Display_Dictionary` the link field of the header is tested to see if it is **nil**; if so, the program reports that the dictionary is empty.

If we can insert new words into our dictionary, how can we delete words? Deleting words is more complex than inserting them, because it can require us to adjust some of the nodes of the binary tree in order to keep the proper alphabetic ordering. As Figures 12.13a and 12.13b show, deletion of a word is simple if the node to be deleted is a leaf node or has only one descendant.



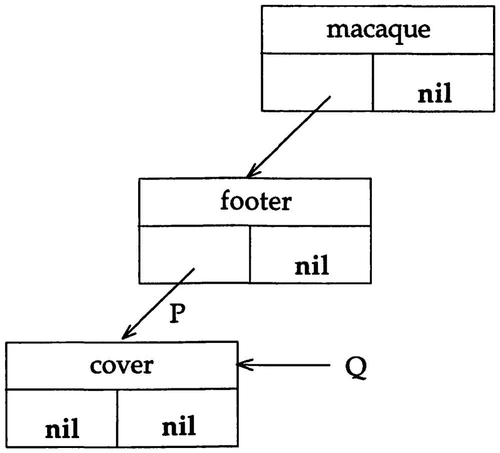
Node Q is to be deleted:

```
P <-- P^.Right_Link
dispose(Q)
```

`P` represents the right link field of the preceding node.

Figure 12.13a Deletion of a node having one descendant.

In the case of a leaf node only the link field of a preceding node is set to **`nil`**, because the leaf node has no descendants. In the case of a node with a single descendant, only the link field pointing to the node for deletion is modified by assigning either the left link or right link field of this node to a corresponding left or right link field of a preceding node before disposing of the node.



Node Q is to be deleted:

```
P <-- P^.Left_Link
dispose(Q)
```

P represents the right link field of the preceding node.

Figure 12.13b Deletion of a leaf node, no descendents.

The difficult case is removing a node having two descendants. There are two ways to do so. One is to replace the word to be deleted by the word in the rightmost node of the left link of the node containing the word for deletion, and then delete the rightmost node. The second approach is to replace the word to be deleted by the leftmost node of the right link of the node containing the word for deletion. The leftmost node is then deleted. Figures 12.14a and 12.14b show two examples displaying a leftmost node and a rightmost node. In either case, the rightmost or leftmost node is assumed to have at most one descendant. Using either approach, the word contained in the root node will always be properly ordered with respect to the other words in the dictionary. In addition, the root node will keep its proper descendants. We will use the second approach; the first is left as an exercise.

Assuming that a pointer called P points to the root of the tree, these are the basic steps in deleting a node:

1. Test if P is **nil**, and if it is, report that the word does not exist.
2. Otherwise, test if the word is alphabetically less than P^.Value, and if it is, repeat this algorithm by passing to P the pointer value of P^.Left_Link.
3. Otherwise, test if the word is alphabetically greater than P^.Value, and if it is, repeat this algorithm by passing to P the pointer value of P^.Right_Link.
4. Otherwise, prepare to delete node P.
5. Set a temporary pointer Q to point to where node P is pointing.
6. Test if P^.Right_Link is **nil**, and if it is, let P become P^.Left_Link.
7. Otherwise, test if P^.Left_Link is **nil**, and if it is, let P become P^.Right_Link.

8. Otherwise, trace the tree starting with `Q^.Right_Link` until the leftmost node is found. Replace the contents of `Q` with the contents of the leftmost node, and delete the leftmost node.
9. Dispose of the node pointed to by `Q`.

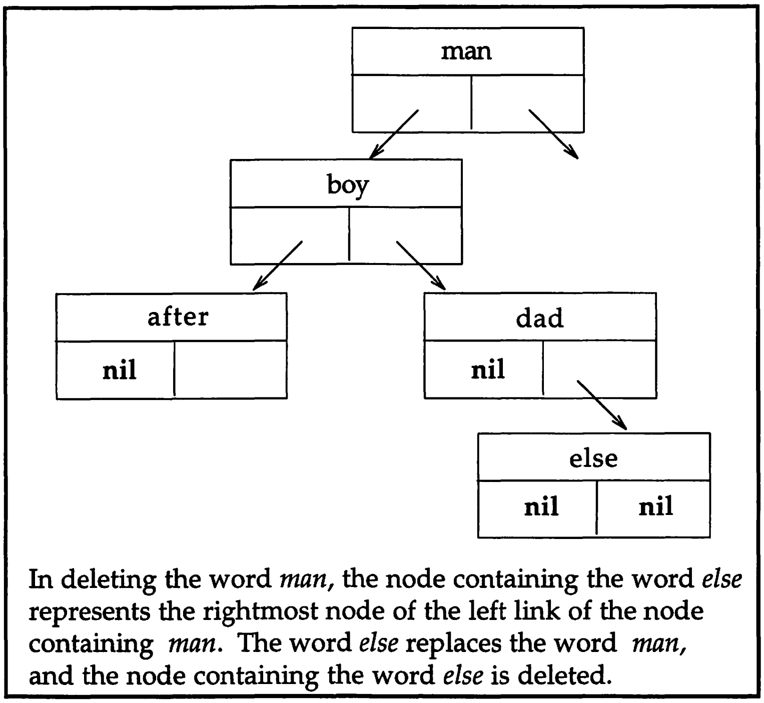
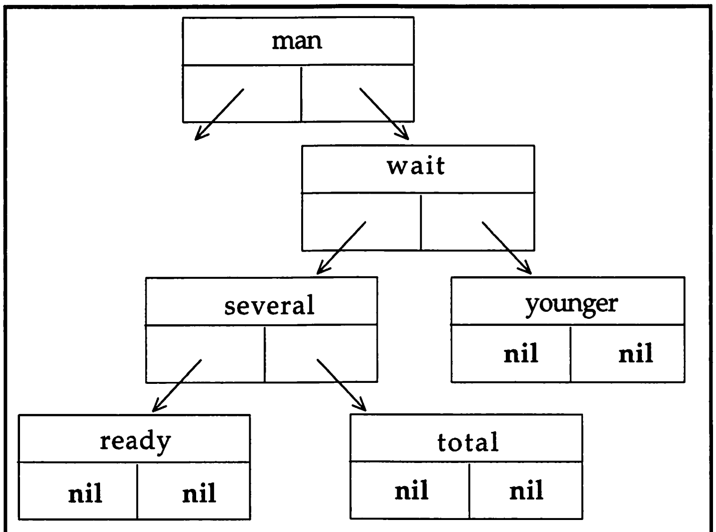

man
boy
after
nil
dad
nil
else
nil
nil
In deleting the word *man,* the node containing the word *else* represents the rightmost node of the left link of the node containing *man*. The word *else* replaces the word *man,* and the node containing the word *else* is deleted.

Figure 12.14a Deleting a node with two descendants, first approach.

Here is a refinement of our algorithm:

```
procedure Delete_Word( var P : Pointer; Word : string );
   var
      Q : Pointer;
begin
{ Test if the word is not in the dictionary. }
   if P = nil then
      writeln(' The word ', Word , ' is not in this dictionary.' )
   else { Test if the word is to the left of node P. }
      if Word < P^.Value then
      { Continue to delete to the left of node P. }
         Delete_Word( P^.Left_Link, Word )
      else { Test if the word is to the right of node P. }
         if Word > P^.Value then
         { Continue to delete to the right of node P. }
            Delete_Word( P^.Right_Link, Word )
         else { Dispose of node P. }
            begin
            Q := P;
            if P^.Right_Link = nil then
               P := P^.Left_Link
            else
               if P^.Left_Link = nil then
                  P := P^.Right_Link
               else
                  Delete_Leftmost_Node( Q^.Right_Link, Q );
            dispose( Q );
         end
end;
```



In deleting the word *man,* the node containing the word *ready* represents the leftmost node of the right link of the node containing *man.* The word *ready* replaces the word *man,* and the node containing the word *ready* is deleted.

Figure 12.14b Deleting a node with two descendants, second approach.

The steps for deleting a leftmost node require special attention. Assume that pointer `R` is passed the value of `Q^.Right_Link`. The steps for `Delete_Leftmost_Node` are then as follows:

1. Test if `R^.Left_Link <> nil`, and if so, repeat this algorithm by passing to `R` and `Q` the pointer values of `R^.Left_Link` and `Q`.
2. If not, there are no further left descendants, since `R` is now pointing to the leftmost node.
3. Let `Q^.Value <-- R^.Value; Q <-- R;` and `R <-- R^.Right_Link;`.

Here is a refinement of the algorithm:

```
procedure Delete_Leftmost_Node( var R, Q : Pointer );
begin
{ Test if we have reached the leftmost node in the tree. }
   if R^.Left_Link <> nil then
      Delete_Leftmost_Node( R^.Left_Link, Q )
   else { No further descendants exist to the left of node R. }
      begin
         Q^.Value  :=  R^.Value;
         Q  :=  R;
         R  :=  R^.Right_Link
      end
end;
```

Before ending this discussion, we must note that Pascal cannot support a file of pointers. Pointers represent actual addresses in memory during the execution of a program. Although it is possible to write pointers to a file, reading pointers and assigning their values to pointer variables would only disrupt the execution of a Pascal program. The objects pointed to by a pointer can be written to a file, and an object stored in a file can be read and placed in the memory allocated for such an object.

12.5 HEURISTICS FOR WRITING RECURSIVE ROUTINES

Divide and conquer is the process of solving a problem by dividing it into smaller subproblems. Often a step in a problem can divided into two or more substeps when refining the solution. This method is also associated with solving problems from the top down, because a solution is assumed to exist, and we then try to define one or more steps that represent a solution to a problem. This often leads us to refine each step by defining two or more substeps as subproblems.

When defining a formal algorithm for a problem, we often find it helpful to divide a step (substep) by having the refined step call upon itself to conquer the solution. There are heuristic rules for writing recursive algorithms that we should review. First, you must understand the problem you want to solve before writing an algorithm for a solution. Second, you must understand the abstract data types and the objects of these types that are involved in the solution of the problem. For example, if an algorithm involves objects that have a recursive structure, such as a linked list or a binary tree, we can often apply steps in solving the problem to a lesser part of the structure by using the algorithm presently being defined, because parts of a recursive object have the same properties as the complete object. The objective is to write an algorithm that applies to the whole data object as well as to its parts. Third, understand the trivial case(s) for terminating computation on an object, for example, reaching the end of a linked list when the link field is **nil**, or continuing to apply the same algorithm to the remainder of a list until an empty list is reached. Fourth, understand the nontrivial cases and try to reduce initial expressions for objects to trivial cases. For example, we may continue to copy a linked list by making a copy where the link field is pointing. Often, we can replace a loop needed for an iterative step of an algorithm with a recursive call. Fifth, combine the third and fourth hints with a conditional expression using trivial or terminal case(s), and then apply the algorithm to the remaining portion or portions of a data object(s). Last, check the definition of the algorithm by using both trivial and nontrivial examples to verify correctness.

As an example, assume that a list is represented by an element followed by a sublist. A sublist is in turn a list having an element followed by a sublist. Only when a sublist is empty does the list terminate. As you can see, the concept of a list is recursive, because it is composed of an element followed by a list. What we need is an algorithm for making a copy of a list. The only trivial case is where an empty list is given, because a copy of an empty list is itself an empty list. If the given list is not empty, the first element of this list is copied. The same definition is then applied to the sublist of the given list until the sublist being copied is empty.

In implementing this algorithm, we define a list as a structure represented by a singularly linked list composed of elements that each have two fields: a value field and a link field. The link field points to a successor element representing the beginning of a sublist. In a trivial case, the list is empty, represented by the value of the given list being **nil**. In turn, a link field of a list element with a **nil** value defines the list as being terminated.

The following is a recursive procedure called `Copy_List`. In this definition we assume that the formal parameter `L` is the given list ready to be copied and `C` represents a copy of list `L`. Initially, the trivial case is checked, because if list `L` is empty, a copy of it is itself an empty list. If not, a new element is created, and the value of the next element in the given list is assigned to the element pointed to by `C`. The sublist that follows list `L` is now copied by calling on `Copy_List`, where the value of the actual parameter associated with `L` is the sublist given by the linked field of `L`.

When a copy of this sublist is returned, it is assigned to the link field of the new element pointed to by `C`. The procedure `Copy_List` is directly recursive, but recursion ends when the value of the formal parameter `L` is **nil**. Without this trivial case, the procedure would in theory go on forever.

```
procedure  Copy_List( L : List; var C : List );
{Purpose:  This routine makes a copy of a singularly linked }
{          list, assuming that no header node is present for }
{          list L.}
   var
      R : List;
begin
{ Check for a trivial case where list L is empty and recursion }
{ terminates.}
   if L = nil then
   { Return a copy of an empty list.}
      C := nil
   else
      begin
      { Create the next element for list C.}
         new(C);
      { Copy the value field from the next element in list }
      { L to C.}
         C^.Value := L^.Value ;
      { Make a copy of the sublist of L and assign this }
      { to R. }
         Copy_List(L^.link, R);
      { Link a copy of the sublist of L to the next }
      { element of C.}
         C^.link :=  R;
      end;
end;
```

Why not let pointer C be assigned the value of L and avoid writing a recursive algorithm such as Copy_List? If the algorithm has only the assignment statement C := L as its body, disposing of list L would leave dangling pointers. Procedure Copy_List avoids this problem.

12.6 ADDITIONAL COMMENTS ON NEW AND DISPOSE

The command new can also be used to allocate storage for a record having variant fields. In this case the new command uses the following syntax:

new(Pointer_Variable, c_1, c_2, . . . , c_n);

where Pointer_Variable points to a record having one or more variant fields, and the constants c_2, . . . , c_n are expressions of an ordinal type related to a tag field of a variant record. For example, consider the following definitions:

```
type
   Education = ( Elem, High, College, Graduate );
   Profile =   record
                  Employee_Name : string;
                  Street_Address: string;
                  City_State: string;
                  Zip_Code: string;
                  Age: integer;
                  Gender: char;
                  case Grade_Level  : Education  of
                     Elem : (Last_Grade : integer);
                     High  : (Junior: integer; Senior: integer);
                     College : ( Year : (Fr, Sp, Jr, Sr));
                     Graduate: ( Degree : ( MA, MS, PHD, PHE ));
               end;
   Employee_Record = ^Profile;
var
   Employee : Employee_Record;
   Educational_Experience : Education;
```

We can allocate memory by using the variant that corresponds to one of the tag fields. The short program titled Pointer_Varying_Record shows how to create a record and how to select the proper variant record using the command new.

```
program Pointer_Varying_Record(input, output);
{ This program demonstrates the extended use of new and dispose. }
```

```
   uses
      QuickDraw1;
   type
      Education = (Elem, High, College, Graduate);
      Profile =   record
                     Employee_Name : string;
                     Street_Address : string;
                     City_State  : string;
                     Zip_Code : string;
                     Age : integer;
                     Gender : char;
                     case Grade_Level : Education of
                        Elem : ( Last_Grade : integer);
                        High : ( Junior : integer;
                                 Senior : integer);
                        College : ( Year : (Fr, Sp, Jr, Sr) );
                        Graduate : ( Degree : (MA, MS, PHD, PHE) )
                  end;
      Employee_Record = ^Profile;
   var
      Employee : Employee_Record;
      Educational_Experience : Education;
begin
   ShowText;
   write(' Enter educational experience (Elem, High, College,
           Graduate): ');
   readln(Educational_Experience);
   new(Employee, Educational_Experience);
   with Employee^ do
      begin
         Grade_Level := Educational_Experience;
         case Grade_Level of
            Elem :
               begin
                  Last_Grade := 7;
                  writeln(Last_Grade);
               end;
            High :
               begin
                  Senior := 12;
                  writeln(Senior);
               end;
            College :
               begin
                  Year := Fr;
                  writeln(Year);
               end;
            Graduate :
               begin
                  Degree := MA;
                  writeln(Degree);
               end;
           end;  { case }
         end;
   dispose(Employee, Educational_Experience);
end.
```

If during execution the response is the value Graduate, only the invariant part of the record and the variant part for the label constant Graduate are allocated storage. The variant selector Grade_Level must be explicitly assigned a value, but only after the record for the pointer Employee has been created. In this example, execution of the command new only establishes the storage for the record; it does not assign any value to the variant selector Grade_Level.

Pascal requires that, for a record created in the manner just described, storage be deallocated in the same order in which it was created. That is, we must execute the command dispose using the following syntax:

```
dispose( Pointer_Variable, c1, c2, . . . , cn );
```

For example, execution of

```
dispose( Employee, Educational_Experience );
```

deallocates storage for the record associated with the pointer Employee if Educational_Experience still has the value Graduate.

12.7 MACINTOSH MEMORY MANAGER AND THE CONCEPT OF HANDLES

Macintosh Pascal supports several procedures and functions for managing dynamically allocated objects during the execution of a program. Unlike the pointers controlled by the commands new and dispose, the Memory Manager makes use of a special object referred to as a *handle.* A handle is a pointer that points to an object indirectly through the path of another pointer. For example, consider a special handle called Employee_Handler pointing to Employee_Record, with Employee_Record pointing to a record called Profile:

```
type
   Profile =    record
                   Name : string[30];
                   ID_Number : string[10];
                   Age : integer
                end;
   Employee_Record = ^Profile;
   Employee_Handler = ^Employee_Record;
var
   Employee : Employee_Handler;
```

By using this indirect referencing scheme, the Memory Manager can periodically move the contents of the Profile record to maximize available free storage space in memory. This is different from pointers that allocate storage in terms of nonrelocatable blocks. In the context discussed here, handles are assignment-compatible with any pointer type.

Handles can be allocated storage by a function called `NewHandle` and deallocated storage by a procedure called `DisposeHandle`. The function `NewHandle` takes only one argument, an integer expression representing the total number of bytes required to allocate storage for the dynamic variable created by this function. For example, to allocate storage and create a handle indirectly pointing to an object represented by `Profile`, we would execute the following assignment statement:

```
Employee := NewHandle( sizeof( Profile ) );
```

The value returned by this function is a handle pointing to a region in memory having a size determined by its argument. The function `sizeof` returns a long `integer`, and it measures the number of bytes of storage for either a variable identifier or a programmer-defined type. The function `sizeof` cannot measure the sizes of file types or structure types containing file types.

To reference a field in the record `Profile`, we use a double pointer representation. For example, a value for the name field would be referenced by `Employee^^.Name`. Disposing of a handle and releasing the storage associated with it is accomplished by the command `DisposeHandle`. The syntax for this routine follows:

```
DisposeHandle( Handle );
```

We must be very careful when using the commands `NewHandle` and `DisposeHandle`, because they bypass the normal type-checking mechanisms of the Macintosh Pascal translator. This means that there is *no protection* for the handles or your program if these routines are used improperly.

An additional function and procedure exist for measuring and changing the handle size. The function `GetHandleSize` takes as an argument a handle and measures the actual number of bytes created by `NewHandle`. The type of value returned by this function is a long `integer`. The procedure `SetHandleSize` takes as input two actual parameters, an existing handle and a value for its new size. It establishes a new memory size for the object indirectly pointed to by the handle. When using this procedure, use the function `GetHandleSize` to confirm the size allocated to the new memory area.

The last procedure to be discussed in this section is used to copy the contents of one region and write them to another. The procedure `BlockMove` copies the number of bytes of memory occupied by the region starting at the location given by `Source_Pointer` and writes this number of bytes to a region of memory starting at `Destination_ Pointer`. The syntax for this routine follows:

```
BlockMove( Source_Pointer, Destination_Pointer, Memory_Area );
```

For this particular procedure, `Source_Pointer` and `Destination_Pointer` can be of any pointer type. For example, the following block move will shift a block of bytes starting at the memory address given by `Employee^` to the memory address given by `Temporary^`:

```
Temporary := NewHandle( sizeof( Profile );
BlockMove( Employee^, Temporary^, sizeof(Profile) );
```

The following program, titled `Handles`, illustrates the moving of a block of bytes from a record indirectly pointed to by a handle called `Employee` to another handle called `Temporary`.

```
program Handles(input, output);
{ Purpose:  This program uses the the memory management routines }
{           of Macintosh Pascal. }
   uses
      QuickDraw1;
   type
      Profile =   record
                     Name : string[20];
                     ID_Number : string[10];
                     Age : integer
                  end;
      Employee_Record = ^Profile;
      Employee_Handler = ^Employee_Record;
   var
      Temporary, Employee : Employee_Handler;
begin
   ShowText;
   Employee := NewHandle(sizeof(Profile));
   Temporary := NewHandle(sizeof(Profile));
   with Employee^^ do
      begin
         Name := 'Jack & Jill';
         ID_Number := '0123456789';
         Age := 34
      end;
   writeln(' Size of record profile: ', sizeof(Profile) : 3);
   writeln;
   BlockMove(Employee^, Temporary^, sizeof(Profile));
   with Temporary^^ do
      begin
         writeln('Employee name: ', Name);
         writeln('Identification number: ', ID_Number);
         writeln('Age: ', Age : 3)
      end;
   DisposeHandle(Employee);
   DisposeHandle(Temporary);
end.
```

Beware: `BlockMove` does *no* error checking; it simply moves bytes of data between two locations in memory. Executing the command

```
BlockMove ( Employee, Temporary, sizeof( Profile ))
```

can bomb the Macintosh operating system, because both the pointers `Employee` and `Temporary` are pointing to handles, not record structures.

12.8 THINK PASCAL VERSUS STANDARD PASCAL

THINK Pascal has several extensions that are not supported in standard Pascal. First, both Macintosh and THINK Pascal support the referencing operator @ for returning the address of a pointer to the variable upon which it operates. Whereas standard Pascal is defined to hide the values of the addresses assigned to pointer variables, both THINK and Macintosh Pascal support two explicit functions: `ord4` and `pointer`. Function `ord4` takes as an argument an expression representing the value of an `ordinal` type or `pointer` type and returns a `longint` value. For a `pointer` type, this represents the absolute address of the dynamic variable pointed to by the argument of `ord4`. The function named `pointer` converts an `integer` type value to a generic `pointer` type value. Because it has only a single argument with the value of an `integer` type, this function returns a generic pointer matching any `pointer` type.

SUMMARY

Pointers provide a new dimension for the programmer wanting to develop sophisticated algorithms containing complex data structures such as stacks, queues, linked lists, and binary trees. By using pointers in Pascal, we can dynamically allocate and deallocate storage for data objects by the executing the commands `new` and `dispose`. The Macintosh Pascal extensions for creating handles give us the option of having the Macintosh Memory Manager optimize heap storage during normal execution. In particular, we can collect and remove garbage, adding storage to the free storage pool for later objects created dynamically.

REVIEW QUESTIONS

1. What is an abstract data type?
2. What purpose is served by a constructor? destructor? modifier? selector? iterator?
3. What is a pointer?
4. How is a pointer declared in Pascal?
5. How can a pointer be declared by using a programmer-defined type?
6. What is the initial value for a pointer in Macintosh Pascal?
7. What is the purpose of the command `new`? What are the actions of command new when executed?
8. What is the purpose of the command `dispose`? What are the actions of command `dispose` when executed?
9. Why are pointer types different from other data types found in Pascal?
10. Why are pointers considered dynamic storage objects in Pascal?
11. What is the purpose of using a stack to store information during the execution of a program?
12. What is the concept of heap storage in Pascal?
13. What is the purpose of using a heap to store information during the execution of a program?

14. What can happen in Pascal if pointers are not type-compatible? Give an example to support your answer.
15. Assuming that both X and Y are of the same pointer type, are the assignment statements `X := Y;` and `X^ := Y^;` equivalent? Explain your answer.
16. Consider the following declarations:

```
type
   Employee_Record = record
                        ID_Number : integer;
                        Name : string[30];
                        Age : integer;
                     end;
   Person = ^Employee_Record;
var
   Employee : Person;
```

Is the data type for the variable `Person` a pointer or a record? Explain your answer.

17. What kind of object is created when the command `new(Employee)` is executed? What is the name of the object pointing to what has been created?
18. Consider the following declarations:

```
type
   Employee_Record = record
                        ID_Number : integer;
                        Name : string[30];
                        Age : integer;
                     end;
   Person = ^Employee_Record;
var
   Employee : array[1..100] of Person;
```

What is the difference between the variable `Employee` here and the variable `Employee` declared in Question 16? What occurs when the command `new(Employee)` is executed?

19. Does Macintosh Pascal allow a data type declared as a pointer of a pointer? Can you show an example that has been verified?
20. What unusual side effect can occur in Macintosh Pascal with the execution of the command `dispose`?
21. What is the purpose of the special unary operator represented by the character @ in Macintosh Pascal?
22. How can the operator @ be used to access the memory address of a formal parameter? Give an example to support your answer.
23. What differences exist between the values returned by operator @ when operating on a value-type parameter and a variable-type parameter?
24. What is the concept of a linked list?
25. What are the advantages of using a linked list during the execution of a Pascal program?
26. Find the errors in the following declarations, and correct them.

```
type
```

```
   Node = Profile^;
   Profile =   record
                  Name : ^string;
                  ID_Number : ^string;
                  Wage : real;
                  Link : ^Node
               end;
var
   Employee : Node;
   List_Head : Node;
```

27. What purpose is served by having a header node at the beginning of a linked list?
28. What are the steps for inserting a new node at the beginning of a linked list? Assume that this new node is to be inserted immediately following the header node.
29. What are the steps for deleting a node at the beginning of a linked list? Assume that the node to be deleted follows the header node.
30. How can a stack be represented by an array?
31. How can a stack be represented by a linked list?
32. What are the advantages of using a linked list to represent a stack? What are the disadvantages?
33. What primitive operations can be performed on a stack?
34. Explain the concept of a queue.
35. How can a queue be represented by an array?
36. How can a queue be represented by a linked list?
37. What are the advantages of using a linked list to represent a queue? What are the disadvantages?
38. What primitive operations can be performed on a queue?
39. How is a stack tested for being empty? for being full?
40. How is a queue tested for being empty? for being full?
41. What purpose can a header node serve for a queue represented by a linked list?
42. Why is a binary tree a special type of data object?
43. How can a binary tree be used to store a dictionary in RAM?
44. Explain the steps for inserting a new word in a dictionary represented by a binary tree.
45. Explain the steps for deleting a word in a dictionary represented by a binary tree.
46. Using Pascal, how can a binary tree be stored as a file on a diskette?
47. How is execution of the command `new(Pointer_Variable, C1, C2, ...,Cn)` different from execution of the command `new(Pointer_Variable)`?
48. What is meant by the data type referred to as a *handle*?
49. What purpose is served by the routine `NewHandle`? By the routine `DisposeHandle`?
50. What is the purpose of routine `GetHandleSize`?
51. What is the purpose of the routine `SetHandleSize`?
52. How can the function `sizeof` be used to measure the amount of RAM storage for a variable?
53. How can the routine `BlockMove` be used to copy the individual byte storage of a variable?
54. Why must you be careful when using the command `MoveBlock`?
55. Why are Macintosh Memory Management routines important?

56. What is meant by *garbage*?
57. If a formal parameter represents a pointer type, when should it be declared a value type, and when should it be declared a variable type?
58. Define the heuristic rules for writing a recursive algorithm.

PROGRAMMING EXERCISES

Although not all programming exercises require you to write an algorithm, you may better understand the problem and what is required by first writing an algorithm and tracing it by hand with several examples before writing a Pascal program.

1. Test the concepts for pushing onto and popping from a stack when the stack is represented by an array. Assume that the stack is to be represented by the following record format:

```
const
   N = 50;
type
   Stack =  record
               St_Pt : 0..50;
               Table : array[1..N] of char
            end;
var
   S : Stack;
```

Write a program that will read a line of text entered from the keyboard one character at a time and, after each character is read, push the character onto the stack. After the line of text has been read, have your program pop each element off the stack, displaying each character to a line in the Text window. Test your program using lines of text less than, equal to, and greater than 50 characters.

2. Repeat Exercise 1, but instead of using an array to represent the stack, use a linked list, where each node is to store a single character. Test this new program with the examples you used in Exercise 1.

3. A queue can be represented by an array, where the *n* th element is assumed to be adjacent to its first element. Figure 12.15 is a model of what is referred to as a *circular queue*. To represent this circular queue, consider the following record format:

```
const
   N = 50;
type
   Queue =  record
               Front_Pt : -1..N;
               Rear_Pt :  -1..N;
               Q_Table  : array[0..N-1]  of  char
            end;
var
   Q : Queue;
```

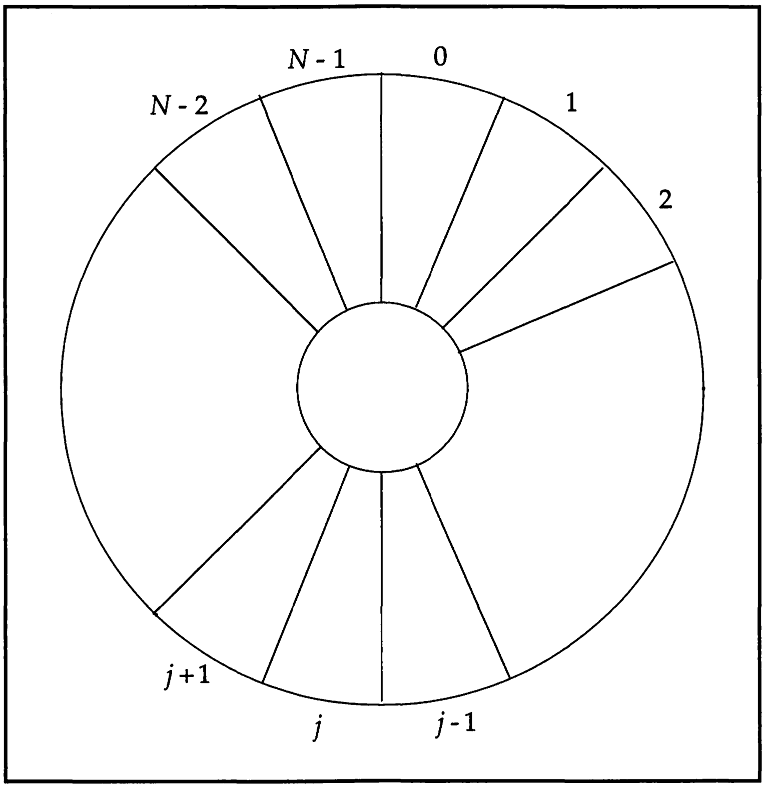


Figure 12.15 A circular queue.

Assume the following:

(a) An empty queue exists when both the front and rear pointers have a value of −1.
(b) A queue is filled when the rear pointer catches the front pointer.
(c) A queue becomes empty when the front pointer catches the rear pointer.
(d) The front pointer points to the next element to be deleted from the queue.
(e) The rear pointer points to the last element that was inserted into the queue.

Develop procedures for inserting and deleting from the queue when it is represented by an array. Use the **mod** operation when computing pointer values. Once your procedures are written, write a program for reading characters from a line of text, inserting one character at a time into the queue as a character is read. After a line of text has been read, display the elements in the queue by deleting each element from the queue and writing it to a line in the Text window. Test this new program with the examples you used in Exercise 1.

4. Repeat Exercise 3, but use a linked list to represent the queue, where each node is to store a single character. Test this new program with the examples you used in Exercise 3.

5. In a doubly linked list, the node has two pointer fields: a left link field for pointing to a preceding node and a right link field for pointing to a succeeding node. Figure 12.16 shows an example of a doubly linked list.

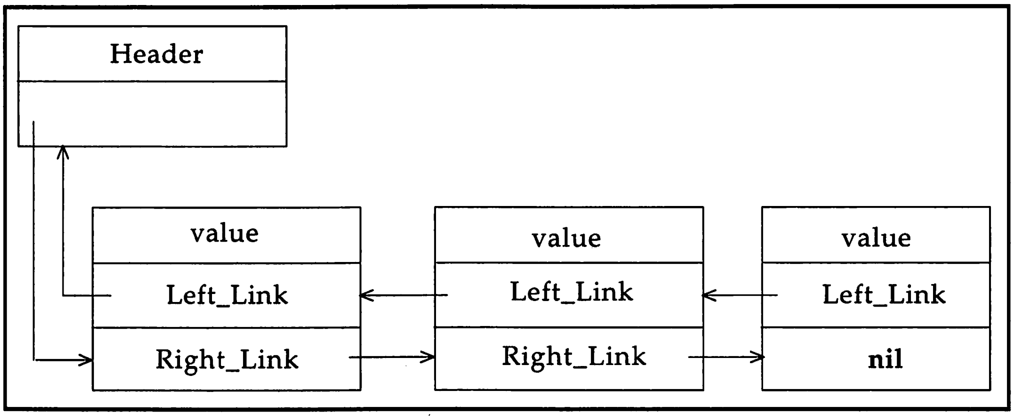


Figure 12.16 A doubly linked list.

Write a program that will offer the following options, by means of a menu:

(a) Create a header node for a doubly linked list.
(b) Add an item to a doubly linked list.
(c) Delete an item from a doubly linked list.
(d) Display the value in each node of a doubly linked list.
(e) Quit this program.

6. Develop a nonrecursive function that can measure the length (the number of nodes) in a simple linked list. Write a program for testing this new function by applying several linked lists as test data. Try writing a recursive function for performing the same action.

7. Develop a nonrecursive procedure that can test if two simple linked lists are equivalent. Write a program for testing this new procedure by applying several linked lists as test data. Try writing a recursive procedure for performing the same action.

8. Develop a procedure for appending one linked list to the end of another linked list. Keep in mind that the first node of the linked list, not the header node, is to be appended to the last node of the first linked list. Write a program for testing this new procedure.

9. Develop a recursive procedure for copying the nodes of one linked list and assigning these to the header node representing a second linked list. Write a program for testing this new procedure.

10. In some instances we can use a linked list to represent a set of elements. Develop a procedure that will take two linked lists, each representing a set of elements, and compute the union of the elements of both sets. Write a program for testing this new procedure, using several linked lists as test data.

11. If we consider a linked list to represent a set, write a procedure that can compute the intersection of two sets. Write a program for testing this new procedure using several linked lists as data.

12. In Chapter 10, we introduced an algorithm for merging a single record with a sequential file of records sorted according to some property. Using this concept, write an algorithm that can merge a node into a linked list, where all the nodes are sorted in some type of order. For example, use a linked list where each node contains a name, and the linked list holds the names alphabetically. Convert your algorithm into a Pascal procedure, and test this procedure by writing a program. Use several instances of data to test your algorithm.

13. By using an *inorder* tree traversal, we can display the words held in the dictionary in alphabetic order. There are two additional ways of walking through the tree, called *preorder* traversal and *postorder* traversal. Here are the basic algorithms for *preorder* and *postorder* traversals:

```
procedure Preorder( P : Pointer );
begin
   if P <> nil then
      begin
      { Visit the node pointed to by P. }
         Preorder( P^. Left_Link);
         Preorder(P^. Right_Link )
      end
end; { Preorder }

procedure Postorder( P : Pointer );
begin
   if P <> nil then
      begin
         Postorder( P^. Left_Link);
         Postorder(P^. Right_Link );
      { Visit the node pointed to by P. }
      end
end; { Postorder }
```

The term *visit* means to perform some type of action on the value field of the node. Using the dictionary example, add two procedures to the program so that the words (nodes in the binary tree) can be displayed in *preorder*, *postorder*, and *inorder*.

14. Modify the dictionary program so that a menu offers the user the following options:

    (a) Create an empty dictionary.
    (b) Enter an existing dictionary from a file.
    (c) Add one or more words to the dictionary.
    (d) Display the words alphabetically.
    (e) Save the dictionary presently held in memory to a file.
    (f) Exit from this program.

    When option (c) is chosen, the following steps are to be executed:

    (i) Prompt the user for one or more words.
    (ii) As the user enters words from the keyboard, have the program push each word onto a stack (or queue).
    (iii) Once all the words have been entered from the keyboard, have the program add each word to the dictionary by popping the stack (or deleting the queue).

    To implement option (e), use the *preorder* traversal scheme for copying the word from a value field of a node and writing it to a text file. Pascal cannot write pointers to a file, because it does not support a type called a **`file of`** `pointer`. Since the tree was originally grown using a *preorder* traversal, it will be preserved by using the *preorder* procedure described in Exercise 13.

15. Modify the program in Exercise 14 by adding a seventh option: Delete one or more words from the dictionary. When implementing this option, follow the approach taken in adding one or more words. First prompt the user; second, enter the words in a stack or queue. After entering all of the words, delete a word from the dictionary by popping the stack or queue until it is empty.

16. The ABC Telephone Company will issue a Macintosh computer to its single telephone operator to assist with local customer calls. You have been chosen to develop an on-line phone directory to help the operator find a phone number, given the name of the customer. During execution of the phone directory program, the operator will be given the following options by a menu:

    (a) Enter the phone directory from the file.
    (b) Search for a phone listing.
    (c) Add a new customer to the directory.
    (d) Delete a customer from the directory.
    (e) Ask for help.
    (f) Save the phone directory to the file.
    (g) Exit from this program.

    Keep in mind that the phone directory will always be stored in alphabetic order by customer name. It is impossible to know the upper limit on the number of customers, so a binary tree or a simple linked list should be used for holding customer records in RAM. The external file will be a sequential file of records. This system must also provide short help files to explain how the other options are to be chosen and used by an operator. Can you think of other options the phone company may need? Can you think of ways to protect the program during execution, so that the wrong option will not be chosen?

17. Write a program that checks a list of words entered from the keyboard against a dictionary stored in a file. If any word in the list is not in the dictionary, provide a message to the user that either the word may be misspelled, or it is not in the dictionary. If it is not in the dictionary, offer the user the option of adding the word to the dictionary. Think about using both the Text window and the Drawing window for displaying text and for prompting the user.

18. It is not uncommon to use a computer to perform polynomial arithmetic. The problem with most ASCII-oriented computers is that they lack the graphic capability of displaying two-dimensional output. Consider that a polynomial of the form

$$c_n x^n + c_{n-1} x^{n-1} + c_{n-2} x^{n-2} + \ldots + c_2 x^2 + c_1 x + c_0$$

can be typed as input by representing the polynomial in the following format:

$$c_n * x * * n + c_{n-1} * x * *(n-1) + \ldots + c_2 * x * * 2 + c_1 * x + c_0$$

where $c_n, c_{n-1}, \ldots, c_0$ are integer coefficients.

(a) Derive a method for reading a polynomial entered from the keyboard in the format shown. Use a linked list to store the polynomial in RAM. Assume that only a single-variable polynomial will be entered.
(b) Derive a method for displaying the polynomial in a two-dimensional format using the Drawing window. Display only the terms having nonzero coefficients.
(c) Write routines for adding, subtracting, and multiplying two polynomials.
(d) Demonstrate the use of parts (a) through (c) by developing a menu that allows a user to choose to perform some simple polynomial arithmetic.

Chapter 13

Object-Oriented Programming in THINK Pascal

OBJECTIVES

After completing Chapter 13, you will know the following:

1. What is meant by the term *object-oriented programming.*
2. The concept of a class and its implementation in THINK Pascal.
3. How objects are declared and constructed.
4. How to reference instance variables through method definitions.
5. How to apply the override directive and when to use the reserved word **`inherited`**.
6. When to apply the prefix `Self` within method definitions.
7. Using `TObject` as a root class.
8. Using the Class Browser and LightsBug for viewing objects and classes.

13.1 INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

What does the term object-oriented programming (OOP) mean? OOP is a means of implementing programs organized as a cooperative collection of objects. Each object represents an instance of a class, and each class can be a member of a hierarchy of classes united through inheritance relationships. In OOP, classes are viewed as static definitions, whereas objects represent dynamic entities. Although OOP is a style of programming that works with objects derived from classes, it functions as part of a broader methodology referred to as object-oriented analysis and design. It is a style of programming that attempts to model the real world with classes of objects that are abstract in definition but capable of being inherited by descendants. The information and actions associated with a class are implemented through inheritance.

An object-oriented programming language is different from standard programming languages such as BASIC, Pascal, FORTRAN, and C in that it supports the concept of a *class*. By itself a class is an abstraction that allows objects to share a common structure and common behavior. Although a class allows the description of information as well as the methods (operations) that one object may perform upon another object, it is not an entity by itself. Rather, it serves as a template. Thus, an object takes on the properties of a class when it becomes an instance of a class. As an instance of the class, an object is said to have state, behavior, and identity. In object-oriented programming, data and the actions performed upon data are integral parts of the definition of an object.

In procedural programming, data and actions are treated separately. Often, we define data structures and then define routines that operate on the items of particular structures. Every attempt is made to keep the data loosely coupled (using as little information as possible when passing data to a routine) when defining routines. Even when the data and actions are packaged as units, loose coupling between data and actions remains. For any new data structure that we define, we must specify new actions, even though a previous action may in principle be a definition of a method that could operate on the newly defined structure. A weakness of procedural programming is that we cannot reuse what has already been defined.

In object-oriented programming, data and actions are tightly coupled and are treated as a class. When data is defined, the action operating upon the data is also defined. The class now becomes an abstraction that can later be inherited by a subclass. This subclass inherits all the data and actions of its parent class and, in turn, may add data and actions to the class that it inherits. In addition, a subclass can define an action that overrides the action of the class that it inherits by allowing the name of an action to respond to some common set of operations in a different way than defined by its parent class. A subclass can also become an ancestor to another subclass, with the latter subclass inheriting data and actions from all of its previous ancestors. This is similar to the concept of a family tree, where siblings inherit from a parent, and a parent inherits from an order of ancestors. Not only do siblings inherit the characteristics of their parents and ancestors, they may add characteristics and perform actions that their parents and ancestors did not possess.

13.2 THE CONCEPT AND IMPLEMENTATION OF A CLASS

In the real world we often categorize objects in terms of the classes that they represent. It is often easy to associate an object with the properties and capabilities of a class. When we deal with an object, we are in essence dealing with the definition of a class. In object Pascal a class is much like a record structure in that it is composed of private data, referred to as *instance variables*, similar to the fields within a record. It is different from a record because it can support routines (methods) for performing actions by an object associated with a class. When an object is sent a message to perform an action, it is the routine (method) of the class that is executed. For example, the following defines a class called `Employee`:

```
Class:
   Employee

Instance Variables:
   Full_Name
   Tax_ID_Number
   Year_of_Birth
   Year_To_Date_Earnings
```

Messages	Methods
Get_Full_Name	Gets the full name of employee.
Get_Tax_ID_Number	Gets the employee's tax number.
Get_Birth_Year	Gets the birth year of the employee.
Show_Full_Name	Returns employee's full name.
Show_Tax_ID	Returns employee's tax ID number.
Compute_Age	Returns employee's age as of today's date.
Compute_Pay	Returns employee's weekly pay.

As a class, `Employee` has four instance variables; `Full_Name`, `Tax_ID_Number`, `Year_of_Birth`, and `Year_To_Date_Earnings`. These represent the data that an object of this class will possess. Messages that an object of this class can respond to (the actions that it can take) are `Get_Full_Name`, `Get_Tax_ID_Number`, `Show_Full_Name`, `Show_Tax_ID`, `Compute_Age`, and `Compute_Pay`. Routines that support the actions of a message sent to an object as an instance of class are called *methods*. The names of methods invoked by objects are called *messages.*

A new class defined in terms of an existing class is called a *subclass,* and the existing class is called a *superclass*. A class having no superclass is said to be a *root class*. Although a subclass *inherits* all of the data and variables from its superclass, subclasses can also define additional data and methods. Subclasses can also *override* methods defined by their superclass. As an example of subclasses, consider the following three subclasses called `Hourly_Employee`, `Salaried_Employee`, and `Exempt_Employee`, which inherit data from the superclass `Employee`.

```
Class:
   Hourly_Employee

Superclass:
   Employee

Instance Variables:
   Hourly_Rate
   Overtime_Rate
   Overtime_Hours
```

Message	Method
Compute_Pay	Compute hourly and overtime pay for the week.

```
Class:
   Salaried_Employee

Superclasses:
   Employee


Instance Variables:
   Annual_Salary
```

```
Message              Method
Compute_Pay          Compute weekly salary.

Class:
   Exempt_Employee

Superclass:
   Employee

Instance Variables:
   Monthly_Rate

Message              Method
Compute_Pay          Compute weekly pay.
```

Figure 13.1 shows the superclass `Employee` and its relationship to the subclasses `Hourly_Employee`, `Salaried_Employee`, and `Exempt_Employee`. Each arrow represents the dependency of a subclass on its superclass. Each rectangle has the name of the class, followed by the variables that the class supports, followed by a list of methods.

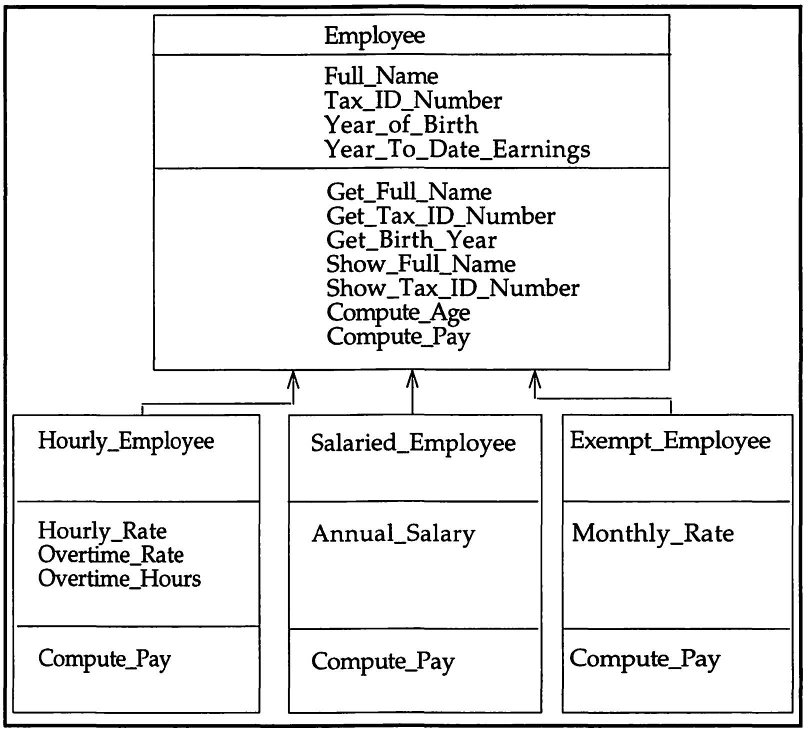


Figure 13.1 A graphic representation of the superclass `Employee` with several subclasses.

Each subclass in this example depends upon the superclass Employee. Each subclass adds information to the superclass from which it inherits as well as retaining the information that it inherits.

A superclass that has no direct instances (objects), but whose descendants can have instances, is referred to as an *abstract class*. A class that has direct instances is referred to as *concrete class*. In Figure 13.1, Employee is an abstract class, and the subclasses Hourly_Employee, Salaried_Employee, and Exempt_Employee are concrete classes. All three subclasses acquire the same messages (methods Get_Full_Name, Get_Tax_ID_Number, Get_Birth_Year, Show_Full_ Name, Show_Tax_ ID_Number, Compute_Age and Compute_Pay) from class Employee. The ability to send the same message to objects of different classes is called *polymorphism*. In Figure 13.1, the message Compute_Pay requires a different method, depending upon the object associated with one of the three subclasses. It is an *abstract method*, because its definition is deferred to one of the subclasses that directly inherits class Employee.

In THINK Pascal a class is declared with a class name, a list of instance-variable declarations, and a list of method declarations. Figure 13.2 shows the format for declaring a class as an object type. A class declaration begins with a class identifier name, followed by the reserved word **object**, optionally followed by the name of a single superclass name within parentheses, followed by field list representing instance-variable declarations, and a method list representing method declarations.

```
Class_Name = object (Superclass_Name)
                  instance-variable declarations
                  method declarations
          end;
```

Figure 13.2 The format for declaring an object-class type in THINK Pascal.

A superclass name always represents the immediate ancestor class for the class type being declared. Although a class is defined as a type, it is different from a record type, because it can include header declarations for methods. In place of the reserved word **record**, we use the word **object**. Class types are also different from record types in that a class type cannot be declared with a variant part. Understand that the method declarations in a class declaration are not method definitions. They are header declarations for methods defined by procedures and functions. The complete definitions of methods come after the class declaration as either internal or external routine definitions.

Note that for convenience the abstract class called Employee is declared in a separate unit from the subclasses. The advantage comes from information hiding. Definitions of declared methods are not required for a class to be compiled. However, they are required when a unit becomes linked with other program units. The following is the THINK Pascal code representing the superclass Employee and the subclasses given in Figure 13.1.

```
unit Abstract_Class_Employee;
interface
type
{ Declaration of a superclass called Employee. }
   Employee = object
          { Instance Variables }
             Full_Name: string[50];
```

```
                Tax_ID_Number: string[11];
                Year_of_Birth: integer;
                Year_To_Date_Earnings: real;
             { Method Declarations }
                procedure Get_Full_Name;
                procedure Get_Tax_ID_Number;
                procedure Get_Birth_Year;
                procedure Show_Full_Name;
                procedure Show_Tax_ID_Number;
                function Compute_Age: integer;
                function Compute_Pay: real;
             { This function will be overridden by each subclass.}
             end;
implementation
   { Methods are defined later. }
end.
```

The following unit provides declarations for all three subclasses of Employee. In each class type a method called Compute_Pay is declared. This method uses the override directive for overriding the definition in the immediate ancestor class called Employee. Without the override directive, a syntax error occurs, because the compiler assumes that a new method is being introduced with the same name as a method that the subclass is attempting to inherit. Not only is the class Employee considered a root class for this example, it is assumed to be an abstract class as well, provided that no instances of Employee are constructed. The remaining methods defined by class Employee are inherited by each subclass:

```
unit Subclasses_of_Employee;
interface
uses
   Abstract_Class_Employee;
type
{ Declarations of three subclasses dependent upon the }
{ abstract class Employee.}
   Hourly_Employee = object(Employee)
                     { Instance variables }
                        Hourly_Rate, Overtime_Rate: real;
                        Overtime_Hours: integer;
                     { Method Declaration }
                        function Compute_Pay: real;
                        override;
                     end;
   Salaried_Employee = object(Employee)
                     { Instance Variable }
                        Annual_Salary: real;
                     { Method Declaration }
                        function Compute_Pay: real;
                        override;
                     end;
   Exempt_Employee = object(Employee)
```

```
                          { Instance Variable }
                             Monthly_Salary: real;
                          { Method Declaration }
                             function Compute_Pay: real;
                             override;
                          end;
implementation
   { The definitions of methods are defined later. }
end.
```

Figure 13.3 shows the format for defining the methods of a class either as procedures or functions.

```
procedure Class_Name.Method_Name(parameter-list);
{ Declarations of constants, types, variables, procedures }
{ or functions local to this method. }
begin
  { Body of the method. }
end;

function Class_Name.Method_Name(parameter-list): return-type;
{ Declarations of constants, types, variables, procedures }
{ or functions local to this method. }
begin
  { Body of the method.}
end;
```

Figure 13.3 Formats for defining a method in THINK Pascal, using either a procedure or a function.

Each header begins with a dotted pair for either the procedure or function name. That is, the header begins with the class name followed by the name of the method. Within the body of a function, only the name of the method, not the class name, is used for returning a value, or for calling upon itself if it is defined as a recursive routine. The following lists the definitions of the methods for the classes given in Figure 13.1. If class names do not precede method names, either syntax or link errors are reported when the unit is compiled or built, because not all program errors in OOP are discovered during either the syntax check or the build step of the compiler. In order to shorten the listing of functions and procedures, class declarations are not shown.

```
unit Abstract_Class_Employee;
interface
type
   { Declaration of instance variables and methods for the }
   { abstract class Employee.}
   ...
implementation
   { Definitions for the methods declared by the class type }
   { Employee.}
```

```
procedure Employee.Get_Full_Name;
{ Purpose:  This routine reads an employee's full name from }
{           the keyboard.}
   begin
      readln(Full_Name);
   end;

procedure Employee.Get_Tax_ID_Number;
{ Purpose:  This routine reads an employee's tax number from }
{           the keyboard.}
   begin
      readln(Tax_ID_Number);
   end;

procedure Employee.Get_Birth_Year;
{ Purpose:  This routine reads an employee's year of birth from }
{           the keyboard.}
   begin
      readln(Year_of_Birth);
   end;

procedure Employee.Show_Tax_ID_Number;
{ Purpose:  This routine displays employee's tax number to the }
{           screen.}
   begin
      writeln('Employee`s tax number: ', Tax_ID_Number);
   end;

procedure Employee.Show_Full_Name;
{ Purpose:  This routine displays the employee's name to the }
{           screen.}
   begin
      writeln('Employee`s full name: ', Full_Name);
   end;

function Employee.Compute_Age: integer;
{ Purpose:  This routine computes the employee's present age.}
   var
      Temp_Date: DateTimeRec;
   begin
      GetTime(Temp_Date);
      Compute_Age := Temp_Date.Year - Year_of_Birth;
   end;

function Employee.Compute_Pay: real;
{ Purpose:  This is a dummy routine that is overridden by }
{           subclass definitions.}
   begin
   { This dummy routine must be present to keep the link step }
   { from failing.}
   end;
```

```
end.

unit Subclasses_of_Employee;
interface
uses
   Abstract_Class_Employee;
type
   { Declaration of instance variables and methods for the }
   { subclasses Hourly_Employee, Salaried_Employee, and }
   { Exempt_Employee.}
   ...
implementation
{ ============================================================ }
{ Definition for a method declared by the class type }
{ Hourly_Employee.}

function Hourly_Employee.Compute_Pay: real;
{ Purpose:  This function computes the weekly pay of an hourly }
{           employee.}
   var
      Weekly_Pay: real;
   begin
      Weekly_Pay := Hourly_Rate * 40 + Overtime_Hours *
                          Overtime_Rate;
      Year_To_Date_Earnings := Year_To_Date_Earnings + Weekly_Pay;
      Compute_Pay := Weekly_Pay;
   end;

{ ============================================================ }
{ Definition for a method declared by the class type }
{ Salaried_Employee.}

function Salaried_Employee.Compute_Pay: real;
{ Purpose:  This function computes the weekly pay of a salaried }
{           employee.}
   var
      Weekly_Pay: real;
   begin
      Weekly_Pay := Annual_Salary / 52;
      Year_To_Date_Earnings := Year_To_Date_Earnings + Weekly_Pay;
      Compute_Pay := Weekly_Pay;
   end;

{ ============================================================ }
{ Definition for a method declared by the class type }
{ Exempt_Employee.}

function Exempt_Employee.Compute_Pay: real;
{ Purpose:  This function computes the weekly pay of an exempt }
{           employee.}
   var
```

```
      Weekly_Pay: real;
   begin
      Weekly_Pay := Monthly_Salary / 4;
      Year_To_Date_Earnings := Year_To_Date_Earnings + Weekly_Pay;
      Compute_Pay := Weekly_Pay;
   end;
end.
```

To avoid link errors, we must define all methods, even when the body of a method is used as a stub and has no executable code (as in the function `Employee.Compute_Pay`). This method is declared in class `Employee` in order to provide a complete definition of the class. It is overridden by each of the three subclasses, because each subclass has its own rule for computing weekly pay.

13.3 DECLARING AND USING OBJECTS

Once a class is defined, we associate an object with a class through an object declaration, referred to as an *object reference*. In THINK Pascal, object references are actually handles (a handle is a pointer of a pointer), but the ^ symbol is not required. Handles allow both Macintosh and THINK Pascal to move content variables more easily and efficiently in memory. An object is declared as an object reference using a variable declaration as shown:

```
var
   Object_Name : Class_Name;
```

Though an object is associated with a class through a formal declaration, it is not allocated memory when the program begins execution. Allocation takes place through the constructor command **new**, and memory can be deallocated by using the destructor command **dispose**. Once we have constructed an object, we can reference a member (instance variable) by using a dotted pair composed of the object name followed by the instance variable, a format similar to referencing the field of a record:

```
Object_Name.Instance_Variable
```

As a rule, it is best to allow the methods of a class to have access to an object's instance variables, because it is the class that defines all of the responsibilities and data associated with a class. It is the responsibility of a class to get, set, and report values of any instance variable, because it is by defining a class that information associated with the class is hidden. Due to the use of tight coupling, few formal parameters are necessary when defining the methods of a class.

When an object is given storage, it has access to all the instance variables defined by its class as well as all of the instance variables through the chain of ancestors that the object's class inherits. It also has access to all of the methods defined by an object's class as well as all of the methods inherited by the object's class. An object acts as if it is an autonomous element with its own set of characteristics. If a second object of the same class is declared and constructed, it also acts as an autonomous element with its own data and actions. This is one of the properties that makes object-oriented programming languages different from procedural languages.

A message is sent to an object by using the syntax

Object_Name.Message(parameters).

While Object_Name is necessary in this syntax, the prefix Class_Name given in the definition of a method is unnecessary; the object name has been declared as having a particular type of class. Following is a short program designed to test the methods of the abstract class Employee and the subclass Hourly_Employee.

```
program Testing_Classes_And_Instances(input, output);
{Purpose:   This program creates an instance of a class and }
{           sends messages to the class for performing various }
{           actions.}
uses
   Abstract_Class_Employee, Subclasses_of_Employee;
var
   Person_1: Hourly_Employee;
begin
{ Show the Text window for viewing information on object }
{ Person_1. }
   ShowText;
{ Create an instance of the subclass Hourly_Employee. }
   new(Person_1);
{ Assign a name and tax ID number to instance variables of }
{ Person_1.}
   write('Enter employee`s full name: ');
   Person_1.Get_Full_Name;
   write('Enter employee`s tax identification number: ');
   Person_1.Get_Tax_ID_Number;
   write('Enter employee`s year of birth (19dd): ');
   Person_1.Get_Birth_Year;
{ Assign temporary data to instance variables of Person_1.}
   Person_1.Hourly_Rate := 6.50;
   Person_1.Overtime_Rate := 9.75;
   Person_1.Overtime_Hours := 1;
{ Send messages to object Person_1 for the purpose of }
{ displaying information.}
   Person_1.Show_Full_Name;
   Person_1.Show_Tax_ID_Number;
   writeln('Employee`s age: ', Person_1.Compute_Age : 3);
   writeln('Wages for the Week: $', Person_1.Compute_Pay : 9 : 2);
   writeln('Wages for the Year: $', Person_1.Year_To_Date_Earnings
              : 9 : 2);
{ Dispose of the object Person_1 before ending execution.}
   dispose(Person_1);
end.
```

Because object Person_1 is of class type Hourly_Employee, and subclass Hourly_Employee has as an ancestor Employee, Person_1 has the following instance variables (data) at its disposal:

```
Full_Name
Tax_ID_Number
Year_of_Birth
Year_To_Date_Earnings
Hourly_Rate
Overtime_Rate
Overtime_Hours
```

The first four instance variables are inherited, and the last three are added by the subclass declaration. The object-reference `Person_1` inherits all of its methods from its immediate ancestor. The method `Compute_Pay` is an abstract method in class `Employee` and is overridden in the subclass associated with `Person_1`.

The property of polymorphism makes it convenient to program with an object-oriented language. As an example, suppose that two new classes `Seasonal_Hourly_Employee` and `Seasonal_Salaried_Employee` are added to the class hierarchy shown earlier in Figure 13.1. They are different from their ancestor classes in that they have a date when employment is initiated. `Seasonal_Hourly_Employee` will inherit its data and methods from `Hourly_Employee` while `Seasonal_Salaried_Employee` will inherit its data and methods from `Salaried_Employee`. Figure 13.4 shows the subclass relationships with their superclasses.

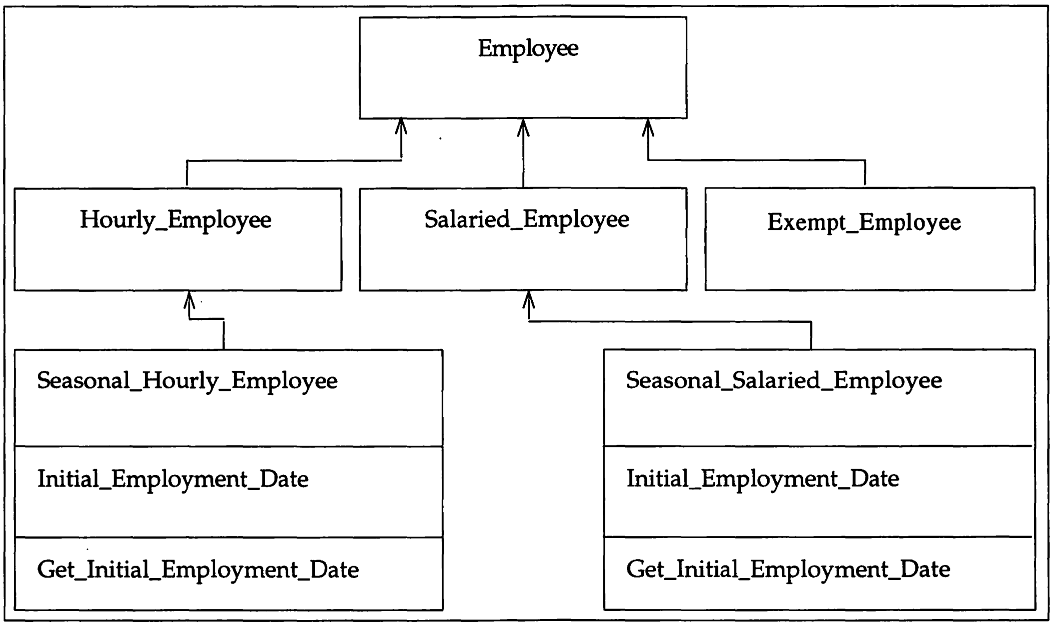


Figure 13.4 A graphic representation of the superclasses `Employee`, `Hourly_Employee`, `Salaried_Employee`, and `Exempt_Employee`, along with two new subclasses.

Each new subclass supports a method for reading the initial date of employment. The following shows a separate program unit designed for the purpose of declaring these two new subclasses and defining their methods.

```
unit Seasonal_Employee_Classes;
interface
   uses
      Abstract_Employee_Class, Subclasses_of_Employee;

type
   Seasonal_Hourly_Employee = object(Hourly_Employee)
                        { Instance Variable }
                              Initial_Employment_Date: string[8];
                        { Method Declaration }
                           procedure Get_Initial_Employment_Date;
                        end;
   Seasonal_Salaried_Employee = object(Hourly_Employee)
                        { Instance Variable }
                           Initial_Employment_Date: string[8];
                        { Method Declaration }
                           procedure Get_Initial_Employment_Date;
                        end;

implementation
{ ============================================================ }
{ Definition of the method declared by the class type }
{ Seasonal_Hourly_Employee.}
procedure Seasonal_Hourly_Employee.Get_Initial_Employment_Date;
   begin
      readln(Initial_Employment_Date);
   end;

{ ============================================================ }
{ Definition of a method declared by the class type }
{ Seasonal_Salaried_Employee.}

   procedure Seasonal_Salaried_Employee.Get_Initial_Employment
               _Date;
   begin
      readln(Initial_Employment_Date);
   end;
end.
```

Unfortunately, Seasonal_Hourly_Employee and Seasonal_Salaried_Employee cannot inherit a method that is overridden by their immediate ancestor class. For example, if function Compute_Pay in class Employee is redefined with actions that both seasonal classes can apply, only the method Compute_Pay in the immediate ancestor class is available to each of the seasonal subclasses. In THINK Pascal, the method Compute_Pay in class Employee is out of reach unless the method Compute_Pay is eliminated from the subclasses Hourly_Employee and Salaried_Employee, and the units Abstract_Class_Employee and Subclasses_of_Employee are rebuilt.

Can one object of a particular class be directly assigned to an object of another class by using the assignment operator? In THINK Pascal, two objects of different classes are type-compatible *provided that* the class of the left-hand operand of the assignment operator is an ancestor to the class of the right-hand operand. The assignment operation is allowed for assigning an object to an object having an ancestor class but not a descendant class. In addition, type-casting is allowed for an object being cast into a descendant class as long as the class of the object being cast is an ancestor. Improper type-casting is only checked during execution, provided that the programming unit containing the type-casting has been built with the range check turned on. As an example, consider the following declarations and statements for a segment of a program:

```
var
   Company_President: Employee
   Person_1: Hourly_Employee;
   Person_2: Seasonal_Hourly_Employee;
   Person_3: Seasonal_Salaried_Employee;
begin
...
{ The following assignment statements are type-compatible, since }
{ the class types associated with Company_President and Person_1 }
{ are ancestors of the subclass associated with Person_2.}

   Person_1 := Person_2;
   Company_President := Person_2;

{ The following assignment statements are type-incompatible, }
{ since the class type of Person_2 is not an ancestor to the }
{ class types associated with Person_1 and Company_President.}

   Person_2 := Person_1;
   Person_2 := Company_President;

{ The following type-casting of Company_President is allowed, }
{ since the class type of Company_President is an ancestor to }
{ Seasonal_Hourly_Employee.}

   Person_2 := Seasonal_Hourly_Employee(Company_President);

{ The following type-casting of Person_1 will fail during }
{ execution if the program unit containing this code has been }
{ built with range checking turned on.}

   Person_3 := Seasonal_Salaried_Employee(Person_1);
...
end.
```

THINK Pascal objects are actually handles, so no binary operations other than the assignment operator are allowed.

In addition to the procedures new and dispose for creating a new instance of an object and for deleting a specified object, THINK Pascal also supports a function called member. The following describes this function:

```
function Member( Object_Reference, Class_Name ) : Boolean;
```

When executed, this function checks to see if the Object_Reference is a member of Class_Name. If so, the Boolean value *true* is returned; otherwise, *false* is returned. An object reference is a member of a class provided that the object reference is of the type Class_Name, or that Class_Name is a superclass of the object's actual class. The following is a portion of code that implements the function Member:

```
if Member(Person_1, Hourly_Employee) then
   writeln('Person_1 is a member of subclass Hourly_Employee.')
else
   writeln('Person_1 is not a member of subclass
            Hourly_Employee.');
if Member(Person_1, Employee) then
   writeln('Person_1 is a member of the superclass Employee.')
else
   writeln('Person_1 is not a member of the superclass
            Employee.');
if Member(Person_1, Salaried_Employee) then
   writeln('Person_1 is a member of subclass Salaried_Employee')
else
   writeln('Person_1 is not a member of subclass
            Salaried_Employee.');
```

Unfortunately, THINK Pascal supports no type of function to return the type of class for a given object. This is best defined as a function within an abstract class by the programmer. All other operations on objects other then new, dispose, and member must be defined through programmer-defined methods encapsulated within class definitions.

13.4 APPLYING THE RESERVED WORD INHERITED AND USING THE SELF PREFIX

A descendent class must often override the definition of a method of its immediate ancestor. This does not mean that the method being inherited from its immediate ancestor has to be defined as a stub. As an example, consider the case in which each of the three subclasses Hourly_Employee, Salaried_Employee, and Exempt_Employee override the definitions of the methods Get_Full_Name, Get_Tax_ID_Number, and Get_Birth_Date by providing a prompt for the appropriate data, followed by invoking the proper method from class Employee.

Initially, the methods of the subclasses Hourly_Employee, Salaried_Employee, and Exempt_Employee are modified as the code below demonstrates. The steps defined by the method in the immediate ancestor are to be inherited, so only the prompt is given, and the second line in the body of the method uses the steps defined by the method in the abstract class Employee. Where the name of a method in an immediate ancestor is invoked, it is preceded by the reserved word **inherited**. If not, the compiler will interpret the method as a recursive call upon itself. Only the code for subclass Hourly_Employee is given, because the code for these methods in the other two subclasses is similar:

```
unit Subclasses_of_Employee;
interface
uses
   Abstract_Class_Employee;
type
   { Declaration of instance variables and methods for the }
   { subclasses Hourly_Employee, Salaried_Employee, and }
   { Exempt_Employee.}
   ...
implementation
{ ============================================================= }
{ Definitions of methods declared by the subclass }
{ Hourly_Employee. }
procedure Hourly_Employee.Get_Full_Name;
{ Purpose:  This routine prompts for and reads an employee's }
{           full name from the keyboard.}
   begin
      write('Enter the employee`s full name: ');
      inherited Get_Full_Name;
   end;

procedure Hourly_Employee.Get_Tax_ID_Number;
{ Purpose:  This routine prompts for and reads an employee's }
{           tax number from the keyboard.}
   begin
      write('Enter the employee`s tax number: ');
      inherited Get_Tax_ID_Number;
   end;

procedure Hourly_Employee.Get_Birth_Year;
{ Purpose:  This routine prompts for reads an employee's year }
{           of birth from the keyboard.}
   begin
      write('Enter the employee`s year of birth: ');
      inherited Get_Birth_Year;
   end;
...
{ ============================================================= }
...
end.
```

For the test program Testing_Classes_And_Instances, the Pascal code and messages necessary for reading values of instance variables follows:

```
{ Assign a name and tax ID number to the instance variables of }
{ Person_1.}
   Person_1.Get_Full_Name;
   Person_1.Get_Tax_ID_Number;
   Person_1.Get_Birth_Year;
...
```

During execution, Get_Full_Name, Get_Tax_ID_Number, and Get_Birth_Date will prompt the user by executing a write statement from the methods defined in each the related subclasses. The read statement, reading an instance variable from the keyboard, executes from one of the relevant methods defined in the abstract class Employee. *It is important to understand that in THINK Pascal, an inherited call to a method can only override a method from its immediate ancestor class. It cannot apply an inherited call to a method that is farther up the chain of class hierarchy.*

In the method definitions from THINK Pascal class libraries, you will often see a prefix called Self used in referencing the names of instance variables and invoking other methods within the same class. This prefix is important in the definition of methods where during execution an object must refer to itself. The following demonstrates the Self prefix for subclass method Compute_Pay:

```
function Hourly_Employee.Compute_Pay: real;
{ Purpose: This function computes the weekly pay of a hourly }
{          employee.}
   var
      Weekly_Pay: real;
   begin
      Weekly_Pay := Self.Hourly_Rate * 40
               + Self.Overtime_Hours * Self.Overtime_Rate;
      Compute_Pay := Weekly_Pay;
      Self.Year_To_Date_Earnings := Self.Year_To_Date_Earnings
               + Weekly_Pay;
   end;
```

Within the definition of a method, the THINK Pascal compiler supplies an implied declaration of the form

```
Self : class;
```

Each time a method is invoked, an implicit parameter is passed to the method for providing a reference to the current object. Using the prefix Self is optional. Although a method definition can omit the prefix Self when referring to instance variables and to calls upon methods of the same class, its inclusion makes it easier to read the code, because it explicitly indicates that the instance variables and methods being accessed are owned by the current object.

13.5 USING TObject AS THE ROOT CLASS

In THINK Pascal, an object is represented by a special type of pointer called a *handle*. A handle is a pointer that points to another pointer, which in turn points to a location in memory that can be relocated during execution. This scheme allows THINK Pascal to periodically reorganize the content of memory to provide maximum available storage space for the allocation of new dynamic variables. Handles have an important property in both Macintosh and THINK Pascal: they are assignment-compatible with any other pointer type. Because of this property, we must be careful when applying Memory Manager procedures and functions to handles. Handles are different from simple pointers, which point to where the dynamic variables are stored. Here the storage is locked for the dynamic variable and cannot be moved by the system.

THINK Pascal supports a special file called ObjIntf.p containing the definition of a root class called TObject, which is an abstract class defining methods that allow efficient copying and deleting of any object. The following is a declaration of TObject. Because it has no instance variables, no added storage is required of any object whose class inherits from TObject:

```
type
   TObject = object
      function ShallowClone : TObject;
      function Clone: TObject;
      procedure ShallowFree;
      procedure Free;
   end;
```

Method Free disposes of the dynamically allocated storage of an object that is no longer needed. It is different from the standard procedure dispose, which only breaks the link between a handle's first pointer and its second pointer. It fails to free the dynamic storage area of the dynamic variable associated with the handle. We can override Free for any subclass. While method Free directly invokes method ShallowFree, ShallowFree is the lowest level method for freeing an object and should not be overridden.

Method Clone allows the programmer to make a copy of an object. It is different from executing an assignment statement of the form

```
Object_2 := Object_1;
```

because method Clone makes a copy of the values of all instance variables. An assignment statement simply copies an address associated with the handle representing the object. In the case of the given assignment statement, changes to instance variables of Object_2 affect those of Object_1 and vice versa, since the handles of both objects reference the same storage area in memory. If Object_1 is freed by executing Object_1.Free, Object_2 is also freed of any storage. Execution of the command Object_2.Free causes an error unless Object_2 has been assigned another object-reference. A proper way to make a copy of an object is to use the method Clone. An example of the code for cloning follows:

```
Object_2 := Class_Type_of_Object_2( Object_1.Clone );
```

When using Clone, the value returned must be cast into the class type associated with Object_2 in order to avoid a type-incompatibility error. When this function is applied to Object_1, changes to Object_2 will have no effect on the object that is cloned or upon Object_1. Unexpected side effects can be avoided by cloning an object. Also, the programmer is unencumbered in applying the method Free when an object is ready to be disposed of. TObject does allow us to override Clone. The method ShallowClone is invoked by `Clone` and is the lowest level method for copying an object. `ShallowClone` should not be overridden.

13.6 BUILDING A SIMPLE WINDOW SYSTEM

Assume that you are to implement a simple drawing system with limited drawing capability. This system will have provisions for two windows, a Text window for displaying instructions and a Drawing window for drawing either a line, a rectangle, or an oval. Instructions will include selection of the type of geometric shape to be drawn and selection of the left or right corner for drawing a figure. It is assumed that one geometric shape can be drawn over another.

Figure 13.5 shows an overall view of the classes for this system. The hierarchy of this system is based upon an abstract class called `Window`, which has two dependent subclasses.

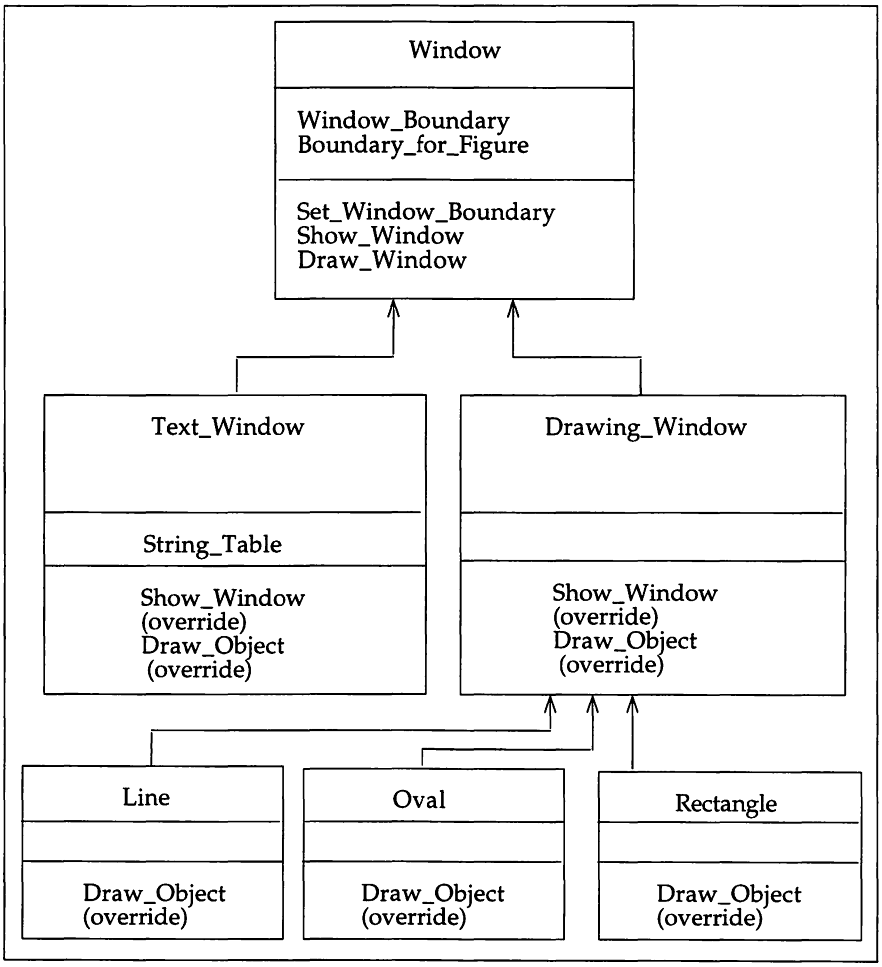


Figure 13.5 The hierarchy of classes for the simple drawing system.

The first subclass represents the Text window, and the second represents the Drawing window. In turn, the Drawing window acts as a superclass to three additional subclasses. These include a class representing a line figure, a class representing a rectangular figure, and a class representing an oval figure. We assume that the superclass `Window` has two members that it must know: the boundary of the window for an object of this type of class and the boundary for a geometrical shape that is to be drawn. These boundaries are represented by an upper-left point and a lower-right point, referred to as *corners*. For a geometrical shape, the upper-left point is selected when the mouse button is pressed, and the lower-right point is selected when the mouse button is released. After the selection of the upper-left point, the button is held down while the mouse is dragged down and to the right. Then the lower-right point is selected with the release of the button. We assume that an oval or rectangle must always be drawn from an upper-left point to a lower-right point of the Drawing window.

The superclass `Window` also defines three methods: one for establishing the boundary of a window, a second for showing the window, and a third for drawing an object within a window. This third method is responsible for capturing the upper-left and lower-right corners bounding the graphical shape being drawn. We assume that a portion of its definition is overridden by one or more descendent subclasses having a greater knowledge of the types of graphical objects to be drawn. Members listed in the class declaration as well as the methods defined are assumed to represent the minimal amount of information required by this abstract class. In some cases these methods are overridden by the methods of subclasses having greater knowledge of the type of object that the subclass is categorizing.

The two subclasses `Text_Window` and `Drawing_Window` are more specific types of classes that inherit from the abstract class `Window`. The subclass `Drawing_Window` has no members, but the subclass `Text_Window` has as a member a `string` array for storing messages. This `string` array stores text (as a primitive type of graphical object) for drawing instructions to the Text window. Both classes define two methods. One method is for showing the window, and the second is for drawing some type of graphical object (or text) for an instance of this type of class. Both methods override those defined in the superclass `Window`.

Another set of subclasses is the set `Line`, `Rectangle`, and `Oval`. Instances of these subclasses represent objects that can reside within the instances of the superclass `Drawing_Window`. While they provide no added members, each is assumed to have its own method of drawing an object in addition to the methods defined by its ancestor classes, `Drawing_Window` and `Window`. What is inherited from the ancestor classes is the set of primitive actions that the present methods need not perform. This arrangement provides a means of sharing responsibilities through a hierarchy of classes, without the need for excessive parameter passing among messages.

The following code represents the unit for declaring the abstract class `Window`, and the definitions of all of its methods. Note that we always try to keep each method as functional as possible and to practice tight coupling:

```
unit Abstract_Class_Window;
{ Purpose:  This unit supports the definition of a class called }
{           Window. It uses as its superclass TObject.}
interface
uses
   ObjIntf;
type
```

```
   Window = object(TObject)
         { Instance Variable Declarations }
            Window_Boundary: Rect;
            Boundary_for_Figure: record
                  Upper_Left_Corner: Point;
                  Lower_Right_Corner: Point;
               end;
         { Method Declarations }
            procedure Set_Window_Boundary (Top, Left,
                        Bottom, Right: integer);
            procedure Show_Window;
            procedure Draw_Object;
         end;

implementation

procedure Window.Set_Window_Boundary;
{ Purpose:  This routine establishes the boundary for a window.}
begin
   SetRect(Self.Window_Boundary, Top, Left, Bottom, Right);
end;

procedure Window.Show_Window;
{ Purpose: This routine shows the window to the viewer.}
begin
   { This method is a stub that is to be overridden by a }
   { subclass.}
end;

procedure Window.Draw_Object;
{ Purpose:  This routine draws a simple figure from an }
{           upper-left corner to a lower-right corner.}
   begin
   { Wait for the mouse button to be pressed.}
      while not Button do
         ;
   { Get the upper-left point for drawing a figure.}
      GetMouse(Self.Boundary_for_Figure.Upper_Left_Corner);
   { Get the lower-right point for drawing a figure by waiting }
   { for the button to be released.}
      while Button do
         ;
      GetMouse(Self.Boundary_for_Figure.Lower_Right_Corner);
   { In drawing the object, this part will be overridden by a }
   { definition of a subclass.}
   end;
end.
```

The subclasses `Text_Window` and `Drawing_Window` are defined by a unit `Window_Subclasses`. Here, subclass `Text_Window` supports a `string` array for storing lines of text. Both subclasses support two methods: one for showing the window and a second for drawing an appropriate object on the screen. For the subclass Text_Window, this object will be lines of text, whereas for the subclass Drawing_Window, it will be a simple geometrical figure.

```
unit Window_Subclasses;
{ Purpose: This unit defines two subclasses of superclass Window.}
interface
uses
   ObjIntf, Abstract_Class_Window;
type
   Text_Window = object(Window)
         { Instance Variable Declaration }
              String_Table: array[1..5] of string;
         { Method Declarations }
              procedure Show_Window;
              override;
              procedure Draw_Object;
              override;
         end;

   Drawing_Window = object(Window)
         { Method Declarations }
              procedure Show_Window;
              override;
              procedure Draw_Object;
              override;
         end;

implementation
{ ============================================================== }
procedure Text_Window.Show_Window;
{ Purpose:  This routine sets the boundary and displays a Text }
{           window on the screen.}
   begin
      SetTextRect(Self.Window_Boundary);
      ShowText;
   end;

procedure Text_Window.Draw_Object;
{ Purpose: This routine displays lines of text to the screen.}
   var
      Index: integer;
   begin
      Index := 1;
      Page;
      while String_Table[Index] <> 'END_STRING' do
         begin
            writeln(String_Table[Index]);
            Index := succ(Index);
         end;
   end;
```

```
{ ============================================================== }
procedure Drawing_Window.Show_Window;
{ Purpose:  This routine sets the boundary and displays the } {
            Drawing window on the screen.}
   begin
      SetDrawingRect(Self.Window_Boundary);
      ShowDrawing;
   end;

procedure Drawing_Window.Draw_Object;
{ Purpose:  This routine inherits some of its actions for }
{           drawing an object from class Windows.}
   begin
      inherited Draw_Object;
      { Remaining actions for drawing an object will be }
      { overridden by a definition of a subclass.}
   end;
end.
```

The third unit, Figure_Classes, defines three subclasses representing three basic geometric figures. These figures are Line, Oval, and Rectangle. Although none of these three subclasses adds information by way of instance variables, each subclass defines only one method, Draw_Object. Each individual method draws one of three types of figures to the screen. Notice that each method overrides Draw_Object from subclass Drawing_Window. In turn, Drawing_Window overrides the method Draw_Object from the abstract class Window. When an object of subclass Line, Rectangle, or Oval sends the message Draw_Object, it applies the inherited steps from Draw_Object in the ancestor class Drawing_Window before completing its own definition. In turn, the method Draw_Object in subclass Drawing_Window uses the definition Draw_Object from the abstract class Window. Method Draw_Object in class Window establishes the upper-left and lower-right corners for bounding the area in which a figure is drawn. The following unit provides declarations and definitions for the subclasses Line, Rectangle, and Oval:

```
unit Figure_Classes;
{ Purpose:  This unit defines three subclasses representing }
{           simple geometric figures.}
interface
uses
   ObjIntf, Abstract_Class_Window, Window_Subclasses;

type
   Line = object(Drawing_Window)
            procedure Draw_Object;
            override;
         end;

   Rectangle = object(Drawing_Window)
            procedure Draw_Object;
         override;
      end;

   Oval = object(Drawing_Window)
         procedure Draw_Object;
         override;
      end;

implementation

procedure Line.Draw_Object;
{ Purpose:  This routine draws a line from an upper-left to a }
{           lower-right point.}
   begin
      inherited Draw_Object;
      DrawLine(Self.Boundary_for_Figure.Upper_Left_Corner.h,
         Self.Boundary_for_Figure.Upper_Left_Corner.v,
         Self.Boundary_for_Figure.Lower_Right_Corner.h,
         Self.Boundary_for_Figure.Lower_Right_Corner.v)
   end;

{ ============================================================ }
procedure Oval.Draw_Object;
{ Purpose:  This routine draws an oval bounded by upper-left and }
{           lower-right corners.}
   var
      Rectangle_for_Oval: Rect;
   begin
      inherited Draw_Object;
      Rectangle_for_Oval.Top  :=
         Self.Boundary_for_Figure.Upper_Left_Corner.v;
      Rectangle_for_Oval.Left :=
         Self.Boundary_for_Figure.Upper_Left_Corner.h;
      Rectangle_for_Oval.Bottom :=
         Self.Boundary_for_Figure.Lower_Right_Corner.v;
      Rectangle_for_Oval.Right :=
         Self.Boundary_for_Figure.Lower_Right_Corner.h;
      FillOval(Rectangle_for_Oval, black);
      InvertOval(Rectangle_for_Oval.Top + 2,
         Rectangle_for_Oval.Left + 2,
         Rectangle_for_Oval.Bottom - 2,
         Rectangle_for_Oval.Right - 2);
   end;

{ ===================================================== }
procedure Rectangle.Draw_Object;
{ Purpose:  This routine draws a rectangle bounded by }
{           upper-left and lower-right corners.}
   var
      Rectangle_for_Rect: Rect;
   begin
```

```
      inherited Draw_Object;
      Rectangle_for_Rect.Top :=
         Self.Boundary_for_Figure.Upper_Left_Corner.v;
      Rectangle_for_Rect.Left :=
         Self.Boundary_for_Figure.Upper_Left_Corner.h;
      Rectangle_for_Rect.Bottom :=
         Self.Boundary_for_Figure.Lower_Right_Corner.v;
      Rectangle_for_Rect.Right :=
         Self.Boundary_for_Figure.Lower_Right_Corner.h;
      FillRect(Rectangle_for_Rect, black);
      InvertRect( Rectangle_for_Rect.Top + 2,
         Rectangle_for_Rect.Left + 2,
         Rectangle_for_Rect.Bottom - 2,
         Rectangle_for_Rect.Right - 2);
   end;
end.
```

As you can see from this unit's implementation of the methods, each method inherits the primitive actions from its ancestors while defining the specific actions for completing its definition. This is one of the advantages of using object Pascal.

The following unit provides supporting routines for presenting instructions to a Text window. Both procedures have a formal parameter of type Text_Window:

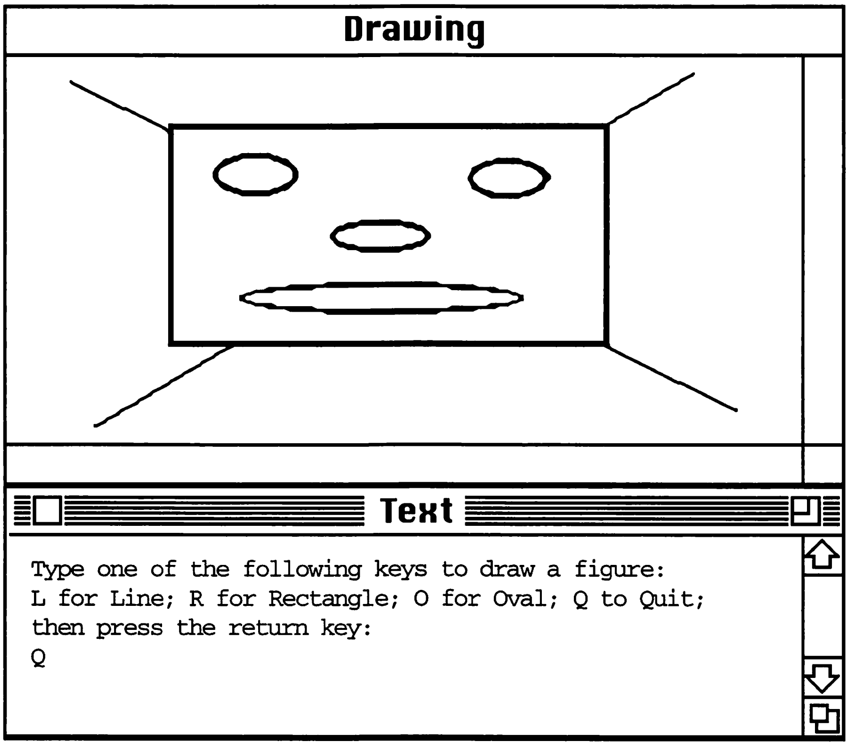
```
unit Instructions;
{ Purpose:  This unit has two supporting routines for passing }
{           instructions to an instance of Text_Window.}
interface
uses
   ObjIntf, Abstract_Class_Window, Window_Subclasses,
         Figure_Classes;

procedure Instructions_for_Choosing_Figure
                (A_Text_Window:Text_Window);
procedure Instructions_on_Drawing_Figure(A_Text_Window:
                Text_Window);

implementation
procedure Instructions_for_Choosing_Figure;
{ Purpose:  This routine passes instructions to an object of }
{           type Text_Window.}
   begin
      with A_Text_Window do
         begin
            String_Table[1] :=
      'Type one of the following keys to draw a figure:';
            String_Table[2] :=
      'L  for Line; , R for Rectangle, O for Oval; Q to Quit;';
            String_Table[3] := 'then press the return key: ';
            String_Table[4] := 'END_STRING';
         end;
   end;

procedure Instructions_on_Drawing_Figure;
{ Purpose:  This routine passes instructions to an object of }
{           type Text_Window.}
   begin
      with A_Text_Window do
         begin
         String_Table[1] :=
   '  Select the top  left corner, then press the mouse button.';
         String_Table[2] :=
   '  Hold  the  mouse  button down until you have selected the ';
         String_Table[3] :=
   '  right point. Then release the button. Use the halt button';
         String_Table[4] :=
   '  if you wish to interrupt execution.';
         String_Table[5] := 'END_STRING';
         end;
   end;
end.
```

The following program drives the simple drawing system described above. As you can see, it begins by hiding all windows. It then establishes two instances of subclasses: an instance representing the Text window as an object, and an instance representing the Drawing window as an object. For both objects, the boundaries of the windows are set and shown by using the messages `Set_Window_Boundary` and `Show_Window`. The program then repeats instructions for selecting a figure and for drawing the figure in the Drawing window. When drawing any of the three types of geometrical figures, its instance is created, drawn, and then disposed of by using the method `Free` from class `TObject`. This suggests a rule of keeping an instance in existence for only as long as it is required. It should be disposed of immediately when it no longer is needed.

Once the choice is made to quit, each of the window instances is destroyed. This program differs from a procedural program in the lack of dependency on parameter passing. In most instances where messages are involved, no parameter passing is required. This is possible because all of the values an object must know are integrated internally in the object as it is used dynamically by the program. At no point in the program do we need to directly reference a member of an object. All references are hidden through method definitions.

```
program Drawing_Basic_Figures (input, output);
{ Purpose:  This program deals with instances of a Text and }
{           Drawing window, and with instance of a figure such }
{           as a line, oval, or rectangle.}
uses
   ObjIntf, Abstract_Class_Window, Window_Subclasses,
   Figure_Classes, Instructions;
var
{ Instance-references }
   A_Text_Window: Text_Window;
   A_Drawing_Window: Drawing_Window;
   A_Line_Figure: Line;
   An_Oval_Figure: Oval;
   A_Rectangular_Figure: Rectangle;
{ Simple Identifier }
   Figure_Type: char;
begin
{ Hide all windows before showing Text and Drawing windows.}
   Hideall;
{ Establish instances of objects for a Text and a Drawing window.}
   new(A_Text_Window);
   new(A_Drawing_Window);
{ Establish the boundary and show the Text window on the screen.}
   A_Text_Window.Set_Window_Boundary(1, 260, 512, 335);
   A_Text_Window.Show_Window;
{ Establish the boundary and show the Drawing window on the }
{ screen.}
   A_Drawing_Window.Set_Window_Boundary(1, 40, 512, 240);
   A_Drawing_Window.Show_Window;
   repeat
   { Establish lines of text for viewing in the Text window.}
      Instructions_for_Choosing_Figure(A_Text_Window);
   { Continue to display instructions until a proper figure-type }
   { is selected.}
      repeat
         A_Text_Window.Draw_Object;
         readln(Figure_Type);
      until (Figure_Type = 'L') or (Figure_Type = 'O') or
         (Figure_Type = 'R') or (Figure_Type = 'Q');
   { Test if the option is to quit.}
      if (Figure_Type <> 'Q') then
         begin
      { Provide instructions to the user for drawing a figure }
      { within the Drawing window.}
            Instructions_on_Drawing_Figure(A_Text_Window);
            A_Text_Window.Draw_Object;
      { Draw the selected figure in the Drawing window. }
            case Figure_Type of
            'L':  { Draw a line from point to point.}
               begin
                  new(A_Line_Figure);
                  A_Line_Figure.Draw_Object;
                  A_Line_Figure.Free
               end;
            'O':  { Draw an oval within a rectangle given by }
            { left and right mouse points.}
               begin
                  new(An_Oval_Figure);
                  An_Oval_Figure.Draw_Object;
                  An_Oval_Figure.Free;
               end;
            'R':  { Draw a rectangle within a rectangle given by }
            { left and right mouse points.}
               begin
                  new(A_Rectangular_Figure);
                  A_Rectangular_Figure.Draw_Object;
                  A_Rectangular_Figure.Free;
               end;
            'Q':
               ;
            end;
         end;
   until (Figure_Type = 'Q');
{ Free storage of all window instances before ending execution.}
   A_Text_Window.Free;
   A_Drawing_Window.Free;
end.
```

Figure 13.6 shows a snapshot taken after executing the program Drawing_Basic_Figures.



```
Type one of the following keys to draw a figure:
L for Line; R for Rectangle; O for Oval; Q to Quit;
then press the return key:
Q
```

Figure 13.6 A snapshot after executing the simple drawing program.

13.7 USING THE CLASS BROWSER AND LIGHTSBUG FOR VIEWING OBJECTS AND CLASSES

THINK Pascal supports a helpful tool for examining files containing class declarations. Called a Class Browser, this tool allows us to view the hierarchical structure of classes defined within a THINK Pascal project. It also allows us to locate files containing the declaration of classes as well as the definitions of methods.

The Class Browser for any project is chosen by selecting **Class Browser** from the menu option **Windows**, which presents a special window, a Class Browser window, as the active window. Like other Macintosh windows, this window can be sized, dragged, and closed, and has both horizontal and vertical scroll bars. Figure 13.7 shows the Class Browser window after it has been applied to the project of the simple drawing system.

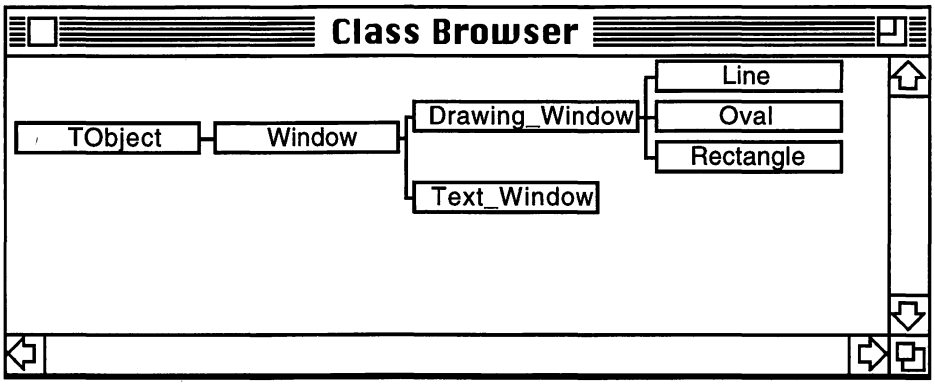


Figure 13.7 Class Browser window for the simple drawing system.

While the Class Browser window is active, the source file for any class name given in the tree chart can be opened (and the window scrolled to where the class declaration begins) by simply double-clicking on the rectangle containing the name of the class.

We can also locate the declaration for a class when in the editor of THINK Pascal by holding down the option key and at the same time highlighting the name of the class. When the mouse button is released, the source file containing the declaration appears in an active editing window. If the source file is lengthy, the Edit window will scroll to where the class is declared.

We can locate method definitions from either the Class Browser or from the editor. On choosing the Class Browser, we can display the names of methods associated with a class through a pop-up menu appearing to the right of class name. This pop-up menu is opened by highlighting the name of a class. Figure 13.8 shows a pop-up menu containing the names of methods declared by class `Text_Window`.

Highlighting the name of the method in the pop-up menu and releasing the mouse button opens the source file containing the definition of a method, with the Edit window scrolled to where the definition begins. A pop-up menu never includes methods inherited from other classes. The definition of a method can also be located from the editor of THINK Pascal by holding down the option key and at the same time highlighting the name of the method. Releasing the mouse button displays the source file containing the method definition in an active editing window. Again, if the source file is lengthy, the Edit window will scroll to where the method is defined. If a method name is highlighted and more than one class has a definition, the Class Browser window is opened and shows those classes that contain definitions.

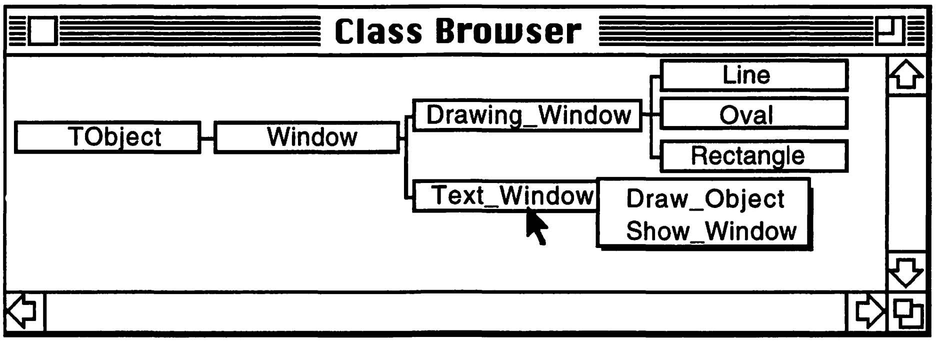


Figure 13.8 Listing the method names of a class by using the Class Browser window.

Figure 13.9 shows the Class Browser window that has been opened after highlighting the message name `Draw_Object` in program `Drawing_Basic_Figures`. From the Class Browser window, we can access the method definition given by a particular class.

Several keyboard shortcuts exist for moving through a class hierarchy in the Class Browser window. Pressing the enter or return key is the same as double-clicking. The Up-arrow key will select the previous sibling of a current class, and the Down-arrow key selects the next sibling of a current class. The Left and Right arrow keys select a superclass or subclass, respectively, and the Tab traverses each class in the class hierarchy.

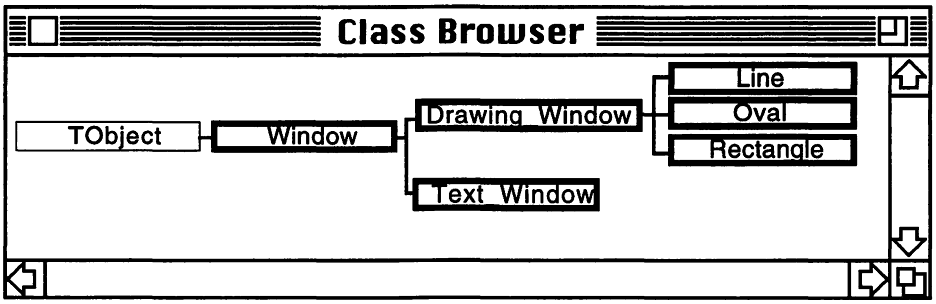
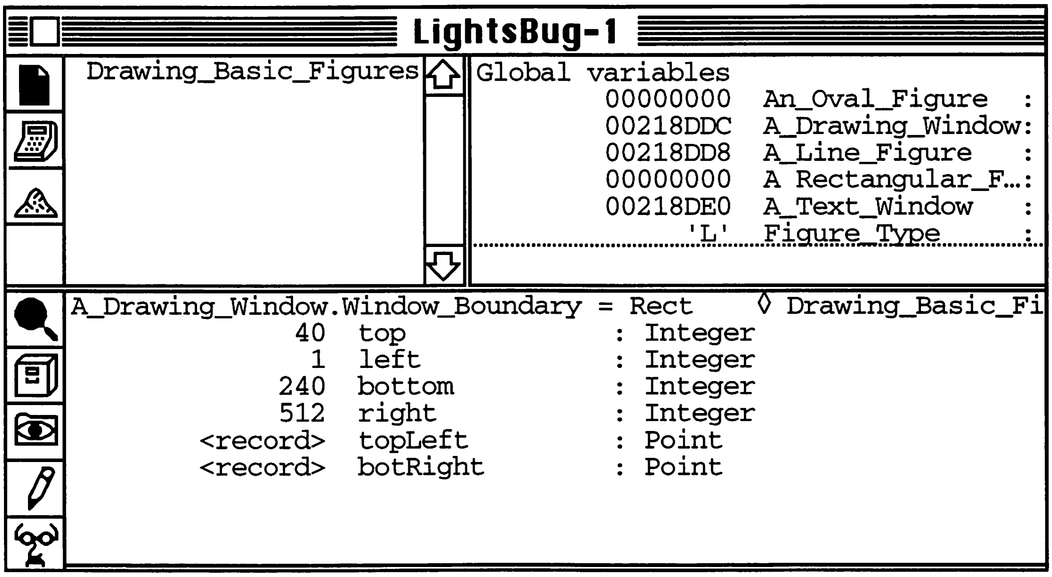


Figure 13.9 In this example, the Class Browser indicates that the method `Draw_Object` is defined in more than one class.

Various options for debugging an object Pascal program are available when using the THINK Pascal system. By setting Option D for any file that is currently in the Project window, we can use the LightsBug debugger to examine the memory content of any object listed in the upper-right pane of the LightsBug window. Other options can be set, such as Option N, for viewing the actions of routine; V, for checking for overflow; and R, for range-checking. Figure 13.10 shows the LightsBug window, where the program

Drawing_Basic_Figures has been stopped to view the content of the object A_Drawing_Window.



```
Drawing_Basic_Figures        Global variables
                                  00000000  An_Oval_Figure   :
                                  00218DDC  A_Drawing_Window:
                                  00218DD8  A_Line_Figure    :
                                  00000000  A Rectangular_F...:
                                  00218DE0  A_Text_Window    :
                                       'L'  Figure_Type      :

A_Drawing_Window.Window_Boundary = Rect     ◊ Drawing_Basic_Fi
                  40  top              : Integer
                   1  left             : Integer
                 240  bottom           : Integer
                 512  right            : Integer
            <record>  topLeft          : Point
            <record>  botRight         : Point
```

Figure 13.10 Using the LightsBug window to examine the content of an object.

13.8 USING OBJECT PASCAL TO IMPLEMENT AN OBJECT LIST

In THINK Pascal, objects are treated as handles. Handles are indirect references using pointers. While in standard Pascal pointers can implement a list as a sequence of linked nodes, object Pascal can implement a list of polymorphic objects representing an abstract data type. The list is by itself an instance of a class. Figure 13.11 illustrates the idea of an object list, beginning with a special object representing the header for the list. The remaining elements represent polymorphic objects, with each object encapsulating an

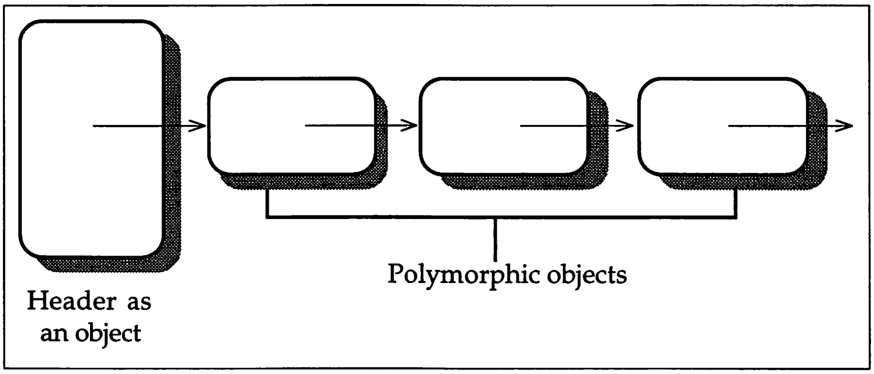


Figure 13.11 Graphic representation of a list of polymorphic objects.

instance variable capable of referencing another polymorphic object. An object list differs from a normal linked list of nodes in the way one object is connected with another. In an object list, each object encapsulates a reference to the next object in the list. The instance variable that references another object is not a pointer type. Rather, the link from one object to another is achieved by making an instance variable a class type.

The implementation that follows defines a class for supporting the declaration of a polymorphic object, as well as a class for supporting the declaration of a header object. The unit that follows specifies the class for declaring our polymorphic objects:

```
unit Link_Class_Definition;
{Purpose:    This unit defines a class for declaring a }
{            polymorphic object.}
interface
uses
   ObjIntf;
type
   Object_Class = object(TObject)
      { Instance Variable Declaration }
            Next_Object: Object_Class;
      { Method Declarations }
            procedure Establish_Link (Next_Object:
                           Object_Class);
            function Next_List_Object: Object_Class;
      end;
implementation

procedure Object_Class.Establish_Link (Element: Object_Class);
{ Purpose:  This routine assigns a reference for the object }
{           Element passed to this routine to the instance }
{           variable Next_Object of an object encapsulating }
{           this method.}
   begin
      Self.Next_Object := Element;
   end;

function Object_Class.Next_List_Object: Object_Class;
{ Purpose:  This routine returns a reference to the next object }
{           in the list.}
   begin
      Next_List_Object := Self.Next_Object;
   end;
end.
```

Although the definitions of the methods are short, they are highly functional for the actions that they perform. Only the instance variable `Next_Object` is necessary, because this class represents only the minimal information for declaring a polymorphic object.

The next unit defines a class for an object declared as a list. An instance of this class will represent the header to the object list:

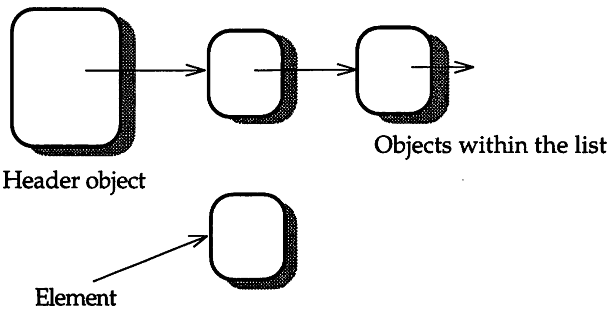
```
unit List_Class_Definition;
{Purpose: This unit defines a class for an object list.}
interface
uses
   ObjIntf, Link_Class_Definition;

type
   List_Class = object(TObject)
      { Instance Variable Declaration }
         Link: Object_Class;
      { Method Declarations }
         procedure Initialize_List;
         procedure Push_Object_To_List (Element: Object_Class);
         function Pop_Object_From_List: Object_Class;
         function Inspect_Header_List: Object_Class;
      end;

implementation
{ Definitions for the different methods are given in the }
{ discussion that follows.}
...
end.
```

As you can see, the class for a list has only one instance variable. This variable is for referencing the first object in the list. The class for a list also has four method definitions for defining the minimal actions for anobject of this class. First, the method `Initialize_List` establishes an empty object list by assigning the instance variable `Link` in the header object the value **nil**. The following procedure defines the actions for this method:

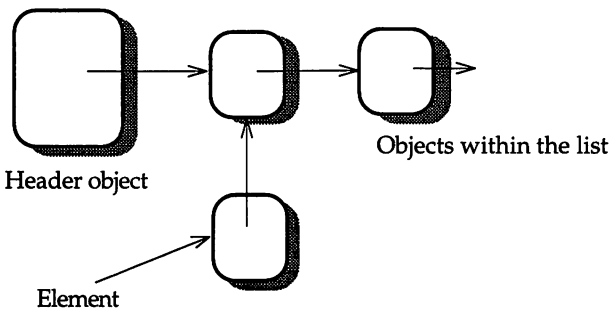
```
procedure List_Class.Initialize_List;
{ Purpose:  This routine initializes an empty object list. }
   begin
      Self.Link := nil;
   end;
```

The instance variable `Link` in the header object must be modified each time an object is added at the front of list. Figure 13.12 demonstrates the basic steps in adding a new polymorphic object at the front of an object list. These steps provide the algorithm for inserting a new polymorphic object at the front of a list. The definition of the method that follows is different only in that it tests for an empty list. If the instance variable `Link` of the header object is **nil**, the instance variable `Next_Object` of the new object being added is assigned the value **nil.** If the list is not empty, the new object is added at the front of the list. The following is a detailed definition of the method `Push_Object_To_List`.

1. Create a new object using the command `new`.



2. Set the instance variable `Next_Object` of `Element` to reference the object at the front of the list by sending the message `Element.Establish_Link` with `Self.Link` as its actual parameter. `Self` represents the header object.



3. Let the instance variable `Link` of the header object reference the new object Element.

Self.Link <-- Element

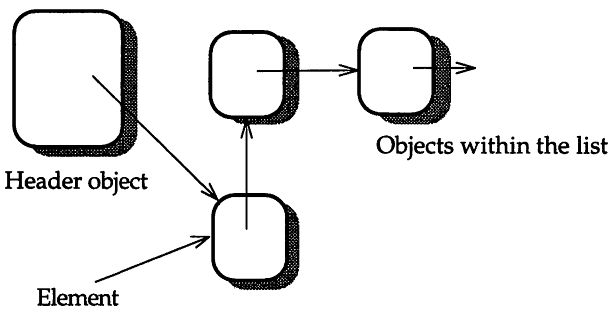


Figure 13.12 Basic steps in adding an object at the beginning of a polymorphic list.

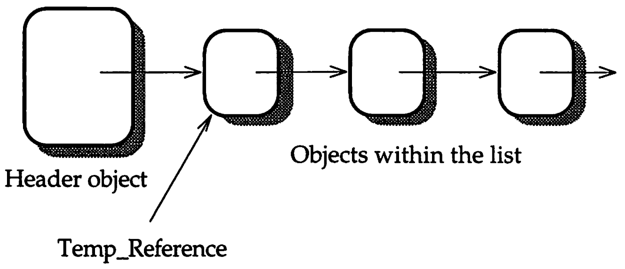
```
procedure List_Class.Push_Object_To_List (Element:
Object_Class);
{Purpose:   This routine adds a polymorphic object at the front }
{           of an object list.}
   begin
   { If the list is empty, set the instance variable for }
   { referencing another object to nil.}
      if Self.Link = nil then
         Element.Establish_Link(nil)
      else
      { If the list is not empty, let the instance variable for }
      { referencing another object in Element reference the }
      { object given by the instance variable Link of the header }
      { object.}
         Element.Establish_Link(Self.Link);
   { Now let the instance variable Link of the header object }
   { reference the new object added to the list.}
      Self.Link := Element;
   end;
```

For deleting an object from the front of an object list, we reverse the steps for adding an object. Figure 13.13 shows an algorithm that removes an object. The steps require establishing a temporary reference variable of type `Link_Class` and requiring it to reference the first object at the front of the list. The second step calls for adjusting the reference in the instance variable `Link` of the header object so that it references the object following the first object in the list. The reference associated with the temporary reference variable can now be returned. The following code defines the method `Pop_Object_From_List`:

```
function List_Class.Pop_Object_From_List: Object_Class;
{ Purpose: This routine removes the front object from a list of
objects.}
   var
      Temp_Reference: Object_Class;
   begin
   { If the list is not empty, remove the object at the front }
   { of the list.}
      if Self.Link <> nil then
         begin
         { Establish a temporary reference to the first object }
         { in the list.}
            Temp_Reference := Self.Link;
         { For the front object, assign the next object that it }
         { references to the instance variable Link of the }
         { header object.}
            Self.Link := Temp_Reference.Next_List_Object;
         { Return the reference of the front object for later }
         { disposal.}
            Pop_Object_From_List := Temp_Reference;
         end
      else
         Pop_Object_From_List := nil;
   end;
```

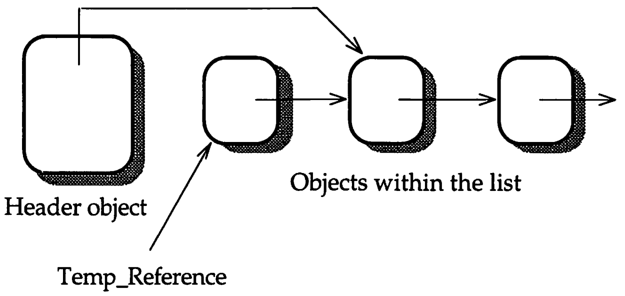
1. Provided that the list is not empty, assign the reference of the instance variable `Link` of the header object to a temporary variable:

   Temp_Reference <-- Self.Link



2. Let the instance variable `Link` of the header object reference the object following the first object of the list. This is done by letting the instance variable `Link` of the header object reference the next link object of the first list object, by using the method `Next_List_Object`:

   Self.Link <-- Self.Link.Next_List_Object



3. Now return the value of the variable `Temp_Reference` for accessing the content of the object and disposing of storage.

Figure 13.13 Basic steps for deleting an object from the front of a polymorphic list.

Notice that if the list is empty, the function returns **nil**. In essence, it is equivalent to the condition of attempting to "underflow" the list. In this example, no effort is made to test for this condition.

The function that follows defines the fourth method in class `List_Class`. Its purpose is to inspect the list and return a reference to the first object in the list. As you can see, the key step in executing this method is to return the value of the instance variable `Link` of the header object.

```
function List_Class.Inspect_Header_List: Object_Class;
{ Purpose:  This routine returns the value of the instance }
{           variable Link in the header object.}
   begin
      Inspect_Header_List := Self.Link;
   end;
```

The following program provides a simple test for Object_Class and List_Class. It defines four new classes as well as several different objects. The object An_Object_List represents the header object for the list. Reference is a temporary object for referencing an object removed from the front of the list and for disposing of its storage once the data has been accessed. Four additional objects are included: Object_A, Object_B, Object_C, and Object_D. These objects represent a class having an ancestor class that is polymorphic. Notice that the function member is used on the variable Reference to test the class associated with the object being referenced and to ask if the class associated with each object it references is one of three defined classes: Class_A, Class_B, or Class_C. Knowing if Reference is one of these classes allows the instance variable Item of an object to be properly referenced. The object variable Reference is of a polymorphic class, so it must be cast into the proper descendant class type before the instance variable Item in the object is accessed. Method Free from TObject is applied to dispose of the dynamic storage for the object referred to by Reference. Method Free is also applied to the header object before the program ends execution.

```
program Test_Object_List (input, output);
{Purpose:   This program builds an object list in which each   }
{           object is of a different class.}
uses
   ObjIntf, Link_Class_Definition, List_Class_Definition;
type
   Class_A = object(Object_Class)
            Item: integer;
      end;
   Class_B = object(Object_Class)
            Item: real;
      end;
   Class_C = object(Object_Class)
         Item: char;
      end;
 Class_D = object(Object_Class)
         Item: string;
      end;
var
   An_Object_List: List_Class;
   Object_A: Class_A;
   Object_B: Class_B;
   Object_C: Class_C;
   Object_D: Class_D;
   Reference: Object_Class;
```

```
begin
{ Show the Text window for viewing data.}
   ShowText;
{ Create the header for an object list and initialize the list }
{ as empty.}
   new(An_Object_List);
   An_Object_List.Initialize_List;
{ Create the first object and add it to the list.}
   new(Object_A);
   Object_A.Item := 123;
   An_Object_List.Push_Object_To_List(Object_A);
{ Create and add the second object to the list.}
   new(Object_B);
   Object_B.Item := 987.6543;
   An_Object_List.Push_Object_To_List(Object_B);
{ Create and add the third object to the list.}
   new(Object_C);
   Object_C.Item := 'A';
   An_Object_List.Push_Object_To_List(Object_C);
{ Create and add a fourth object to the list.}
   new(Object_D);
   Object_D.Item := 'This is a string.';
   An_Object_List.Push_Object_To_List(Object_D);
{ Inspect the instance variable Link of the header of the object }
{ list and, while the list is not empty, remove the object from }
{ the list, access its  data, and dispose of storage for the }
{ object.}
   while An_Object_List.Inspect_Header_List <> nil do
      begin
      { Remove the next object at the front of the linked }
      { object list and dispose of the object.}
         Reference := An_Object_List.Pop_Object_From_List;
         if member(Reference, Class_A) then
            writeln('Output from object of class A: ',
               Class_A(Reference).Item)
         else if member(Reference, Class_B) then
            writeln('Output from object of class B: ',
               Class_B(Reference).Item : 10 : 4)
         else if member(Reference, Class_C) then
            writeln('Output from object of class C: ',
               Class_C(Reference).Item)
         else
            writeln('Reference is no known class type.');
         Reference.Free;
   end;
{ Dispose of the header of the linked object list. }
   An_Object_List.Free;
end.
```

Selecting **Run Options...** from the menu option **Run** and choosing a file name for echoing the text from the Text window captures the following lines when the above program is executed:

```
Reference is no known class type.
Output from object of class C: A
Output from object of class B:   987.6543
Output from object of class A:      123
```

In the above discussion we avoided using the phrase "linking an object to another object." The objects declared in the test program and in these method definitions are not pointers in THINK Pascal. They are variables of class types and for that reason are stored internally as handles. They are not pointers that reference the location of an object stored in memory. Rather, they reference an object that encapsulates data and methods for acting upon the object. While this approach appears to be more abstract when interpreting the steps performed by various methods defined in this section, it is actually as simple as the model of pointers connecting nodes within a linked list.

In writing an algorithm for searching an object list, it is important to test if the next object being examined matches a known class type and, if it does, to test if a search key matches the key within the object being referenced. If so, the proper object has been found. In THINK Pascal this requires us to define a function for testing if an object is one of several defined classes. To do so, we must use the built-in function `member`.

13.9 SOFTWARE ISSUES IN USING OBJECT PASCAL

In object-oriented analysis and design we often work with classes that have multiple inheritances. Such classes are characterized by inheriting from more than one immediate ancestor. Figure 13.14 shows that class C inherits from two different superclasses A and B.

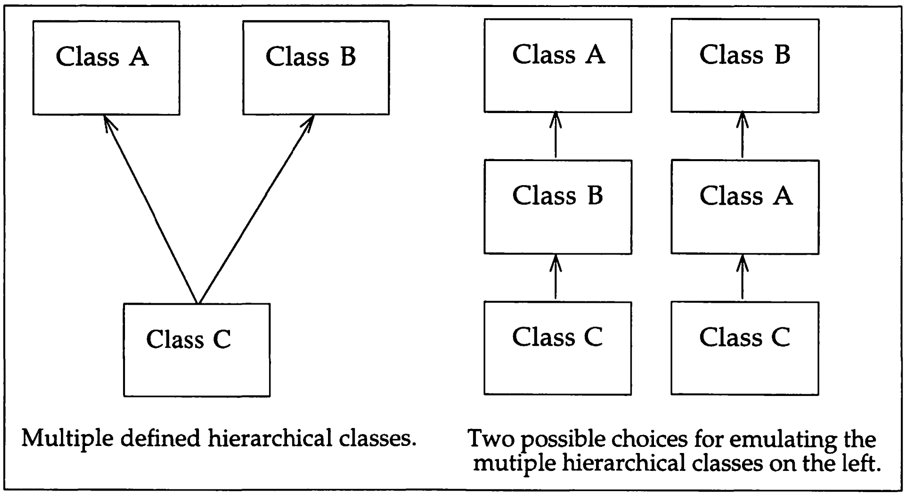

Class A
Class B
Class C
Class A
Class B
Class B
Class A
Class C
Class C
Multiple defined hierarchical classes.
Two possible choices for emulating the mutiple hierarchical classes on the left.

Figure 13.14 Implementing a multiple hierarchical chart through single class inheritance.

In this example, class C inherits some of its properties and methods from class A and others from class B, which raises some issues that are difficult to resolve. For example, both the superclasses A and B declare a method having the same name but different definitions. Which method is invoked by class C if it inherits all of it methods from its superclasses is difficult to define and often cannot be resolved by the language in which the application is implemented. Fortunately, object Pascal can only inherit from a single class and does not support the principle of multiple inheritance. In object Pascal, multiple inheritance can only be emulated through a single chain of classes, as shown at the right in Figure 13.14. Although class C inherits all of the characteristics of its ancestors, either class B must define more unnecessary information for class A or class A must define more unnecessary information for class B.

Often it is important for a method declared by a class that has no direct chain of inheritance to invoke a method defined by another class. In developing an application, it is more important for classes to share some of their responsibilities rather than duplicating their definitions. By allowing the definition of one method of a class to invoke a method of an unrelated class, an object of one class can contract with an object of another to perform actions unknown to itself. This is similar to the real world, where people must often contract outside their own environment in order to perform actions outside their expertise. For example, a college professor might contract with a plumber to get a faucet repaired. This approach allows objects to collaborate with each other without the need for sharing classes through inheritance. It attempts to distribute the intelligence of the application among several unrelated classes. Classes and objects may thus know relatively fewer facts but allow us to produce a more flexible application that is easier to modify in the future.

This principle is implemented in THINK Pascal by employing parameter passing, where a formal parameter referencing an object is of a particular class type. In the following code segment, method One of class A allows an object of class B to be passed. In turn, method One invokes method Two defined by class B:

```
type
   Class_A = object
        ...

      procedure One( Object_Reference : Class_B );
      end;

   Class_B = object
        ...

        procedure Two;
      end;

{ implementation }

   procedure Class_A.One;
      begin
        ...

        Object_Reference.Two;
      end;
```

```
procedure Class_B.Two;
   begin
      ...
   end;
```

It is also important that any class (as well as an object of any class) support the principle of encapsulation. Encapsulation implies that the class encases both data and methods. This requires us to define methods that directly access the values of any instance variables. All direct references to instance variables are contained within the definition of a method. All external accessing of instance variables occurs by sending a message to an object, where the method being invoked accesses the instance variable. Thus the object takes on the appearance of a unique entity. All methods known by the object should be highly functional and easy to comprehend. In turn, the state of the object is understood by the messages sent to it without having to recognize a lengthy list of variables embedded within the object declaration. The programmer only needs to know what is in the messages he or she needs to send to the object.

SUMMARY

In object-oriented programming, a class represents a declaration that encapsulates the data and methods that define actions on its own data. Through an instance of a class, an object of a class becomes a living entity. The properties defined by the class for an object are the properties associated with the object. The object is different from a simple variable or record structure. It has state through values of its instance variables and is capable of being sent messages for performing actions. In developing software applications, the classes form a hierarchical chart; subclasses can inherit properties from superclasses. The design of an application appears as a set of relationships between classes, much as in the real world. The need for procedural hierarchies among procedures and functions defining methods for a class becomes secondary to design and in some instances unimportant. In design, the principle of tight coupling of instance variables with highly functional method definitions is of primary importance. The class and the object encapsulate intelligence. The object acts as an entity providing intelligence to the application that is being executed.

THINK Pascal supports object Pascal programming through object declarations and method definitions. Through the use of the reserved word **object**, we can declare class types containing fields referred to as instance variables as well as header declarations for methods. By using separate program units, we can declare classes and construct a class hierarchy. The superclasses toward the top of the hierarchy are more abstract, and descendant classes toward the bottom are more concrete. In THINK Pascal, objects are implemented as handles: pointer types that employ double indirection. To allow efficient use of objects as handles, THINK Pascal supports a unique abstract class called `TObject`. `TObject` defines two special methods: `Clone`, which allows an object to be properly cloned (duplicated) as a handle, and `Free`, which properly disposes of the dynamic storage associated with an object. The procedure `new` allows storage for objects to be constructed.

By selecting **Class Browser** from menu option **Windows**, we can view the hierarchical relationship of all classes defined for a project. By clicking on the name of a class within the Class Browser window, we can activate the file containing the class, making it the active Edit window, with the screen scrolled to where the class declaration begins. Through a pop-up menu, the Class Browser window can also list the names of all methods associated with any class in the project. If we highlight a method name and release the mouse button, the file containing the method definition appears in the active Edit window, with the screen scrolled to where the method declaration begins. By setting stops, the LightsBug debugger can be used to trace the values of the instance variables of any object within the project. As with other simple and structured variables, the values of instance variables can be redefined and execution continued.

REVIEW QUESTIONS

1. What is meant by the term *object-oriented programming*?
2. What is a class?
3. Why is a class considered abstract and not an entity by itself?
4. What is an object?
5. How is a class declared?
6. Why is a declaration for a class different from that of a record?
7. How is a method declared?
8. How is a method defined?
9. Why is object Pascal different from standard Pascal?
10. What is an instance variable?
11. What is a root class?
12. What is a superclass?
13. How is a superclass different from that of an abstract class?
14. What is a subclass?
15. Can a subclass serve as the definition for an abstract class?
16. What is a concrete class?
17. Why is a program unit advantageous in declaring a class?
18. What is an abstract method?
19. What is meant by the term *polymorphism*?
20. How is polymorphism practiced with THINK Pascal?
21. What is meant by inheritance in THINK Pascal?
22. When should the directive `override` be applied?
23. When is the reserved word **`inherited`** used?
24. What is the purpose of using the prefix `Self` when defining methods?
25. What is meant by the term *object reference* ?
26. How is an object reference declared? How is storage created for and disposed of for an object reference?
27. What is a handle? Why is it different from a simple pointer type?
28. When can one object reference be directly assigned the value of a second object reference?
29. When should type-casting be applied to an object reference?
30. When can the function `member` be applied?
31. Show the syntax for accessing an instance variable of an object.
32. What is the advantage of establishing `TObject` as a root class for all classes within a project?
33. Why is the method `Clone` of class `TObject` important in THINK Pascal?
34. Why is the method `Free` of class `TObject` important? How is `Free` different from the standard procedure `dispose`?
35. In declaring a class, how can the class being declared inherit from another class?

36. Can a class in THINK Pascal have multiple inheritance? Can you think of an example for testing your answer?
37. How can multiple inheritance be defined in THINK Pascal?
38. What is the purpose of using the Class Browser?
39. When should the Class Browser be used, and when should the LightsBug debugger be used?
40. How can an object of class A invoke a method from an object of class B when A and B are not related? This is sometimes referred to as having an object of one class *contract* with an object of some other class.
41. What is meant by the statement that *the class or object of a class encapsulates data and methods*?

PROGRAMMING EXERCISES

1. Define a program unit that will contain an object declaration for a class called `Sphere_Class`. This object declaration should inherit from `TObject` and contain two instance variables: `Sphere_Radius` for storing the radius of a sphere, and `Sphere_Center` for storing the center point of a sphere. This object declaration should also contain the following method declarations:

```
procedure Initialize_Sphere( Radius: real; Center: Point );
{ This method will initialize a sphere as an object by assigning }
{ instance variable Sphere_Radius the value of Radius and }
{ Sphere_Center the value of Center. }

function Compute_Circumference_Sphere: real;
{ This method computes and returns the circumference of a sphere }
{ as an object.}

function Compute_Surface_Area: real;
{ This method computes and returns the surface area of a sphere }
{ as an object.}

function Compute_Volume_Sphere: real;
{ This method computes and returns the volume of a sphere as an }
{ object.}

procedure Display_Spherical_Data;
{ This method displays to the screen the current radius, center, }
{ and other functional results of a sphere as an object.}
```

Write a program that borrows from this unit and that includes a loop for creating several `Sphere_Class` objects. The objects created within the loop can be referenced by an object-reference array; each element of the array of type `Sphere_Class`. Each array element represents a sphere object that can be created and assigned initial data, and that can provide information about itself in the Text window. If you want to be dramatic, use the Drawing window to display a sphere represented by a circle or by a quarter section of the sphere.

2. Define a program unit that declares a new class called Rectangle_Class. Allow this new class to have three instance variables: Rectangle_Length, Rectangle_Height, and Rectangle_Width. Methods should include initialization of instance variables where the object is a rectangle, computation of surface area, the computation of volume, and the display of information that relates to a rectangle.

3. For both Problems 1 and 2, define an abstract class that represents a general geometric object having the properties of volume and surface area. Use this class to define abstract methods for computing surface area and volume, and for displaying information. Redefine your concrete classes Sphere_Class and Rectangle_Class so that they directly inherit from the general geometric class. Have these two subclasses override the methods for computing surface area and volume, and for displaying information. Where it is appropriate within a method definition, inherit the use of a method from the ancestor class.

4. For the example discussed in Section 13.8, write a function called TypeOf that will determine the appropriate class type for several class types. Have this function return an enumerated value representing one of several classes that have been defined.

5. For the example discussed in Section 13.8, write a search procedure that will locate an appropriate object within the list of objects and that has as basic knowledge a key and the type of class associated with the object.

6. Building on Problem 5, define a procedure that can delete an object from within an object list. (Review the concepts for deleting a node within a linked-list of nodes).

7. For creating an object queue rather than an object stack, define a new Queue_Class that will represent a new class for a header object. Two instance variables are required: one for referencing the first object in a queue and a second referencing the last object. Methods should include pushing an object to the rear of an object queue, deleting from the front of an object queue, inspecting the head of an object list, and inspecting the rear of an object list.

8. For the example discussed in Section 13.8, define a new Object_Class having two instance variables: Succeeding_Object and Preceding_Object, both of type Object_Class. This class will be the basis for building a doubly referenced list of objects. Methods for this new object class include establishing a predecessor link, successor link, reference to successor, and reference to predecessor.

9. Develop a program that can write information encapsulated by each object within an object list to a file. Understand that each object within the object list may be of a different class.

10. Consider the object-class relationships shown in Figure 13.15.

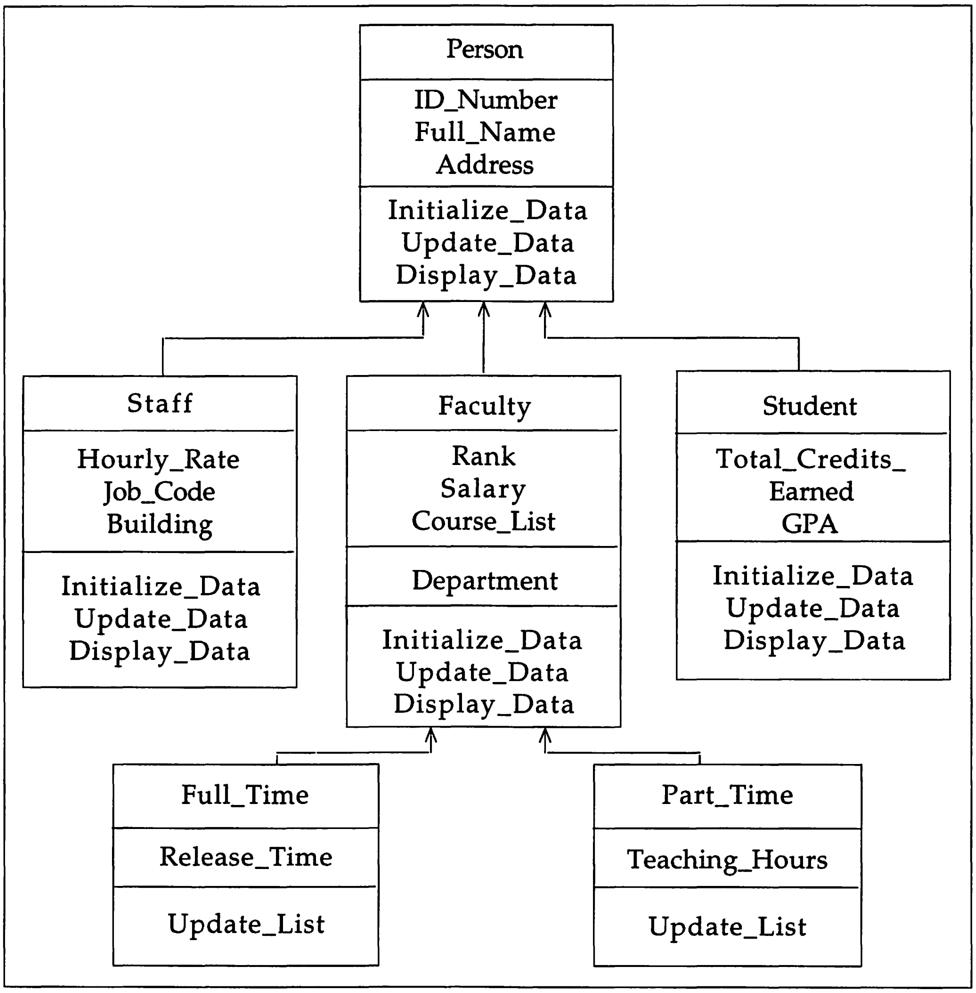


Figure 13.15 Class hierarchy for Person, Student, Faculty, and Staff.

For each class define a program unit representing the class and the methods. For Person, the method Initialize_Data initializes the ID_Number, Full_Name, and Address, while Display_Data displays the values of the instance variables. Update_Data allows data for Person to be modified. For class Student, Initialize_Data inherits the steps from the method Initialize_Data in class Person as well as initializing the data for Student. Display_Data displays all relevant data for Student as well as Person, while Update_Data allows changes in the information for both Student and Person. Class type Faculty stores Rank, Department, Course_List, and Salary and supports three methods: Initialize_ Data, Display_Data, and Update_Data. In turn, class Faculty has two subclasses: Part_Time and Full_Time. Class Part_Time contains part-time teaching hours, while class Full_Time contains hours for release time involving research. Both subclasses Full_Time and Part_Time support a method for updating a list of courses being taught and release time for research. For class `Staff`, the only data needed is `Hourly_Rate`, `Job_Code`, and `Building` where the staff member is employed. Class `Staff` will also define methods for `Initialize_Data`, `Display_Data`, and `Update_Data`.

Develop a menu-driven application that can create a new object as it is entered into the `Employment_System`. This system must be able to add a new employee, delete an old employee, search for information on a present employee, and use files to maintain information on all employees, presently employed as well as previously employed.

11. How could the class hierarchy in Problem 10 be changed to include student subclasses of `State`, `Foreign`, and `Out_of_State`. Data relevant to each class is the rate of tuition, and home address. For foreign students, a guardian and guardian address are also required.

Chapter 14

QuickDraw Library

OBJECTIVES

After completing Chapter 14, you will know the following:

1. The mathematical foundation of the QuickDraw Library.
2. The properties of the data type `grafPort` and how to use it as a separate entity.
3. The routines for defining and using a `grafPort`.
4. The routines for drawing points, lines, and rectangles.
5. The routines for drawing arcs and wedges.
6. The routines for drawing text.
7. The routines for defining and drawing regions and polygons.
8. The routines for defining and drawing pictures.
9. The concepts for understanding transfer bits and bit-transfer operations.
10. The concepts and routines for the mapping and scaling of points, rectangles, regions, and polygons.

14.1 BASIS OF THE QUICKDRAW LIBRARY

As explained earlier, the memory of the Macintosh computer is divided into two basic structures: RAM (random-access memory), for storing programs and data, and ROM (read-only memory), for storing special procedures and functions. Macintosh ROM is further divided into three special libraries: the Macintosh Operating System, the QuickDraw library, and the User Interface Toolbox. The Operating System handles low-level tasks such as managing RAM memory, input and output from disk drives, and serial communication ports such as the printer. The User Interface Toolbox allows interaction with higher level constructs, including windows and menus. The QuickDraw library allows graphics routines to be implemented from several programming levels. This library contains the definitions for various constants, types, variables, procedures, and functions needed to use the graphics capability of the Macintosh computer. The Macintosh Pascal language further subdivides the QuickDraw library into two parts: `QuickDraw1` and `QuickDraw2`. As you have seen in earlier chapters, using the `QuickDraw1` and `QuickDraw2` libraries simply requires adding these names to the **uses** clause. THINK Pascal does not require (or allow) the **uses** clause with QuickDraw, so the distinction between QuickDraw1 and QuickDraw2 is less important for the THINK Pascal programmer.

`QuickDraw1` contains all of the declarations needed to perform basic graphics and text operations as they relate to the Macintosh Pascal Drawing window. When all of the Macintosh windows are hidden by execution of the statement `HideAll`, data types and procedures from `QuickDraw2` can be used to frame and open grafPorts (complete drawing environments) of your own choosing.

The procedures `HideAll`, `ShowText`, `ShowDrawing`, `SetTextRect`, `SetDrawingRect`, `GetTextRect`, `WriteDraw`, `GetDrawingRect`, `SaveDrawing`, `DrawLine`, `DrawCircle`, and `InvertCircle`, discussed earlier, will not be emphasized in this chapter. The emphasis will be on using the QuickDraw routines, not the Macintosh/THINK Pascal window-manipulation procedures.

There are few important differences between THINK Pascal and Macintosh Pascal in the use of the QuickDraw libraries. The need for a **uses** clause in Macintosh Pascal is one difference. As indicated in Section 6.4, the `GetMouse` function is different in THINK Pascal and Macintosh Pascal. The programs in this chapter were written for THINK Pascal, but with minor modifications they will execute under Macintosh Pascal as well.

WARNING

The QuickDraw routines involving grafPorts, cursors, pointers, and handles can disrupt the normal operation of the Macintosh Pascal system, causing it to "bomb" if not properly used. Have a backup diskette for all your examples, and before executing the **GO** *option, save your program and apply the* **Check** *or* **Check Syntax** *option.*

14.2 MATHEMATICAL FOUNDATION OF THE QUICKDRAW LIBRARY

QuickDraw requires the use of four mathematical constructs throughout its data types, procedures, and functions. They are the point, the rectangle, the region, and the coordinate plane. Locating, placing, or moving information is done in terms of the coordinates of a plane, because all information given to QuickDraw routines is related to that of a coordinate plane. QuickDraw uses a finite, two-dimensional grid, as shown in Figure 14.1.

This coordinate plane contains 4,294,967,296 individual points. Each grid line is infinitely thin, and all grid coordinates are represented by integers. The coordinate plane is not infinite, because both horizontal and vertical coordinates must be specified within the range from −32767 to +32767. All coordinates are specified as integers, and all mathematical operations in QuickDraw are performed in integer arithmetic, producing exact results. Therefore there is no need to be concerned with rounding or truncation errors in QuickDraw computations.

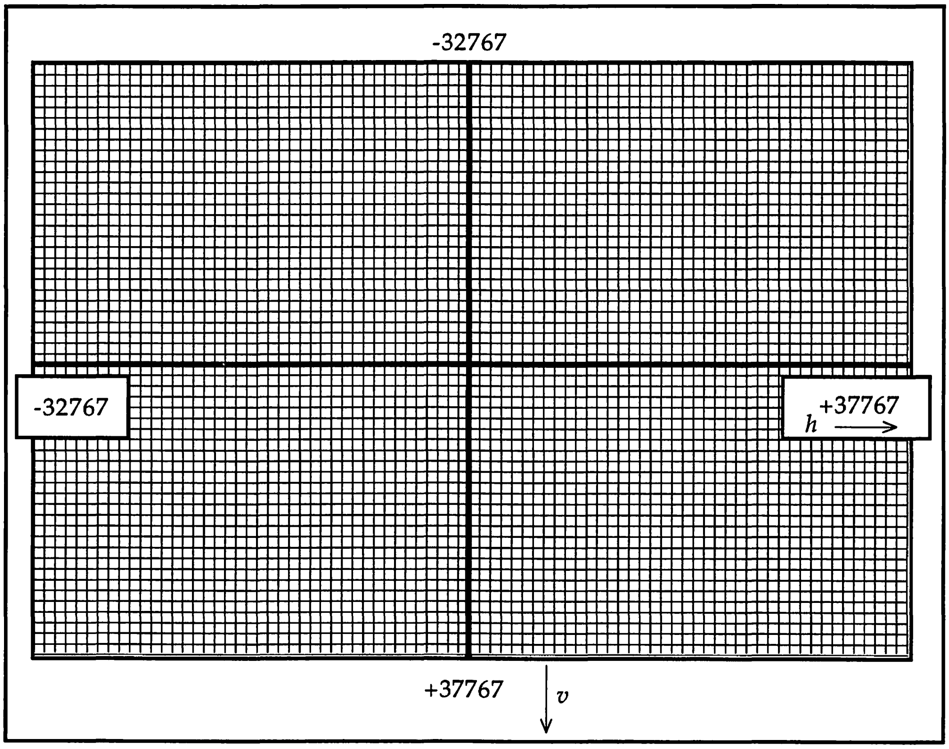


Figure 14.1 The two-dimensional grid used by `QuickDraw`.

The values of horizontal coordinates increase from left to right, and the values of vertical coordinates increase from top to bottom. This is in keeping with the process of scanning a monitor screen from top left to bottom right. The intersection of a horizontal grid line and a vertical grid line represents a point in the coordinate plane, and with respect to the Macintosh screen, the intersection of an infinitely thin horizontal and an infinitely thin vertical grid line represents the existence of a pixel.

On the coordinate plane a point is mathematically represented by an `integer` pair (*h*, *v*). In QuickDraw a point is defined in terms of a Pascal record containing two integers. The following variant record defines its structure:

```
type
  VHSelect =  (v, h);
  Point       =  record case integer of
                    0 :  ( v : integer;  h: integer );
                    1 :  (vh: array[VHSelect] of integer)
                 end;
```

For example, the THINK Pascal program `Display_Mouse_Point` uses the data type called `Point`:

```
program Display_Mouse_Point;
{ Purpose:  This program executes the routine GetMouse using }
{           the data structure Point, and displays the value of }
{           the data point for the mouse in the Text window. }
   var
      Mousepoint: Point;
begin
   ShowText;
   while not Button do { nothing }
      ;
   while Button do
      begin
         GetMouse(Mousepoint);
         writeln(Mousepoint.h, Mousepoint.v);
      end;
end.
```

This program stores the `(x, y)` pair in the structure called `Mousepoint`, and each pair of `h` and `v` values is printed in the Text window of the screen. The mouse point can be referred to as `Mousepoint.h` and `Mousepoint.v` or as `Mousepoint.vh[h]` and `Mousepoint.vh[v]`.

Mathematically, a rectangle in the coordinate plane is represented by an infinitely thin border having no direct representation on the screen. Rectangles can be used to define active drawing areas on the screen, can associate themselves with local coordinate systems, and can specify locations and sizes for QuickDraw routines. Figure 14.2 shows two points, one representing the top left and the second representing the bottom right, that define a rectangle.

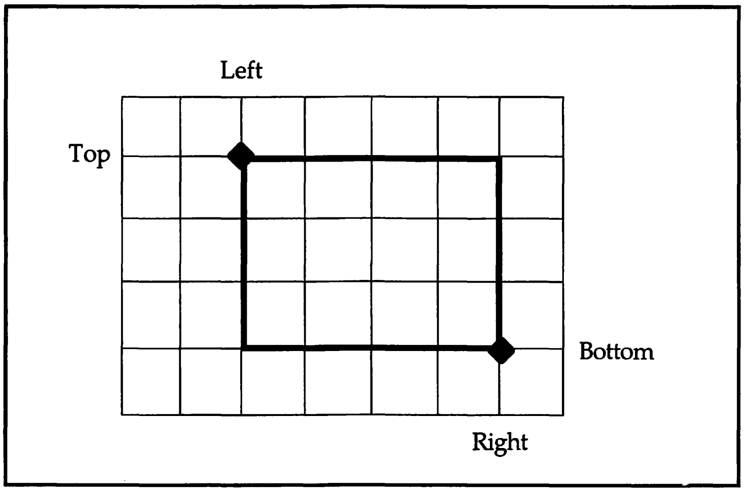


Figure 14.2 Two points in the plane define a rectangle.

Here is the QuickDraw definition for a rectangle in terms of a variant record:

```
type
   Rect   =   record case integer of
                    0 : ( top : integer; left : integer;
                        bottom : integer; right : integer);
                1 :  ( topLeft: Point;  botright : Point)
        end;
```

As an example of using the data type Rect, consider the following THINK Pascal program, titled Rectangles.

```
program Rectangles;
{ Purpose:  This program draws a rectangle in the Drawing. }
{           window. Press and hold the mouse button to test }
{           the program. Try moving the cursor in and out of }
{           the rectangle while holding down the button. }
   var
      Box: Rect;
      Mousepoint: Point;
begin
   ShowDrawing;
{ Set the size of the rectangle. }
   SetRect(Box, 100, 100, 300, 250);
{ Draw the boundary of the rectangle. }
   FrameRect(Box);
{ With the mouse button depressed, listen for short tones. }
   while not Button do {nothing }
      ;
   while Button do
      begin
         with Mousepoint do
            GetMouse(Mousepoint);
         { Check if the mouse point is within the boundaries of }
         { the rectangle Box. }
         if PtInRect(Mousepoint, Box) then
            SysBeep(2);
      end;
end.
```

Notice that a special Boolean function called PtInRect is used to test if Mousepoint lies within the rectangle called Box; this function is discussed in Section 14.4. If the point lies anywhere within the rectangle called Box, the function returns the Boolean value *true*; if the point lies outside, it returns the Boolean value *false*. A rectangle is considered empty when the bottom coordinate of the rectangle is equal to or less than the top coordinate or when the right coordinate is equal to or less than the left coordinate. Be careful not to form empty rectangles, because it is impossible to draw in them.

A *region* allows a more complex drawing area in QuickDraw. By definition, a region must be composed of one or more closed loops. A closed loop can be composed of oval and/or rectangular shapes as well as lines with curvatures that can be either concave or convex. A region can also be created through the union or intersection of several other regions, either disjointed or undisjointed. A region may contain one or more "holes." For example, the white area in Figure 14.3 represents a region generated by executing

QuickDraw routines. This region is formed from two other regions. `Region_1` is composed of the large outside oval, and `Region_2` is the small inside oval. The final step involves subtracting `Region_2` from `Region_1`, leaving the result as `Region_3`. In QuickDraw a region is represented by the following data type:

```
type
   Region = record
                rgnSize  : Integer;
                rgnBBox :  Rect;
                { additional data declarations if region is }
                { not rectangular }
            end;
```

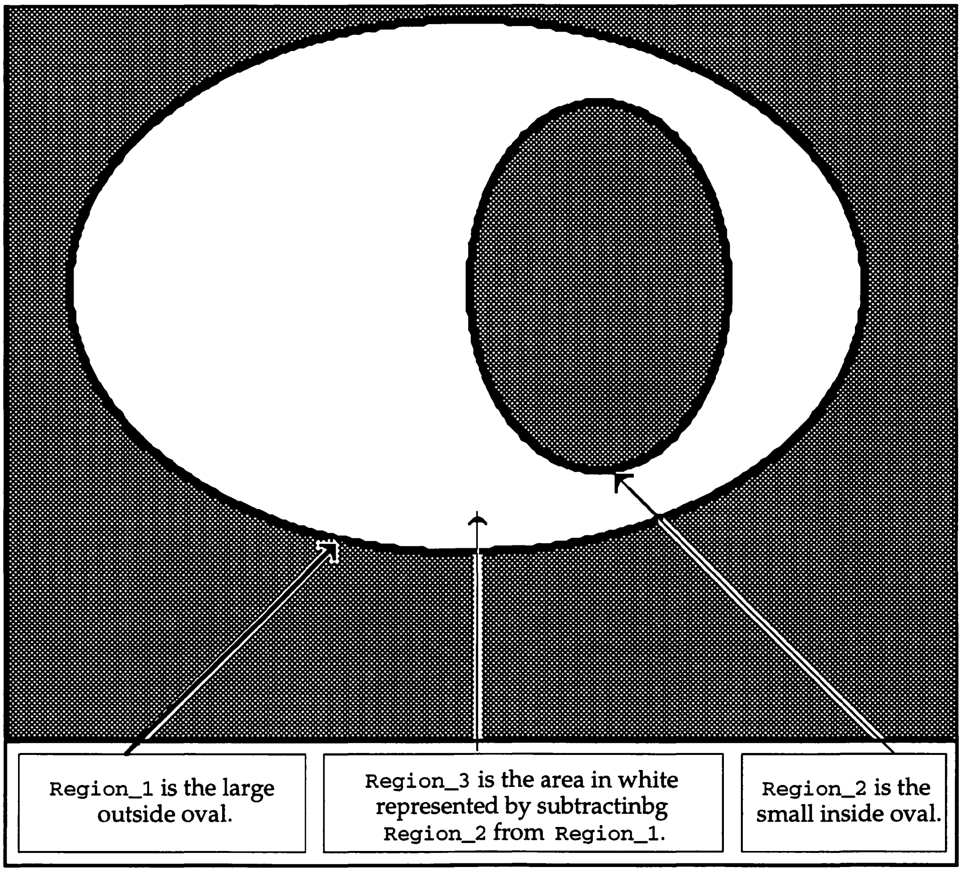

Region_1 is the large outside oval.
Region_3 is the area in white represented by subtractinbg Region_2 from Region_1.
Region_2 is the small inside oval.

Figure 14.3 Example of regions.

The field `rgnSize` specifies the amount of memory needed to store the region, and the field `rgnBBox` specifies the boundary of a rectangle that completely encloses the region.

The coordinate plane represents an abstract mathematical entity for a drawing; the screen represents reality in viewing a drawing. The screen is itself a matrix composed of

175,104 pixels enclosed within a region bounded by the points (0, 0) and (512, 342). Each pixel represents a single bit of information, a zero bit for a white pixel and a 1 bit for a black pixel. Any information displayed on the screen is stored as a bit image within a structure called a `BitMap`. The coordinate plane contains approximately 4.2 billion points, so it is impossible to display all of the coordinate plane on the screen at any one time.

The importance of the QuickDraw Library is its ability to establish distinct ports for displaying information on the screen. A port (`grafPort`) represents a complete drawing environment composed of its own coordinate system, drawing location, character and font set, background and foreground colors, patterns, and viewing location. Each drawing environment is expressed in terms of its own local coordinate system, rather than having one large coordinate plane. It can act as a local region for drawing, or it can be mapped into another region for viewing. In QuickDraw a port is represented by the following structure:

```
type
   GrafPtr = ^GrafPort;
   GrafPort = record
                     device :       integer;
                     portBits:      BitMap;
                     portRect:      Rect;
                     visRgn:        RgnHandle;
                     clipRgn:       RgnHandle;
                     bkPat:         Pattern;
                     fillPat:       Pattern;
                     pnLoc:         Point;
                     pnSize:        Point;
                     pnMode:        integer;
                     pnPat:         Pattern;
                     pnVis:         integer;
                     txFont:        integer;
                     txFace:        Style;
                     txMode:        integer;
                     txSize:        integer;
                     spExtra:       longint;
                     fgColor:       longint;
                     bkColor:       longint;
                     colrBit:       integer;
                     patStretch:    integer;
                     picSave:       QDHandle;
                     rgnSave:       QDHandle;
                     polySave:      QDHandle;
                     grafProcs:     QDProcsPtr
                  end;
```

All QuickDraw routines refer to grafPorts by way of the pointer `GrafPtr`. `GrafPorts` should be used dynamically; therefore, it is best to represent a `grafPort` as a dynamic data structure rather than a static structure. This requires using the procedure new before opening a `grafPort`.

The device field indicates the physical output device for displaying the information of a grafPort. This is necessary because there are physical differences for the same fonts for different output devices. The device number for the Macintosh screen is 0. The field portBits defines the memory area for the bit image as well as the coordinate plane associated with the image on the screen. Keep in mind that the screen is represented in terms of a global coordinate system, whereas the coordinates for points in a grafPort are expressed in terms of a local coordinate system. Each grafPort supports its own local coordinate system. The QuickDraw library converts the information stored by the grafPort into the global coordinates of the screen. The subfield portBits.bounds defines a rectangle specifying the limits of the screen. The top left and bottom right points of this rectangle are expressed in terms of the local coordinates of the grafPort. The third field, portRect, defines a rectangular region for drawing to a grafPort that has coordinates in the system defined by the rectangle portBits.bounds. Initially, portBits.bounds is represented by screen boundary points: (0, 0) and (512, 342). Drawing to any port occurs within the rectangle defined by portRect. This field specifies the active rectangular drawing area for the grafPort's coordinate system. In short, it specifies the rectangular area of the screen where a graphic can be displayed.

The field visRgn, used by the Toolbox and not to be modified by the user's program, defines the region that is visible on the screen. For example, if there is a window that overlaps and suppresses part of the view of a second window, the visRgn field of this second window will contain only that region that is visible and not suppressed by the overlaid window. The field clipRgn defines the region where the drawing is suppressed (clipped) and can be established or changed by using one of several QuickDraw routines. For example, the Macintosh Pascal statements

```
SetRect( Box_1, 50, 100, 250, 300);
ClipRect( Box_1 );
```

limit the visible region to within the rectangle called Box_1. Everything drawn outside the boundaries of Box_1 is clipped (not shown). Initially the visRgn is a handle to the rectangular region given by the points (0, 0) and (512, 342), and the clipRgn is a handle to the region given by the points (–32767, –32767) and (32767, 32767). Drawing always occurs within the grafPort's coordinate system, always within the intersection of the rectangles of grafPort's portBits.bounds and portRect, and always within the clipping boundaries defined by the intersection of the regions visRgn and clipRgn.

The fields bkPat (default value is *white*) and fillPat (default value is *black*) represent the background and fill patterns, respectively. These can be altered through special QuickDraw routines. In addition, the field pnLoc contains the current pen location [default value is (0, 0)]; pnSize contains the current pen size [default value (1,1)]; pnMode, the current pen mode (default value is patCopy); pnPat, the current pen pattern (default value is *black*); and pnVis indicates if the pen is currently visible (0 if visible, negative if not visible; default value is 0). The field txFont contains the current text font (0 is the system font, default value is 0); txFace, the current text style (default value is *normal*); txMode, the current text mode (default value is *srcOr*); and txSize, the current text size (default value is 0). The field spExtra contains information on extra space necessary for justifying the text (default value is 0). The field picSave provides a handle to memory for storing picture information, rgnSave provides a handle to memory for storing the current region definition, and polySave provides a handle to the current polygon definition. The fields fgColor, bkColor, and colrBit contain values related to color drawing, and the field patStretch is used by the printer software. The last field, `grafProcs`, is a handle to memory containing the definitions of customized QuickDraw routines.

14.3 DEFINING A PORT USING GRAFPORT ROUTINES

Figure 14.4 lists the procedures used to define and manipulate grafPorts. Each routine is listed by its header, including the procedure name and one or more parameters, with a brief explanation of its actions.

procedure `OpenPort( Port : grafPort );` This procedure opens a `grafPort`, allocating memory, initializing fields with default values, and making `Port` the current active `grafPort`. Execute `new(Port)` before executing `OpenPort(Port)`.

procedure `InitPort( Port : GraftPort );` For a `grafPort` that has already been opened, this procedure reinitializes the `grafPort` data structure, making it the current port.

procedure `ClosePort( Port : grafPort );` This procedure deallocates the memory space occupied by the handles `visRgn` and `clipRgn`. Execute this procedure when the `grafPort` is no longer needed.

procedure `SetPort( Port : grafPort );` This sets the port to be the current `grafPort`.

procedure `GetPort(` **var** `Port : grafPort );` For the corresponding actual parameter, this procedure returns a pointer to the current port.

procedure `PortSize( Width, Height : integer );` This procedure changes the size of the current `grafPort's` field `portRect`. It does not affect the screen, but changes the size of the active window of the current `grafPort`. The top left corner remains unchanged, while the bottom right corner is adjusted to the dimensions given by `Width` and `Height`. This procedure has no effect on `clipRgn` or `visRgn`, nor does it affect the local coordinate system of the current `grafPort`.

procedure `MoveToPort(Global_Left , Global_Right : integer );` This procedure moves the position of the current `Grafport's portRect`. Though it does not affect the screen, it changes the position for drawing inside the `grafPort`. No changes are made to `clipRgn` and `visRgn`, nor are local coordinates affected.

procedure `SetOrigin( H, V : integer );` This procedure changes the coordinate system of the current `grafPort`. Though it does not affect the screen, it updates the fields `portBits`, `portRect`, and `visRgn`. This procedure does not update the `clipRgn` of the current `grafPort`.

procedure `SetClip( Regn : RgnHandle );` This procedure changes the `clipRgn` of the current `grafPort` by making a copy of the region given by `Regn`.

procedure `GetClip( Regn : RgnHandle );` This procedure makes a copy of the `clipRgn` of the current `grafPort` and assigns it to the region given by `Regn`.

procedure `ClipRect( Box : Rect );` This procedure changes the `clipRgn` of the current `grafPort` to the rectangle specified by `Box`.

procedure `BackPat( Pat : Pattern );` This procedure sets the background pattern of the current `grafPort` to the value given by `Pat`.

Figure 14.4 Procedures used to define and manipulate a `grafPort`.

The following program, `Drawing_Windows`, applies some of these routines.

```
program Drawing_Windows(input, output);
{ Purpose:  This program shows the basic concept of drawing }
{           grafPorts. All windows and drawings are in terms of }
{           the global coordinates of the screen. }
   type
      Port = GrafPtr;
      Counter = integer;
   var
      Window : array[1..4] of Port;
      Rectangle : array[1..4] of Rect;
      J : Counter;
{ ********************************************************** }
   procedure Open_Window (var Viewport : Port);
   begin
   { Establish dynamic structure for window. }
      new(Viewport);
   { Open the grafPort for drawing. }
      OpenPort(Viewport);
   end;
{ ********************************************************** }
   procedure Initialize_grafPort (Viewport : Port;  Box : Rect);
   begin
   { Set window as current grafPort. }
      SetPort(Viewport);
   { Establish clipping region for grafPort. }
      ClipRect(Box);
   end;
{ ********************************************************** }
```

```
   procedure Draw_in_Window (Viewport : Port;  Box : Rect;  Jth
                                    : integer);
   begin
   { Set window as current grafPort. }
      SetPort(Viewport);
   { Fill box with white background. }
      FillRect(Box, white);
   { Set pen width for drawing border. }
      PenSize(3, 3);
   { Draw a border for viewing the window. }
      FrameRect(Box);
   { Return pen size to normal width and height. }
      PenSize(1, 1);
   { Adjust text face before drawing string. }
      TextFace([condense, outline]);
   { Move pen to a new location in the window. }
      with Box do
         MoveTo( left + (right - left) div 4, top + (bottom -
                        top) div 2);
   { Draw the message 'Window Number ... ' within each window. }
      DrawString(concat('Window Number ', chr(48 + Jth)));
   end;
{ ********************************************************** }
   procedure Dispose_of_Window (var Viewport : Port);
   begin
      ClosePort(Viewport);
      dispose(Viewport);
   end;
{ ********************************************************** }
   procedure Pushbutton;
      var
         Time: longint;
   begin
   { Wait for mouse button to be pressed. }
      while not Button do { nothing }
         ;
      Delay(10, Time);
   end;
{ ********************************************************** }
begin { Body of main program. }
{ Hide all Macintosh Pascal windows. }
   HideAll;
{ Establish data structures for each window and open them. }
   for J := 1 to 4 do
      Open_Window(Window[J]);
{ Establish the boundaries of four window areas. }
   SetRect(Rectangle[1], 0, 0, 256, 171);
   SetRect(Rectangle[2], 0, 172, 256, 342);
   SetRect(Rectangle[3], 257, 0, 512, 171);
   SetRect(Rectangle[4], 257, 172, 512, 342);
{ Initialize the clipping region of each window. }
```

```
   for J := 1 to 4 do
      Initialize_grafPort(Window[J], Rectangle[J]);
{ Open each of the four windows. }
   for J := 1 to 4 do
      Draw_in_Window(Window[J], Rectangle[J], J);
{ Study the screen. }
   Pushbutton;
{ Dispose of storage for all four windows. }
   for J := 1 to 4 do
      Dispose_of_Window(Window[J]);
end.
```

This program divides the screen into four separate windows, as shown in Figure 14.5.

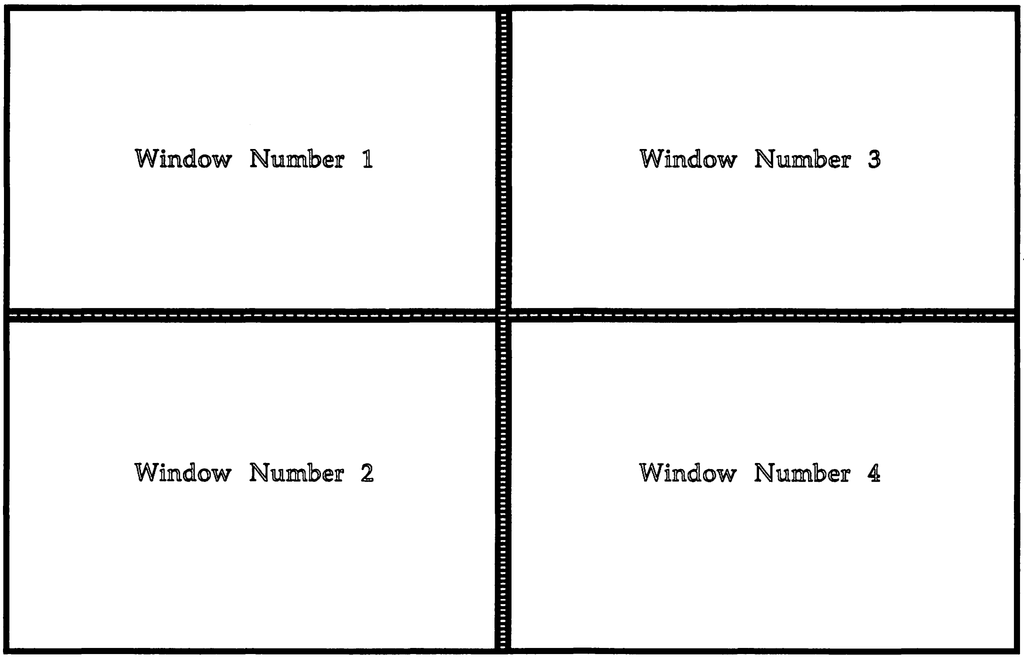


Figure 14.5 Four separate windows created by the program `Drawing_Windows`.

The main program has six major steps:

1. Hide all of the Macintosh Pascal windows.
2. Allocate the data structures and initialize the grafPorts of each window.
3. Establish the drawing boundary for each window.
4. Initialize the clipping region for each window.
5. Draw a simple message to each window.
6. Dispose of storage for each window before ending execution.

Step 2 is performed by calling on the procedure Open_Window. In this procedure the library routine new establishes the dynamic structure. The QuickDraw routine OpenPort is then executed, initializing the window's grafPort structure. Notice that the formal parameter Viewport is of type variable. Failure to declare this parameter as variable causes the program to crash. The last grafPort to become the current window is Window[4]. Step 3 uses the procedure SetRect to establish the boundaries for each of the four separate rectangles. Step 4 calls on the procedure Initialize_grafPort. This procedure sets the current grafPort by executing SetPort. The routine ClipRect sets the current grafPort's clipping region to the value given by Box. Without the statement calling on ClipRect, the clipping region is represented by its default values. Step 5 is calls the procedure Draw_in_Window. If the procedure SetPort did not exist, Window[4] would always be the current grafPort, and only this window would appear on the screen.

The routines FillRect, FrameRect, and DrawString draw to the window given by Viewport. The procedure DrawString displays the window message as a single string concatenated from the substring Window_Number and the character returned by executing the chr function. In this program, the local coordinates of each grafPort coincide with the global coordinates of the screen. The last step calls on the procedure Dispose_of_Window for disposing of a grafPort. This procedure in turn calls on the function ClosePort followed by dispose. It is important to *close a grafPort before disposing of it.* If you fail to do this, portions of memory used by the handles visRgn and clipRgn cannot be recovered by Pascal's Memory Manager.

A second approach to the problem is shown in the following THINK Pascal program, titled Drawing_Windows_Revised:

```
program Drawing_Windows_Revised(input, output);
{ Purpose:  This program employs the routine SetOrigin to shift }
{           the local coordinate of the grafPort while leaving }
{           the global coordinates of the screen unchanged. }
  type
     Port = GrafPtr;
     Counter = integer;
  var
     Origin : array[1..4] of Point;
     Window : array[1..4] of Port;
     J : Counter;
 { ********************************************************** }
  procedure Open_Window (var Viewport : Port);
     begin
        new(Viewport);
        OpenPort(Viewport);
     end;
 { ********************************************************** }
  procedure Draw_in_Window (Viewport : Port;  Topleft :
                                      Point; Jth : integer);
     var
        Rectangle : Rect;
  begin
     SetPort(Viewport);
     SetOrigin(-Topleft.h, -Topleft.v);
     SetRect(Rectangle, 0, 0, 256, 171);
```

```
      ClipRect(Rectangle);
      FillRect(Rectangle, white);
      PenSize(3, 3);
      FrameRect(Rectangle);
      PenSize(1, 1);
      TextFace([condense, outline]);
      with Rectangle do
         MoveTo(left + (right - left) div 4, top + (bottom - top)
                          div 2);
      DrawString(concat('Window Number ', chr(48 + Jth)));
   end;
 { ******************************************************** }
   procedure Dispose_of_Window (var Viewport : Port);
      begin
         ClosePort(Viewport);
         dispose(Viewport);
      end;
 { ******************************************************** }
   procedure Pushbutton;
      var
         Time: longint;
   begin
   { Wait for mouse button to be pressed. }
      while not Button do { nothing }
         ;
      Delay(10, Time);
   end;
{ ******************************************************** }
begin { Body of main program. }
   HideAll;
   for J := 1 to 4 do
      Open_Window(Window[J]);
   SetPt(Origin[1], 0, 0);
   SetPt(Origin[2], 0, 171);
   SetPt(Origin[3], 256, 0);
   SetPt(Origin[4], 256, 171);
   for J := 1 to 4 do
      Draw_in_Window(Window[J], Origin[J], J);
   Pushbutton;
   for J := 1 to 4 do
      Dispose_of_Window(Window[J]);
end.
```

The procedure `Initialize_grafPort` has been replaced by `SetOrigin` in the procedure `Draw_in_Window`. The main body of the program assigns four origins for four top left points. The procedure `Draw_in_Window` thus appears to be drawing to a general window, given in terms of the local coordinates specified by the variable called `Rectangle`, which is assigned top left coordinates of (0, 0), and bottom right coordinates of (256, 171). The clipping region is specified by the boundaries of `Rectangle`. Each time that procedure `Draw_in_Window` is executed, the routine `SetOrigin` changes the local coordinate system of the current `grafPort` by assigning a new value to the top left corner of portRect. It does not affect the screen at this point, but it does update portBits.bounds, portRect, and visRgn, while leaving the fields clipRgn and pnLoc unchanged. Subsequently, all drawing and computations related to drawing are performed in terms of the new coordinate system. Notice that SetOrigin *treats all of its arguments as negated values.*

We can use the command MovePortTo(Topleft.h, Topleft.v) in place of SetOrigin to achieve the same actions for drawing to the screen. MovePortTo is different from SetOrigin; it moves the active drawing area on the screen to the coordinates specified by Topleft.h, and Topleft.v. This pair of arguments now specifies new global coordinates for drawing, leaving the local coordinates of the grafPort unchanged. Drawing and computations for the procedure Draw_in_Window still appear as if they were local to the coordinates given by Rectangle.

14.4 DRAWING WITH POINTS, LINES, AND RECTANGLES

Figures 14.6 through 14.8 summarize the numerous procedures and functions for using points as well as drawing lines, rectangles, ovals, and rounded rectangles. Figure 14.6 lists the QuickDraw routines related to drawing lines and setting the drawing pen.

procedure HidePen; This procedure decrements the current grafPort's pnVis field. Initially zero when **OpenPort** is executed, the pen remains hidden and does not draw whenever pnVis is negative. Calls to **HidePen** should always be balanced with an equal number of calls to **ShowPen.**

procedure ShowPen; This procedure increments the current grafPort's pnVis field. When the value of pnVis becomes zero, the pen resumes drawing to the screen. Extra calls on this procedure can cause the value of pnVis to be incremented beyond zero. Calls to **ShowPen** should always balance calls to **HidePen**.

procedure GetPen(**var** Pen_Location : Point); This procedure returns the location of the pen in terms of the current grafPort's coordinate system.

procedure GetPenState(**var** Pen_State : PenState); This procedure assigns the present state of the pen (location, size, pattern, mode) to the parameter Pen_State.

procedure SetPenState(Pen_State : PenState); This procedure sets the pen state of of the current grafPort to the value stored in the parameter Pen_State.

procedure PenSize(Width, Height : integer); This procedure sets the pen's drawing width and drawing height of the current grafPort.

procedure `PenMode( Pen_Mode : integer );` This procedure establishes the transfer mode through which the pen pattern is transferred to the bit map when drawing lines, rectangles, ovals, or shapes.

procedure `PenPat( Pen_Pat : Pattern );` This procedure assigns the pattern given by the value of `Pen_Pat` to the pen of the current `grafPort`. Standard pattern values are `white`, `black`, `gray`, `ltgray`, and `dkgray`. The initial value for the field `pnPat` is `black`.

procedure `PenNormal;` This procedure reinitializes the pen state (pen size, pen mode, and pen pattern) for the current `grafPort`. The field `pnSize` becomes (1, 1), `pnMode` takes the value `patCopy`, and `pnPat` takes the value `black`.

procedure `MoveTo( Horizontal, Vertical : integer );` This procedure changes the location of the pen by moving it to the point `(Horizontal,Vertical)` in the coordinates of the current `grafPort`. This procedure leaves the screen unaffected.

procedure `Move( Delta_h, Delta_v : integer);` This procedure moves the location of the pen through a horizontal distance `Delta_h` and vertical distance `Delta_v`. A positive value in `Delta_h` moves the pen to the right; a positive value in `Delta_v` moves the pen downward.

procedure `LineTo( Horizontal, Vertical : integer );` This procedure draws a direct line from the current pen location to the point `(Horizontal,Vertical)`. The new location for the pen is the point `(Horizontal,Vertical)`.

procedure `Line( Delta_h, Delta_v : integer );` This procedure draws a direct line from the current pen location through a horizontal distance `Delta_h` and vertical distance `Delta_v`. The new pen location becomes the coordinates at the end of the drawn line.

Figure 14.6 QuickDraw procedures for drawing lines and setting the drawing pen.

The pen acts as a drawing tool on the screen, having four characteristics: location, size, mode for drawing, and pattern for drawing. The pen location is represented by a point in the grafPort's coordinate system. There is no restriction on the pen location. It can be set by either executing the commands `MoveTo` or `Move`. The pen is considered rectangular in shape, and its size (width and height) can be adjusted by executing `PenSize`. For widely spaced lines, the value of the pen pattern affects the type of pattern (`white, black, gray, ltgray,` or `dkgray`) for filling between the lines of boundaries. This is useful for drawing thick boundaries around the edges of a window. `PenMode` establishes a mode through which the pen pattern is transferred to individual bits in the bit map for drawing lines or shapes. PenMode can be set to one of eight possible transfer modes.

Pen settings are local to a grafPort and are not affected by pen settings in other grafPorts. GetPenState allows the temporary storing of pen location, size, pattern, and mode using a variable of type PenState. This is used when execution of a procedure may temporarily change the current pen state. SetPenState allows the pen state to be restored from a temporary storage variable.

HidePen and ShowPen have a direct effect on the value of the current grafPort's pnVis field. HidePen decrements and ShowPen increments the value of the current grafPort's pnVis field. A value for pnVis greater than or equal to zero indicates that the pen is visible; a negative value hides the pen, keeping it from drawing on the screen. This is useful when we make nested calls for hiding and showing the pen. PenNormal resets the fields pnSize, pnMode, and pnPat to the pen's initial state.

For drawing a point, we use one of two methods:

```
MoveTo( x, y );
Line( dx, dy );
```

or

```
MoveTo( x, y );
LineTo( x + dx, y + dy);
```

where dx and dy are both zero. If both dx and dy are either +1 or −1, these sets of commands result in a line being drawn between two points.

Figure 14.7 lists various routines for calculations with rectangles.

procedure SetRect(**var** Box : Rect; left, top, right, bottom : integer); This procedure assigns the parameter Box with the left top and right bottom points, establishing the boundary coordinates of a rectangle.

function SectRect(SrcRect_A, SrcRect_B : Rect; **var** DistRect : Rect): Boolean; This function performs two operations. First, it computes the rectangle resulting from the intersection of the rectangles SrcRect_A and SrcRect_B, assigning it to the rectangle DistRect. Second, it returns a Boolean value *true* if the intersection is a nonempty rectangle and *false* if the intersection is an empty rectangle. Areas or rectangles are defined as those within the boundaries of the coordinates of the rectangles. Rectangles that touch at a point or line do not intersect. An empty rectangle is noted by the coordinates (0, 0) and (0, 0). One of the source rectangles can also be specified as a destination rectangle.

procedure UnionRect(SrcRect_A, SrcRect_B : Rect; **var** DistRect : Rect); This procedure computes the smallest rectangle enclosing the union of the rectangles SrcRect_A and SrcRect_B and assigns this to DistRect. One of the source rectangles can also be specified as a destination rectangle.

function PtInRect(Coordinate_Pt: Point; Box: Rect): Boolean; This function checks if a pixel below and to the right of Coordinate_Pt is enclosed within the rectangle specified by Box, returning either a Boolean value *true* if the point is inside or *false* if it lies outside.

procedure Pt2Rect(Point_A, Point_B : Point; **var** DistRect : Rect); This procedure returns the smallest rectangle enclosed by the two points Point_A and Point_B.

function EqualRect(Rect_A , Rect_B : Rect) : Boolean; This function returns the Boolean value *true* if both rectangles Rect_A and Rect_B have identical boundary coordinates. If not, it returns the Boolean value *false.*

function EmptyRect(Box : Rect) : Boolean; This function returns the Boolean value *true* if the parameter Box is an empty rectangle, and *false* if the parameter Box is not an empty rectangle.

Figure 14.7 QuickDraw procedures and functions for using rectangles.

Calculations involving rectangles are independent of the current grafPort coordinate system and will execute the same whether or not a grafPort is active. The program Drawing_Windows_Revised shows how rectangles are applied in terms of a common coordinate system, using SetOrigin to perform computations and draw to a new coordinate system.

Figure 14.8 lists various routines for performing graphics operations on rectangles, rectangles with rounded corners, and ovals. Ovals are always drawn within a specified rectangular region. If the rectangle is square, the oval drawn is a circle.

procedure FrameRect(Box : Rect); This procedure draws the outline of a rectangle within the region bounded by the rectangle Box, using the pen state (pattern, mode, and size) of the current grafPort. Pen location remains unchanged.

procedure PaintRect(Box : Rect); This procedure paints the rectangle Box, with the current pen pattern given by pnPat. Pen location remains unchanged.

procedure EraseRect(Box : Rect); This procedure paints the rectangle Box, with the current background pattern given by bkPat. The fields pnPat and pnMode are ignored, and the pen location remains unchanged.

procedure InvertRect(Box : Rect); This procedure inverts the pixels within the rectangle Box. The fields pnPat, pnMode, and bkPat are ignored, and the pen location remains unchanged.

procedure FillRect(Box : Rect; Pen_Pat : Pattern); This procedure fills the rectangle Box, with the pattern specified by the value Pen_Pat. The fields pnPat, pnMode, and bkPat are ignored, and the pen location remains unchanged.

procedure FrameOval(Box : Rect); This procedure draws an oval within the boundary specified by the rectangle Box, using the pen state (pattern, mode, and size) of the current grafPort. Pen location remains unchanged.

procedure PaintOval(Box : Rect); This procedure paints an oval within the boundary specified by the rectangle Box using the current pen pattern given by pnPat Pen location remains unchanged.

procedure EraseOval(Box : Rect); This procedure fills an oval with the current background pattern within the boundary specified by the rectangle Box. The fields pnPat and pnMode are ignored, and the pen location remains unchanged.

procedure InvertOval(Box : Rect); This procedure inverts the pixels within the boundary of an oval, with the oval bounded by the rectangle Box. The fields pnPat, pnMode, and bkPat are ignored, and the pen location remains unchanged.

procedure FillOval(Box : Rect; Pen_Pat : Pattern); This procedure fills an oval with the pattern given by the value of Pen_Pat. This oval is bounded by the rectangle Box. The fields pnPat, pnMode, and bkPat are ignored, and the pen location remains unchanged.

procedure FrameRoundRect(Box : Rect; Oval_Width, Oval_Height : integer); This procedure draws the outline of a rectangle with rounded corners within the region bounded by the rectangle Box, using the current pen state. The parameters Oval_Width and Oval_Height specify corner diameters with respect to width and height. Pen location remains unchanged.

procedure PaintRoundRect(Box : Rect; Oval_Width, Oval_Height : integer); This procedure paints a rectangle with rounded corners using the current pen pattern and pen mode. The rectangle `Box` specifies the region bounding the rounded rectangle, and the parameters `Oval_Width` and `Oval_Height` specify the corner diameters with respect to width and height. Pen location remains unchanged.

procedure `EraseRoundRect( Box : Rect; Oval_Width, Oval_Height : integer );` This procedure fills a rounded rectangle with the current background color. The rectangle `Box` specifies the region bounding the rounded rectangle, and the parameters `Oval_Width` and `Oval_Height` specify the corner diameters with respect to width and height. The fields `pnPat`, `pnMode`, and `bkPat` are ignored, and the pen location remains unchanged.

procedure `InvertRoundRect( Box : Rect; Oval_Width, Oval_Height : integer );` This procedure inverts the pixels within the boundaries of the rounded rectangle. The rectangle `Box` specifies the region bounding the rounded rectangle, and the parameters `Oval_Width` and `Oval_Height` specify the corner diameters with respect to width and height. The fields `pnPat`, `pnMode`, and `bkPat` are ignored, and the pen location remains unchanged.

procedure `FillRoundRect( Box : Rect; Oval_Width, Oval_Height : integer; Pen_Pat : Pattern );` This procedure fills the rounded rectangle with a pattern given by `Pen_Pat`. The rectangle `Box` specifies the region bounding the rounded rectangle, and the parameters `Oval_Width` and `Oval_Height` specify the corner diameters with respect to width and height. The fields `pnPat`, `pnMode`, and `bkPat` are ignored, and the pen location remains unchanged.

Figure 14.8 QuickDraw procedures for performing graphics operations on rectangles.

The three procedures `DrawLine`, `PaintCircle`, and `InvertCircle` are not discussed here because they are a part of the Macintosh Pascal `QuickDraw1` library and are not part of the ROM-based QuickDraw library. Refer to Chapter 6 for a review of these procedures.

14.5 DRAWING WITH ARCS AND WEDGES

QuickDraw's routines for drawing arcs and wedge-shaped ovals provide added opportunities for displaying objects on the screen. They can be used to draw pie charts in various elliptical shapes, create regions that are nonrectangular, and provide a foundation for a

CAD (computer-aided design) library. Figure 14.9 lists the routines available for operating with arcs and wedges.

procedure `FrameArc( Box : Rect; Starting_Angle, Arc_Angle : integer );` This procedure draws an arc along an oval, bounded by the rectangle `Box`, beginning at `Starting_Angle` and extending to `Starting_Angle` + `Arc_Angle`, using the pen state of the current `grafPort`. The path of the arc is as wide and high as specified by the field `pnPat`. All angular arguments are assumed in degrees, with positive angles rotated clockwise, negative angles rotated counterclockwise. Zero degrees is positioned at 12 o'clock. Pen location remains unchanged.

procedure `PaintArc( Box : Rect; Starting_Angle, Arc_Angle : integer );` This procedure paints a wedge within an oval, bounded by the rectangle `Box`, beginning at `Starting_Angle` and extending to `Starting_Angle` + `Arc_Angle`, using the pen state of the current `grafPort`. Pen location remains unchanged.

procedure `EraseArc( Box : Rect; Starting_Angle, Arc_Angle : integer );` This procedure paints a wedge within an oval, bounded by the rectangle `Box`, beginning at `Starting_Angle` and extending to `Starting_Angle` + `Arc_Angle`, using the background pattern of the current pen state. The fields `pnPat` and `pnMode` are ignored, and the pen location remains unchanged.

procedure `InvertArc( Box : Rect; Starting_Angle, Arc_Angle : integer );` This procedure inverts the pixels enclosed by a wedge within an oval, bounded by the rectangle `Box`, beginning at `Starting_Angle` and extending to `Starting_Angle` + `Arc_Angle`. The fields `pnPat` , `bkPat`, and `pnMode` are ignored, and the pen location remains unchanged.

procedure `FillArc( Box : Rect; Starting_Angle, Arc_Angle : integer );` This procedure fills a wedge within an oval, bounded by the rectangle `Box`, beginning at `Starting_Angle` and extending to `Starting_Angle` + `Arc_Angle`, using the current pattern of pen state. The fields `pnPat` , `bkPat`, and `pnMode` are ignored, and the pen location remains unchanged.

procedure `PtToAngle(Box : Rect; Given_Point : Point;` **var** `Angle : integer );` This procedure computes the `integer` angle from `Given_Point` to an imaginary vertical line intersecting the center of the rectangle `Box`. Positive angles are measured clockwise, with zero degrees at 12 o'clock.

Figure 14.9 QuickDraw procedures for drawing arcs and wedges.

The frame of reference with respect to angles differs from that of the coordinate plane described in the first section of this chapter. Figures 14.10a and 14.10b show this difference. With arcs and wedge shapes, angles have either positive or negative values. Positive angles rotate clockwise; negative angles rotate counterclockwise. Figure 14.10a shows the method for specifying angles in relation to the screen; 0 degrees is positioned at 12 o'clock, 90 degrees (–270 degrees) at 3 o'clock, 180 degrees (–180 degrees) at 6 o'clock, and 270 degrees (–90 degrees) at 9 o'clock.

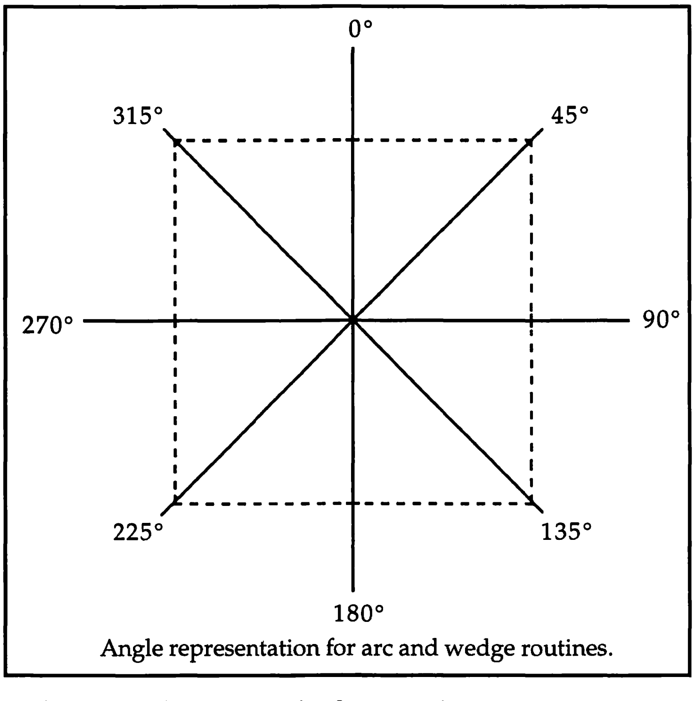


Figure 14.10a Frame of reference with respect to angles.

This is different from the coordinate plane shown in Figure 14.10b. Here angles in the coordinate plane lead those of arc and wedge angles by 90 degrees. This is important when using trigonometric functions such as sine and cosine, since they assume angles that correspond to the coordinate plane, not the screen. Also keep in mind that the angular arguments for arc and wedge routines are specified in degrees, whereas the arguments for Pascal sine and cosine functions are given in radians. In addition, the angle through which an arc or wedge is drawn is relative to the corner of the bounding rectangle, whether the rectangle is square or not. For example, a wedge with a starting angle of 0 degrees and an arc angle of 45 degrees has as one side a vertical line intersecting the center of the bounding rectangle. A second side is a line intersecting the center and directed toward the upper right corner of the bounding rectangle.

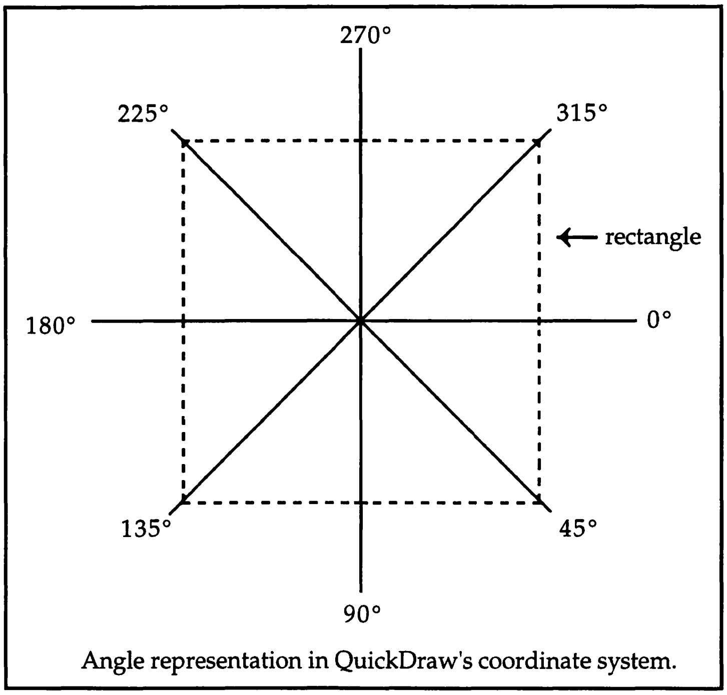


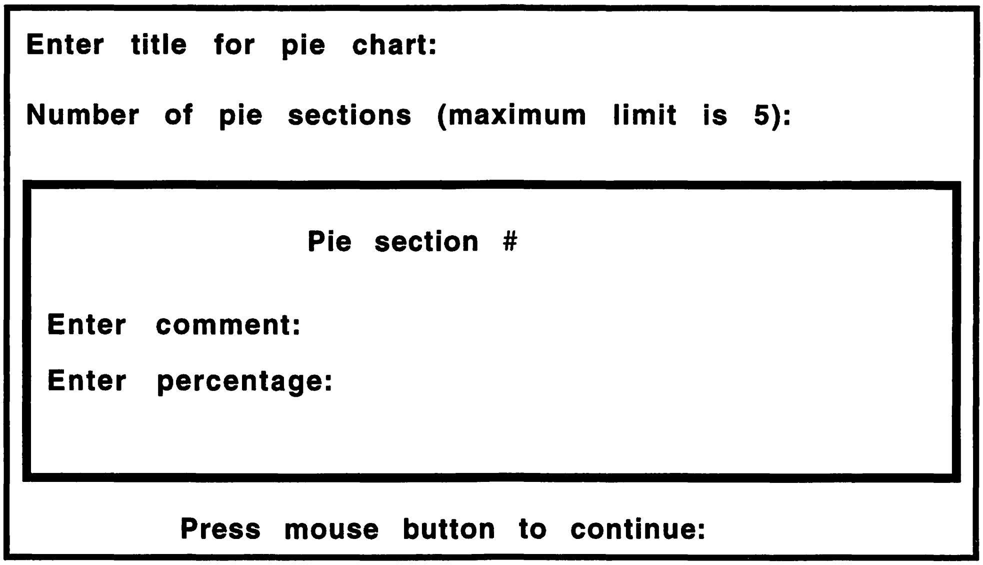
Figure 14.10b Frame of reference with respect to angles.

Let us apply some of these QuickDraw routines to drawing a pie chart. Assume that we need a program to draw a pie chart with a maximum of five sections. Each section of the pie is to be labeled with a comment and a percentage. Input required from the terminal is as follows: a title for the pie chart, the number of sections, and the comment and percentage for each section (the percentage must be an integer number). An example of the window that prompts the user for information is shown in Figure 14.11. This window uses an outside rectangle: the left top point is (50, 50), and the right bottom point is (462, 292). The inner rectangle's left top point is (80, 135) and right bottom point is (432, 247). The program is designed to check for two possible errors: (1) whether any sections lie outside the range 1 through 5, and (2) whether the sum of the percentages of the pie sections does not equal 100. If either of these tests fails, the Macintosh will emit a short beep, redraw the prompting window, and require the user to reenter the information.

The body of the main module contains the following major steps:

1. Hide all Macintosh Pascal windows.
2. Open two grafPorts: one for the prompt window and one for the pie chart.
3. Show the prompt window, and enter requested data from the keyboard.
4. Show the pie chart window, and draw the pie chart.

5. Close and dispose of all grafPorts.



```
Enter title for pie chart:

Number of pie sections (maximum limit is 5):

                    Pie section #

 Enter comment:

 Enter percentage:

           Press mouse button to continue:
```

Figure 14.11 Prompt window for entering data when executing the pie-chart program.

The following user-defined types and variables are employed:

```
type
      Port = GrafPtr;
      Pie_Section =  record
                        Comment : string[20];
                        Percentage : integer
                     end;
      Pie_Table = array[1..5] of Pie_Section;
var
      Pie_Data : Pie_Table;
      Prompt_Window, Piechart_Window : Port;
      Title : string[45];
      Sections : integer;
```

The initial design for this system is represented by the structure diagram shown in Figure 14.12, which shows the direction of data that is passed between modules. Pascal code for the main module requires only a sequential set of procedure calls. The modules for opening and closing grafPorts are identical to the procedures `Open_Window` and `Dispose_of_Window` in the program `Drawing_Windows` and need not be discussed again. These two procedures should exist before any other procedures are tested. `Open_Window` establishes the structure and initially opens a `grafPort`, and `Dispose_Window` closes a `grafPort` and disposes of its corresponding data structure.

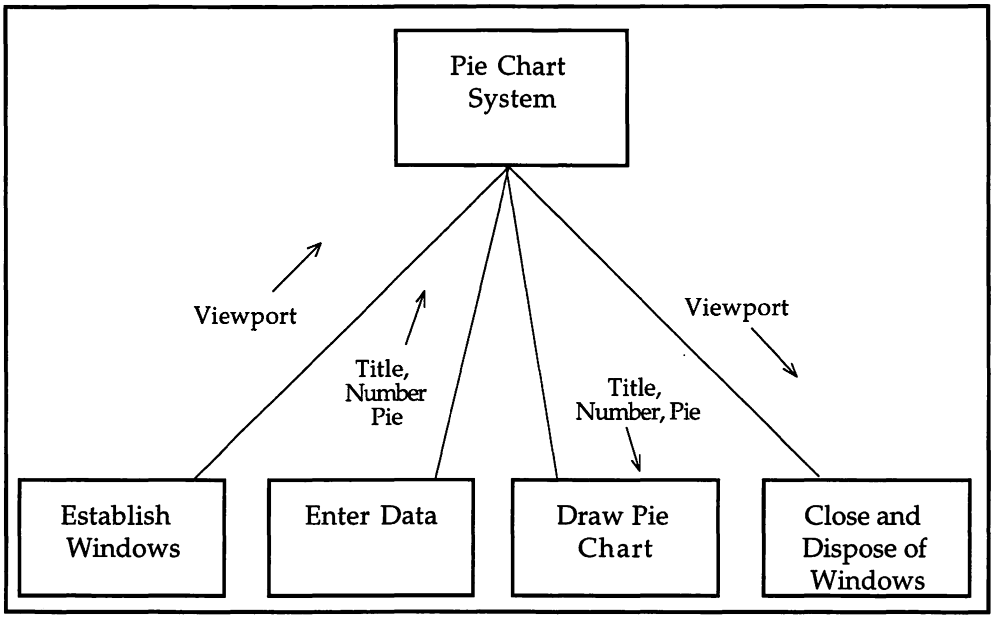


Figure 14.12 Structure diagram for the pie-chart program.

Here are the steps for showing the prompt window and allowing the data to be entered from the keyboard:

```
1. Set the prompt window as the current window.
2. Establish the clipping region and the rectangle boundary for viewing a
   frame.
3. Fill the rectangle with white background, and draw a frame for the
   rectangle.
4. Prompt user for title, and enter this title.
5. Prompt user for number of sections, and enter this value.
6. Check if number of sections is within the range 1 through 5.
7. If within bounds, then
   (a) initialize the control counter, and partial sum.
   (b) while the counter is less than or equal to the number of sections, do
       (i)    establish the boundary of inner box, fill the box with white
              background, and draw the frame;
       (ii)   prompt user for comment, and enter the comment;
       (iii)  prompt user for percentage, and enter the amount;
       (iv)   continue to compute partial sum for total percentages;
       (v)    prompt user to continue by pressing mouse button;
       (vi)   erase prompt message;
       (vii)  increment control variable counter by 1;
       end { while-do loop };
   (c) Check if total sum of percentages is equal to 100;
   end { if-then };
8. If the parameters are not within bounds, then beep the user.
9. Repeat Steps 3 through 8 until all parameters are within bounds.
```

Here is the Pascal code for performing these steps:

```
procedure Enter_Data (var Title : string; var Number :
            integer; var Pie : Pie_Table);
   var
      Rectangle, Box : Rect;
      Counter, Partial_Sum : integer;
      Within_Bounds : Boolean;
begin
{ Set grafPort for prompt window. The variable Prompt_Window is }
{ global to this procedure. }
   SetPort(Prompt_Window);
{ Establish clipping region and boundary for viewing frame. }
   SetRect(Rectangle, 50, 50, 462, 292);
   ClipRect(Rectangle);
   repeat
{ Fill rectangle with white background and draw frame of view }
{ port. }
      FillRect(Rectangle, white);
      PenSize(3, 3);
      FrameRect(Rectangle);
   { Prompt user for title. }
      MoveTo(60, 80);
      DrawString(' Enter title for pie chart:  ');
      readln(Title);
      MoveTo(64, 100);
      DrawString(Title);
   { Prompt user for number of sections. }
      MoveTo(60, 120);
      DrawString(' Number of pie sections (maximum limit is 5):');
      readln(Number);
      MoveTo(340, 120);
      DrawString(StringOf(Number));
   { Check if the number of sections is within range. }
      Within_Bounds := (Number >= 1) and (Number <= 5);
   { If within bounds, enter comment for each section and }
   { percentage. }
      if Within_Bounds then
         begin
            Counter := 1;
            Partial_Sum := 0;
            while Counter <= Number do
               begin
               { Draw frame of inside box and display subtitle. }
                  SetRect(Box, 80, 135, 432, 247);
                  FillRect(Box, white);
                  FrameRect(Box);
                  MoveTo(180, 165);
                  DrawString(concat('Pie section #',
                              StringOf(Counter:1)));
```

```
                { Prompt user for section's comment. }
                  MoveTo(90, 195);
                  DrawString('Enter comment :  ');
                  readln(Pie[Counter].Comment);
                  MoveTo(240, 195);
                  DrawString(Pie[Counter].Comment);
                { Enter percentage.}
                  MoveTo(90, 215);
                  DrawString('Enter percentage : ');
                  readln(Pie[Counter].Percentage);
                  MoveTo(240, 215);
                  DrawString(StringOf(Pie[Counter].Percentage:2));
                { Continue to compute partial sum. }
                  Partial_Sum := Partial_Sum +
                  Pie[Counter].Percentage;
                { Prompt user for continuing execution. }
                  MoveTo(140, 270);
                  DrawString('Press mouse button to continue: ');
                  Pushbutton;
                  MoveTo(140, 270);
                  SetRect(Box, 140, 260, 400, 270);
                  FillRect(Box, white);
                { Modify loop-control variable. }
                  Counter := Counter + 1
                end;
          { Check if partial sum is equal to 100%. }
             Within_Bounds := (Partial_Sum = 100);
          end;
        { Beep user if not within bounds. }
        if not Within_Bounds then
        SysBeep(10);
   until Within_Bounds
end; { Enter_Data }
```

This procedure is composed and tested in parts. First, we draw the outer rectangle to see how it is positioned on the screen. Then we add prompts for the title and number of sections, and test the procedure again to view the prompts and echo their respective inputs. We can make corrections if prompts and echoed values need to be repositioned. As you will notice, checking bounds on the number of sections (also on the sum of percentages) is performed with a simple assignment statement; the right side is a relational expression. The body of the loop is then added to draw the inner rectangle and tested to see how the rectangle is positioned on the screen. Then we add prompts for comment and percentage and test them. Notice that the QuickDraw procedure `DrawString` is used to draw a single text string to the screen instead of the Macintosh Pascal procedure `WriteDraw`. Keep in mind that our aim in this chapter is to use as many of the QuickDraw routines as possible.

The procedure `StringOf` from the Macintosh Pascal library is borrowed for converting a numeric value into a string representation in order to read numeric data and echo it to the screen. You may ask how the equivalent action could be executed directly in Pascal if `StringOf` or `WriteDraw` were not available. There are several possible tricks you could use; the answer is left as an exercise. At one point we used the concat procedure to concatenate two strings before executing the procedure DrawString. For example, the message Pie section # is concatenated to the value of StringOf(Counter:1).

The second module of importance to our pie-chart program is the one that displays the pie-chart window and draws the pie chart, including comments and percentages for each of the sections. Here are the general steps for performing these actions:

1. Set the current grafPort to the pie-chart window.
2. Establish a clipping window and boundary for viewing a frame.
3. Fill the viewing frame with a white background.
4. Compute the starting angle and arc angle, and set a fill pattern for each section.
5. Draw a title for the pie chart.
6. Set the region for an oval, and draw each section of the pie chart.
7. Label each section of the pie chart.

Step 4 needs some elaboration. First, we need to introduce the local variables: Starting_Angle, Arc_Angle, and Counter. The variable Starting_Angle always contains the starting angle for the next wedge (pie section) and is determined by adding the present value of Starting_Angle and Arc_Angle. In turn, Arc_Angle is computed from the truncated product of a section's percentage and the constant 3.60. The fill pattern is chosen from a function called Pattern_Picker, which returns a value of type Pattern and uses a simple table lookup in the form of a **case** statement to select one of the five standard patterns. All information for drawing a wedge is stored in an array of records, each record having a field for starting angle, arc angle, and fill pattern. The algorithmic steps are as follows:

```
Starting_Angle <-- 0;
for Counter <-- 1 to Number_of_Sections
   begin
      Wedge[Counter].Start <-- Starting_Angle;
      Arc_Angle <-- Round( Pie[Counter].Percentage * 3.60);
      Wedge[Counter].Arc <-- Arc_Angle;
      Wedge[Counter].pat <-- Pattern_Picker( Counter );
      Starting_Angle <-- Starting_Angle + Arc_Angle
   end;
```

The function round is used instead of trunc to avoid the accumulation of truncation errors that could leave a small gap in the pie chart.

Labeling each section of the pie chart also needs further explanation. First, the pie chart drawn is assumed to be circular and not elliptical. Second, the pie chart is drawn within an oval having a center located at the point (250, 195) and a radius of 100. The point for drawing the comment and percentage is located along an angle given by the starting angle plus (Arc_Angle **div** 2) for each individual wedge. Since this is an angle in the wedge and arc coordinate system, it must be adjusted for the coordinate plane by subtracting 90 from its value. This value must then be converted into radians for the application of the trigonometric functions cosine and sine. The starting point for drawing a comment, where the angle for location is less than 180 degrees, is given by the expressions

```
X <-- 250 + trunc(120 * cos(Radian_Angle)),
Y <-- 195 + trunc(120 * sin(Radian_Angle)).
```

Remember that the center of the oval is equivalent to the translation of the origin from the point (0, 0) to (250, 195). If the angle for location is greater than 180 degrees, the value of X is modified by subtracting the length of the comment. The algorithmic steps follow:

```
for Counter <-- 1 to Number_of_Sections
   begin
      Angle <-- Wedge[Counter].Start + Wedge[Counter].Arc div 2;
      Radian_Angle <-- pi * ( Angle - 90 ) / 180;
      X<-- 250 + trunc( 120 * cos(Radian_Angle) );
      Y<-- 195 + trunc( 120 * sin(Radian_Angle) );
      if Angle > 180 then
         X<-- X - StringWidth( Pie[Counter].Comment );
      MoveTo(X, Y);
      DrawString( Pie[Counter].Comment );
      Y<-- Y + 15;
      MoveTo(X, Y);
      DrawString( concat( StringOf( Pie[Counter].Percentage:2 ),
                              '%') )
   end;
```

This requires three new local integer variables, called X, Y, and Angle, and a real variable called Radian_Angle. Here is the Pascal code for this module:

```
procedure Draw_Piechart (Title : string;  Number_of_Sections :
                              integer; Pie : Pie_Table);
   type
      Wedge_Element =   record
                           Start,  Arc : integer;
                           Pat : Pattern
                        end;
   var
      Counter, Starting_Angle, Arc_Angle, Angle, X, Y : integer;
      Radian_Angle : real;
      Wedge : array[1..5] of Wedge_Element;
      Rectangle, Oval : Rect;
begin
{ Set grafPort for pie chart. The variable Piechart_Window is }
{ global to this procedure. }
   SetPort(Piechart_Window);
{ Establish clipping window and boundary for viewing. }
   SetRect(Rectangle, 0, 20, 512, 342);
   ClipRect(Rectangle);
{ Fill background with white pattern. }
   FillRect(Rectangle, white);
{ Compute starting angle, arc angle, and fill pattern for each }
{ section. }
   Starting_Angle := 0;
   for Counter := 1 to Number_of_Sections do
      begin
         Wedge[Counter].Start := Starting_Angle;
         Arc_Angle := Round( Pie[Counter].Percentage * 3.60 );
         Wedge[Counter].Arc := Arc_Angle;
         Wedge[Counter].Pat := Pattern_Picker(Counter);
         Starting_Angle := Starting_Angle + Arc_Angle
      end;
{ Draw title for pie chart. }
   MoveTo(60, 80);
   DrawString(Title);
{ Set region for oval and draw pie chart. }
   SetRect(Oval, 150, 95, 350, 295);
   for Counter := 1 to Number_of_Sections do
      FillArc( Oval, Wedge[Counter].Start,
            Wedge[Counter].Arc, Wedge[Counter].Pat);
   FrameOval(Oval);
{ Label each section of the pie. }
   for Counter := 1 to Number_of_Sections do
      begin
         Angle:= Wedge[Counter].Start + Wedge[Counter].Arc div 2;
         Radian_Angle := pi * (Angle - 90) / 180;
         X := 250 + trunc(120 * cos(Radian_Angle));
         Y := 195 + trunc(120 * sin(Radian_Angle));
         if Angle > 180 then
      { Move farther from the left side of the chart. }
            X := X - StringWidth(Pie[Counter].Comment);
      { Move to a drawing point and display comment and }
      { percentage. }
         MoveTo(X, Y);
         DrawString(Pie[Counter].Comment);
         Y := Y + 15;
         MoveTo(X, Y);
         DrawString(
         concat(StringOf(Pie[Counter].Percentage : 2), ' %'));
      end;
end;
```

The QuickDraw function called `StringWidth` is employed to adjust the value of `X`. This procedure measures the length of the comment as related to its length on the screen. Figure 14.13 shows the results of the program using these procedures, with five sections for the pie chart.

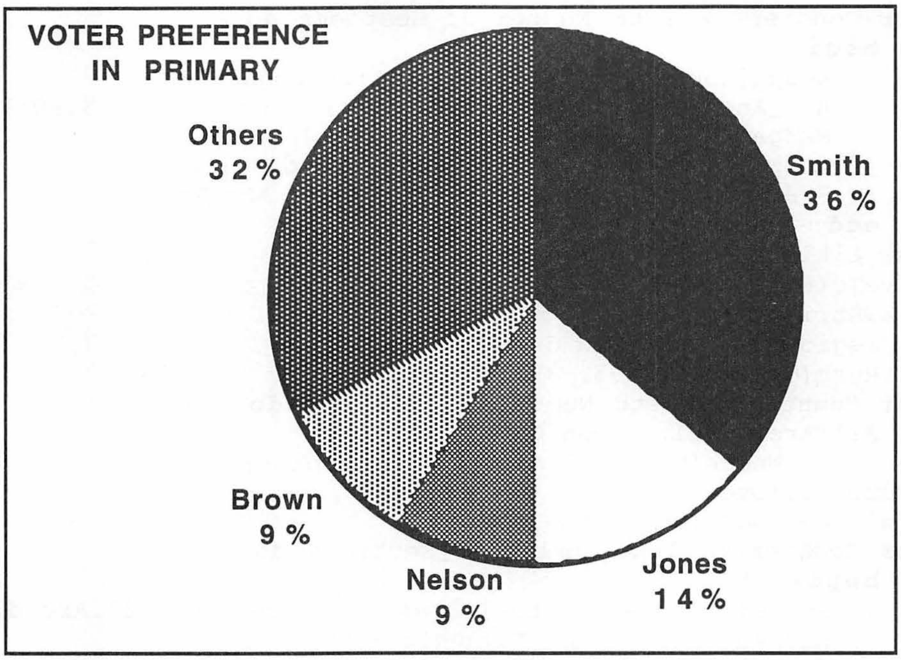


Figure 14.13 Sample output from the pie-chart program.

14.6 TEXT-DRAWING ROUTINES

QuickDraw not only draws lines and shapes but also contains the functions for displaying text characters in various fonts, styles, and modes. For any opened `grafPort`, there are five fields as part of the `grafPort` data structure: `txFont`, `txFace`, `txMode`, `txSize`, and `spExtra`.

A font is defined as a complete set of type having one particular size and face. Not every font used on the Macintosh system has the same number of characters in its set. The field `txFont` of a `grafPort` structure contains an `integer` value indicating the type of font available for display. Whenever a `grafPort` is opened, the `txFont` field is assigned an initial value of 0, representing the system font. The field `txFace` is of type `Style` and is used to control the appearance of the font. The type `Style` has the following definition:

```
type
   StyleItem = ( bold, italic, underline, outline, shadow,
                    condense, extend);
   Style = set of StyleItem;
```

Notice that there are seven types of styles: `bold`, for drawing characters with extra thickness; `italic`, for drawing characters with a slant; `underline`, for drawing a baseline below a character; `outline`, for presenting a shallow character (not solid); `shadow`, for making a heavier outlined character; and `condense` and `extend`, for changing the horizontal distance between characters. The thickness of a character can only be controlled by changing the value of the txFace field. The routine PenSize has no effect on characters drawn by DrawString and DrawChar. The field txMode, controlling how characters are mapped to the bit image, functions much as does the field pnMode. The field txSize, an integer type, specifies the size of the font measured in points (a point is approximately 1/72 of an inch). A default value of 0 tells QuickDraw to use the system font size of 12 points. QuickDraw will automatically adjust the value of this field for any font not having a specified size. The field spExtra is important when characters drawn along a line are fully justified; that is, when characters are aligned to both the left and right margins.

Figure 14.14 lists procedures and functions useful for drawing text characters to a grafPort. Because the fields pnSize, pnPat, and pnMode have no effect on text drawing, the field pnLoc is used. Each character is placed to the right of the current pen location. Special effects, such as the wrapping of words or carriage returns, are not automatically performed as they are when using the Text window. As Figure 14.14 shows, drawing text on a new line requires executing the procedure MoveTo to adjust the pen location.

procedure TextFont(New_Font : integer); This procedure sets the txFont field of the current grafPort to the value of New_Font.

procedure TextFace(Text_Face : Style); This procedure sets the txFace field of the current grafPort to one of the following predefined constants: bold, italic, underline, outline, shadow, condense, or extend. A value of [ ] for Text_Face assigns a normal style.

procedure TextMode(Text_Mode : integer); This procedure sets the current grafPort's transfer mode for drawing text characters. Default value is srcOr, and only two other values should be used: srcXor, srcBic.

procedure DrawChar(Given_Char : char); This procedures draws the given character to the right of the current pen location, advancing the pen to the right by the width of the character.

procedure DrawString(Given_String : Str255); This procedure draws the given string to the right of the current pen location. The pen is advanced to the right of the string.

function CharWidth(Given_Char : char) : integer; This function returns the width of the given character based on the current grafPort's font, size of font, and style of font.

function StringWidth(Given_String : Str255) : integer; This function returns the width of the given string based on the current grafPort's font, size of font, and style of font.

Figure 14.14 QuickDraw procedures and functions for drawing text characters to a grafPort.

Notice that in the procedure Enter_Data, listed earlier, a complete line of text is entered from the keyboard (by executing the readln statement) before being echoed to the screen. You may feel it is more appealing to see the characters echoed as they are entered one at a time from the keyboard. The following procedure, Read_Response, shows the steps for doing that.

```
procedure Read_Response (var Char_String : string; Max_Length :
                                   integer);
   var
      J : integer;
      Character : Char;
begin
{ Initialize control variable and character string. }
   J := 1;
   Char_String := '';
   while not eoln do
   { Read characters and concatenate on the right to character }
   { string. }
      begin
         read(Character);
      { If counter J is less than the allowed maximum length of }
      { the string, draw the character to the screen and }
      { concatenate on the right with a character string. }
         if J <= Max_Length then
            begin
               if (ord(Character) >= 32) and (ord(Character) <=
                     126) then
                  begin
                     DrawChar(Character);
                     Char_String := concat(Char_String,Character);
                     J := J + 1
                  end;
            end;
      end;
    readln;
end;
```

First, a counter called J is employed to ensure that the variable Char_String does not exceed a maximum length. Second, each character is read as it is entered from the keyboard. It is then checked to see if it is in the decimal range of 32 (a space) through decimal 126 (the character ~). This prevents the drawing of a small box that represents any of the special characters outside this range, such as the Backspace. Third, if the ordinal value of the character is within range, the character is drawn by DrawChar and then concatenated to the right of the character string. This procedure assures us that no strange characters will be attached to the character string, but it does not allow us to delete unwanted characters by pressing the Backspace key.

The procedure Enter_Data requires the following changes to incorporate the steps of Read_Response: The lines

```
MoveTo(64,100);
Read_Response(Title,45);
```

replace

```
readln(Title);
MoveTo(64, 100);
DrawString( Title);
```

and the lines

```
MoveTo(240, 215);
Read_Response(Pie[Counter].Comment,20);
```

replace

```
readln(Pie[Counter].Comment);
MoveTo(240, 215);
DrawString(Pie[Counter].Comment);
```

14.7 DRAWING WITH REGIONS AND POLYGONS

A region is defined as a nonrectangular figure having a closed boundary. It can be formed from a combination of calls to procedures for drawing framed shapes of rectangles, ovals, lines, and polygons (except for calls that draw arcs). Regions can also be defined through the intersection, union, and subtraction of other regions. Because we are able to define a region as a nonrectangular object, it is possible to establish clipping regions that are of arbitrary shapes and to dictate to QuickDraw the regions that are visible to the user. Figure 14.15 lists routines for creating and drawing regions. Regions are represented by a specially defined type called a `RgnHandle`. A handle is an indirect pointer to a variable, so it does not point directly to a dynamic variable, but to another pointer that points to a dynamic variable.

function `NewRgn : RgnHandle;` This function allocates space for the dynamic data structure of a new region, initializing it as an empty region with the top left point being (0, 0), and the bottom right point being (0, 0). The value returned is a handle to this new structure. All regions must be allocated using this function before any reference can be made to that region. No other procedures or functions are capable of creating a region.

procedure `DisposeRgn( Source_Rgn : RgnHandle );` This procedure deallocates storage for the data structure representing the source region and returns this space to a free memory pool in the Pascal system.

procedure `CopyRgn( Source_Rgn, Dest_Rgn : RgnHandle );` This procedure makes a duplicate copy of the source region and assigns this copy to the destination region. Later changes in the source region have no effect on the destination region.

procedure `SetEmptyRgn( Present_Rgn : RgnHandle );` This procedure forces the present region to become an empty region, destroying its previously defined structure.

procedure `SetRectRgn(Present_Rgn : RgnHandle; left, top, right, bottom : integer);` This procedure forces the present region to become a rectangular region with boundary points (left, top) and (right, bottom), destroying its previously defined structure. If left is greater than or equal to right, or if top is greater than or equal to bottom, the assigned region will be empty.

procedure `RectRgn( Present_Rgn : RgnHandle; Box : Rect );` This procedure forces the present region to become the region specified by `Box`, destroying its previously defined structure.

procedure `OpenRgn;` This initially begins to collect the commands forming the definition of a region, saving the information related to the region in the current `grafPort's rgnSave` field. While a region is open, the pen is hidden from view. Only one region should be opened at a time.

procedure `CloseRgn( Dest_Rgn : RgnHandle );` This procedure closes a region by ending the collection of commands defining the boundary of the region, assigning the region to the destination region.

procedure `SectRgn( Source_A, Source_B, Dest_Rgn : RgnHandle );` This procedure computes the intersection of regions given by `Source_A` and `Source_B`, assigning the result to the destination region. A destination region can be one of the source regions.

procedure `UnionRgn( Source_A, Source_B, Dest_Rgn : RgnHandle );` This procedure computes the union of regions given by `Source_A` and `Source_B`, assigning the result to the destination region. A destination region can be one of the source regions.

procedure `DiffRgn( Source_A, Source_B, Dest_Rgn : RgnHandle );` This procedure subtracts region `Source_B` from region `Source_A`, assigning the result to the destination region. The destination region can be one of the source regions.

procedure `XorRgn(Source_A,Source_B, Dest_Rgn: RgnHandle );` This procedure computes the difference between the union and intersection of the two source regions, assigning the result to the destination region. The destination region can be one of the source regions.

function `PtInRgn( Source_Pt : Point; Present_Rgn : RgnHandle ): Boolean;` This function will test for the presence of a pixel below and to the right of coordinates given by `Source_Pt` within the region specified by `Present_Rgn`. It returns a value of *true* if it exists, and *false* otherwise.

function `RectInRgn( Source_Rect : Rect; Present_Rgn : RgnHandle ): Boolean;` This function will test for the intersection of the source rectangle with the present region. The value of this function is *true* if the intersection encloses at least one pixel, and *false* otherwise.

function `EqualRgn( Region_A, Region_B : RgnHandle ): Boolean;` This function compares the two regions `Region_A` and `Region_B` for equality. If the regions have identical shapes, sizes, and locations, the value returned is *true*; otherwise it is *false*.

function `EmptyRgn( Source_Rgn : RgnHandle ): Boolean;` This function returns a value of *true* if the source region is empty, and *false* otherwise.

procedure `FrameRgn( Source_Rgn : RgnHandle );` This procedure draws the shell of the specified region using the current `grafPort's` pen state (pattern, mode, and size). Pen location remains unchanged.

procedure `PaintRgn( Source_Rgn : RgnHandle );` This procedure paints the specified region using the pattern and mode of the current `grafPort`. Pen location remains unchanged.

procedure `EraseRgn( Source_Rgn : RgnHandle );` This procedure paints the specified region using the background pattern of the current `grafPort`. Pen location remains unchanged.

procedure `InvertRgn( Source_Rgn : RgnHandle );` This procedure inverts the pixels within the boundary specified by the source region. Pen location remains unchanged.

procedure `FillRgn( Source_Rgn : RgnHandle; Pen_Pat : Pattern );` This procedure fills the specified region given by the source with the pattern specified by `Pen_Pat`. Pen location remains unchanged.

Figure 14.15 QuickDraw procedures and functions for creating and using regions.

Here is the definition of this type:

```
type
   RgnHandle    =   ^RgnPtr;
   RgnPtr       =   ^Region;
   Region       =   record
                       rgnSize : integer;
                       rgnBBox : Rect;
                    { Additional data if region is nonrectangular. }
                    end;
```

The field `rgnSize` contains the memory size required for storing the region, and the field `rgnBBox` represents a rectangle that encloses the region.

All dynamic structures representing regions are allocated by executing the function `NewRgn`. No region should be opened, copied, closed, disposed, operated on, designated as a destination region, merged, or separated unless its data structure has first been allocated by `NewRgn`. As for an arbitrary-shaped region, it is defined by first executing the routine `OpenRgn`, defining the routines for framing the region but not drawing the region, then closing the definition of the region by executing the routine `CloseRgn`. *Only one region should be opened at any one time, and it should be closed when it no longer needs to be be defined.* Once a region has been closed, it can be drawn to the screen by executing one of the routines `EraseRgn`, `FrameRgn`, `PaintRgn`, and `FillRgn`. Data structures for regions are deallocated by executing the routine `DisposeRgn`. This routine should only be executed when the region is no longer useful or before the program ends execution.

Consider the following example program, titled `Drawing_Regions`.

```
program Drawing_Regions;
{ Program:  This example shows how a region of arbitrary shape }
{           can be generated, using several region procedures. }
   type
      Port = GrafPtr;
   var
      Window : Port;
      Rectangle : array[1..2] of Rect;
      J : integer;
      Actual_Rgn : array[1..3] of RgnHandle;
{ ****************************************************** }
   procedure Open_Window (var Viewport : Port);
   begin
      new(Viewport);
      OpenPort(Viewport)
   end;
{ ****************************************************** }
   procedure Dispose_of_Window (var Viewport : Port);
   begin
      ClosePort(Viewport);
      Dispose(Viewport)
   end;
{ ****************************************************** }
```

```
   procedure Grow_Region (var Rgn : RgnHandle;  var Box : Rect;
                          Left, Top, Right, Bottom : integer);
   begin
      OpenRgn;
      SetRect(Box, Left, Top, Right, Bottom);
      FrameOval(Box);
      CloseRgn(Rgn);
   end;
{ ********************************************************* }
begin
{ Hide all Macintosh Pascal windows. }
   HideAll;
{ Establish two initial windows. }
   Open_Window(Window);
   for J := 1 to 3 do
      Actual_Rgn[J] := NewRgn;
{ Generate the first region. }
   Grow_Region(Actual_Rgn[1], Rectangle[1], 56, 73, 356, 273);
{ Generate the second region. }
   Grow_Region(Actual_Rgn[2], Rectangle[2], 206, 103, 306, 243);
{ Generate third region. }
   DiffRgn(Actual_Rgn[1], Actual_Rgn[2], Actual_Rgn[3]);
{ Show the third region. }
   SetPort(Window);
   Window^.clipRgn := Actual_Rgn[3];
   FillRgn(Actual_Rgn[3], white);
{ Paint border on the edges of the third region. }
   PenSize(2, 2);
   FrameRgn(Actual_Rgn[3]);
{ Draw in the third region. }
   MoveTo(100, 160);
   TextFace([italic, underline]);
   DrawString('Third Region');
{ Close windows and dispose of all regions. }
   Dispose_of_Window(Window);
   for J := 1 to 3 do
      DisposeRgn(Actual_Rgn[J]);
end.
```

This program draws the region shown in Figure 14.3. The procedure Grow_Region generates the instructions representing the first two regions, called Actual_Rgn[1] and Actual_Rgn[2]. In the body of this procedure, SetRect establishes a rectangle as a boundary for framing an oval. During execution, Actual_Rgn[1] represents the area within the large oval, and Actual_Rgn[2] represents the area within the small oval. The third region, Actual_Rgn[3], is generated by subtracting Actual_Rgn[2] from Actual_Rgn[1], and assigning the resulting region to this variable. The region is drawn by first setting the current port to Window, assigning this third region as the clipping region of Window, and then filling the region with a white background. After setting the pen size, the frame of this third region is painted using the procedure FrameRgn. The last set of instructions draws text into this third region.

Notice that the procedure `DisposeRgn` is executed at the end of the program to avoid the risk of bombing the program.

Drawing polygons is similar to defining and drawing regions. Figure 14.16 lists the procedures and functions for defining and drawing with polygons. Mathematically, a polygon is a closed plane figure bounded by three or more line segments. In Pascal a polygon is drawn with a closed set of three or more connected line segments. It is represented by a special data type called *`polyHandle`*. Like a region handle, a polyhandle represents an indirect pointer to a dynamic structure created by execution of the function `OpenPoly`.

function `OpenPoly : PolyHandle;` This function allocates a data structure for a polygon, indicating to QuickDraw that it is ready to receive draw instructions of either `Line` or `LineTo`. At this point the pen is hidden.

procedure `ClosePoly;` This procedure stops the polygon from collecting line instructions and shows the pen for drawing.

procedure `KillPoly( Source_Poly : PolyHandle );` This procedure deallocates storage for the data structure representing the source polygon and returns this space to the free memory pool of the Pascal system. This procedure should only be used if the polygon is no longer needed.

procedure `FramePoly( Source_Poly : PolyHandle );` This procedure draws the border of the polygon using the pen state (mode, size, and pattern) of the current `grafPort`. In this procedure the pen is always located below and to the right of each boundary point of the polygon itself, forcing the polygon to extend beyond the right and bottom edges of the rectangle specified by the field `polyBBox`.

procedure `PaintPoly( Source_Poly : PolyHandle );` This procedure paints the area within the source polygon, using the pen pattern and pen mode of the current `grafPort`. Pen location remains unchanged.

procedure `ErasePoly( Source_Poly : PolyHandle );` This procedure paints the area of the source polygon using the background pattern of the current `grafPort`. Pen location remains unchanged.

procedure `InvertPoly( Source_Poly : PolyHandle );` This procedure inverts each pixel of the source polygon. Pen location remains unchanged.

procedure `FillPoly( Source_Poly : PolyHandle; Pen_Pat : Pattern );` This procedure fills the area of the source polygon with the pattern given by `Pen_Pat`. Pen location remains unchanged.

Figure 14.16 QuickDraw procedures and functions for defining and drawing polygons.

The following declarations define this type of handle:

```
type
   PolyHandle   =   ^PolyPtr;
   PolyPtr      =   ^Polygon;
   Polygon      =   record
                       polySize   : integer;
                       polyBBox   : Rect;
                       polyPoints : array[0..0] of Point
                    end;
```

The field `polySize` provides the amount of memory needed for storing the polygon, and the field `polyBBox` represents the rectangle that encloses the polygon. For the third field, `polyPoints`, the bounds of the array can be changed for dynamically storing all of the points representing the line segments of the polygon.

Before a polygon can be used, it must be allocated storage by executing the function `OpenPoly`. Like `OpenRgn`, this function hides the pen and readies the data structure of the polygon for points representing connecting line segments. These points can be generated by executing procedures such as `Move`, `MoveTo`, `Line`, and `LineTo`. This definition ends with the execution of procedure `ClosePoly`. The `ClosePoly` procedure tells QuickDraw to stop saving definitions and show the pen for drawing. *Never open an additional polygon if one is already open.* After execution of `ClosePoly`, a polygon can be drawn by using a procedure such as `FramePoly`, `PaintPoly`, `ErasePoly`, or `FillPoly`. The data structure for a polygon is deallocated by executing the procedure `KillPoly`. Like `DisposeRgn`, this procedure returns the storage space used by the data structure to the pool of free memory in the Pascal system.

14.8 DRAWING PICTURES

QuickDraw allows us to save drawing commands in a special object called a *picture*, with the flexibility of drawing this picture at a later time. The details for drawing a picture can be masked from the user in a procedure or function. Each picture that is defined requires a picture frame that surrounds it. The picture is drawn in a destination frame that is within the boundary of the picture frame. Figure 14.17 lists the functions and procedures for defining and drawing pictures.

function `OpenPicture( Picture_Frame : Rect ): PicHandle;` This function allocates storage for a picture. The rectangle specifies the area in which picture commands are to be drawn. This function results in hiding the pen.

procedure `ClosePicture;` This procedure stops commands from being saved by an opened picture. Only one picture should be open at any one time.

procedure `DrawPicture( Saved_Picture : PicHandle; Dest_Frame : Rect );` This procedure draws the saved picture within the destination frame, expanding or shrinking the picture as necessary. The destination frame must be within the boundary of the picture frame, or else no picture is drawn.

procedure `KillPicture( Saved_Picture : PicHandle );` This procedure deallocates storage for the data structure of the saved picture, returning it to the free memory pool of the Pascal system. This procedure should only be used when the saved picture is no longer needed.

Figure 14.17 QuickDraw procedures and functions for defining and drawing pictures.

Like regions and polygons, pictures require special data types in Pascal. All pictures are represented by picture handles; a picture handle points to a picture pointer; in turn, the picture pointer points to a record called a *picture*:

```
type
   PicHandle    = ^PicPtr;
   PicPtr       = ^Picture;
   Picture      = record
                     picSize : integer;
                     picFrame : Rect;
                  { Data and  commands for drawing the picture. }
                  end;
```

The field `picSize` contains the size of memory needed to store the picture, the field `picFrame` outlines the rectangle surrounding the defined picture. These fields are followed by coded data and commands for drawing the picture. Like regions and polygons, a data structure representing a picture must be allocated by executing the function `OpenPicture`. This function hides the pen and collects all drawing commands to the picture. This collection of drawing commands continues until the procedure `ClosePicture` is executed. This procedure shows the pen but does not draw the picture. The procedure `DrawPicture` must be executed to draw an actual picture.

Consider the following example program, titled Drawing_Pictures.

```
program Drawing_Pictures(input, output);
{ Program:  This program draws a pie chart as a picture. This }
{           picture is then drawn in the upper right corner of }
{           the screen in what is referred to as a destination }
{           frame. The whole screen represents the picture }
{           frame for defining the picture. }
   type
      Port = GrafPtr;
```

```
     Pie_Section =  record
                        Comment : string[20] ;
                        Percentage : integer
                    end;
     Pie_Table = array[1..5] of Pie_Section;
  var
     Pie_Data : Pie_Table;
     Prompt_Window, Piechart_Window : Port;
     Title : string[45];
     Sections : integer;
     Mypicture : PicHandle;
     Picture_Frame, Dest_Frame : Rect;
{ To keep this listing short,  procedures and functions are not }
{ defined. }
begin { Main body of the program. }
{ Hide all Macintosh Pascal windows. }
  HideAll;
{ Establish data structures and open the two grafPorts. }
  Open_Window(Prompt_Window);
  Open_Window(Piechart_Window);
{ Show prompt window and enter relevant data. }
  Enter_Data(Title, Sections, Pie_Data);
{ Set picture frame and destination frame for drawing picture. }
  SetRect(Picture_Frame, 0, 20, 512, 342);
  SetRect(Dest_Frame, 200, 0, 512, 190);
{ Set drawing window for picture commands. }
  SetPort(Piechart_Window);
  EraseRect(Picture_Frame);
  begin
  { Open picture for picture commands. }
     Mypicture := OpenPicture(Picture_Frame);
  { Assign drawing commands to current grafPort. }
     Draw_Piechart(Title, Sections, Pie_Data);
  { Close picture. }
     ClosePicture;
  end;
{ Draw the picture in the destination frame. }
  DrawPicture(Mypicture, Dest_Frame);
{ Close and dispose of storage for all windows and pictures. }
  Dispose_of_Window(Prompt_Window);
  Dispose_of_Window(Piechart_Window);
  KillPicture(Mypicture);
end.
```

This is the main program for Drawing_Pictures, except that all user-defined procedures and functions have been removed. In this example, only a few steps in the main program have been added to draw the pie chart as a picture called Mypicture. Two additional rectangles are defined: Picture_Frame, which picks the whole screen as the picture frame, and Dest_Frame, the destination frame where the pie chart will actually be drawn. After entering data from the keyboard, these two rectangles are set. This is followed by setting Piechart_Window for collecting picture commands. If we attempt to open a picture without a window being a current grafPort, the system will bomb. In addition, picture commands are always assigned to the current grafPort once the function OpenPicture is executed. After opening the picture, drawing commands are assigned to Mypicture when the procedure Draw_Piechart is executed. This means that all of our defined procedures and functions remain unchanged, with no additions or deletions. We stop assigning picture commands to Mypicture by executing ClosePicture. This is followed by executing the procedure DrawPicture for drawing Mypicture in a destination frame in the upper right of the screen. The procedure KillPicture is executed at the end of the program to free the data storage for Mypicture.

14.9 TRANSFER MODES AND BIT-TRANSFER OPERATIONS

Transfer modes are important because they control the transfer of individual pixels between bit maps, from source to destination, or between a pattern and a bit map. There are two types of transfer modes: pattern transfer, for drawing lines and shapes with a pattern, and source transfer, where bit images are transferred between bit maps. For each of these there are four basic operations: Copy, Or, Xor, and Bic. Figure 14.18 shows a table of logical equations between source and destination pixels for the transfer modes srcCopy, patCopy, srcOr, patOr, srcXor, patXor, srcBic, patBic, notSrcCopy, notSrcOr, notSrcXor, notSrcBic, notPatCopy, notPatOr, notPatXor, and notPatBic.

Transfer Mode	Value of Resulting_Pixel at Destination
srcCopy	Source_Pixel
patCopy	Pattern_Pixel
srcOr	Source_Pixel OR Destination_Pixel
patOr	Pattern_Pixel OR Destination_Pixel
srcXor[a]	Source_Pixel XOR Destination_Pixel
patXor[a]	Pattern_Pixel XOR Destination_Pixels
srcBic	NOT(Source_Pixel) AND Destination_Pixel
patBic	NOT(Pattern_Pixel) AND Destination_Pixel
notSrcCopy	NOT(Source_Pixel)
notPatCopy	NOT(Pattern_Pixel)
notSrcOr	NOT(Source_Pixel) OR Destination_Pixel
notPatOr	NOT(Pattern_Pixel) OR Destination_Pixel
notSrcXor[a]	NOT(Source_Pixel XOR Destination_Pixel)
notPatXor[a]	NOT(Pattern_Pixel XOR Destination_Pixel)
notSrcBic	Source_Pixel AND Destination_Pixel
notPatBic	Pattern_Pixel AND Destination_Pixel

[a]b XOR c = ((NOT b) AND c) OR (b AND (NOT c)).

Figure 14.18 Logical equations for determining the value of the resulting pixel during transfer modes.

The Copy operation simply copies all of the bits from the source to the destination without regard to the corresponding pixels in the destination. The or operation performs a binary or operation between corresponding source and destination pixels, Xor is a binary exclusive-or operation, and Bic is a binary bit-clear operation on source and destination pixels. In addition, the operations patOr and srcOr are equivalent to taking the source pixels and overlaying these on the destination pixels; patXor and srcXor are equivalent to using the source to invert the pixels of the destination; and patBic and srcBic are equivalent to using the source pixels to erase destination pixels.

QuickDraw has two important bit-transfer operations: ScrollRect and CopyBits. The procedure ScrollRect has the following header:

```
procedure ScrollRect( Rectangle : Rect; Delta_h, Delta_v :
                                              integer;
                           Updated_Region : RgnHandle );
```

This procedure allows bits contained within the intersection composed of the rectangle, visRgn, clipRgn, portRect, and portBits.bounds to be scrolled (shifted) through a horizontal distance specified by Delta_h, and vertical distance Delta_v. A positive direction is defined as *down* or *right.* Bits that lie outside the area intersected by the five regions are not affected, and bits inside the intersection are lost when scrolled outside this region. The region vacated by executing ScrollRect is filled with the background pattern of the current grafPort; the vacated region is represented by the region handle, Updated_Region.

The following program, titled Scrolling, demonstrates the use of the ScrollRect procedure.

```
program Scrolling(input, output);
{ Purpose:  Sample program that shows how a region of the screen }
{           can be scrolled. }
   type
      Port = GrafPtr;
      Rect_Table = array[1..3] of Rect;
   var
      Rectangle : Rect_Table;
      Window : Port;
      Hscroll, Vscroll : integer;
      Updated_Region : RgnHandle;
      Mousepoint : Point;
{ ********************************************************** }
   procedure Open_Window (var Viewport : Port);
      begin
         new(Viewport);
         OpenPort(Viewport)
      end;
{ ********************************************************** }
   procedure Initialize_Rectangles (var Box : Rect_Table);
      begin
         SetRect(Box[1], 130, 60, 350, 280);
         SetRect(Box[2], 148, 78, 332, 262);
         SetRect(Box[3], 151, 81, 329, 259)
      end;
```

```
{ ********************************************************* }
   procedure Draw_Regions (Box : Rect_Table);
      begin
         PenSize(3, 3);
         FillRect(Box[1], gray);
         FrameRect(Box[1]);
         FillRect(Box[2], white);
         FrameRect(Box[2]);
         PenSize(1, 1);
         FillOval(Box[3], ltgray);
         FrameOval(Box[3]);
      end;
{ ********************************************************* }
   procedure Dispose_of_Window (var Viewport : Port);
      begin
         ClosePort(Viewport);
         Dispose(Viewport)
      end;
{ ********************************************************* }
   procedure Pushbutton;
      var
         Time: longint;
   begin
   { Wait for mouse button to be pressed. }
      while not Button do { nothing }
         ;
      Delay(10, Time);
   end;
{ ********************************************************* }
begin { Body of the program. }
{ Hide all of the Macintosh Pascal windows. }
   HideAll;
{ Initialize the three rectangles. }
   Initialize_Rectangles(Rectangle);
{ Open window for viewing. }
   Open_Window(Window);
{ Establish data structure for updated region when scrolling. }
   Updated_Region := NewRgn;
{ Erase the entire screen, using the default value of the field }
{ visRgn. }
   EraseRect(Window^.visRgn^^.rgnBBox);
   Draw_Regions(Rectangle); { Draw the regions and the oval. }
{ Prompt the user to press the mouse button. }
   MoveTo(130, 40);
   DrawString(' Press mouse button to continue: ');
   Pushbutton;
{ Establish the scrolling point from the center of the oval. }
   while Button do
      begin
         GetMouse(Mousepoint);
      { Scroll only if mouse point is within the boundary of the }
      { second rectangle. }
         if PtInRect(Mousepoint, Rectangle[2]) then
            begin
               if (Mousepoint.v - 170) > 0 then
                  Vscroll := -1
               else
                  if (Mousepoint.v = 170) then
                     Vscroll := 0
                  else
                     Vscroll := 1;
               if (240 - Mousepoint.h) > 0 then
                  Hscroll := 1
               else
                  if (Mousepoint.h = 240) then
                     Hscroll := 0
                  else
                      Hscroll := -1;
               ScrollRect(Rectangle[3], Hscroll, Vscroll,
                                 Updated_Region)
            end
      end;
{ Dispose of storage for the window and Updated_Region. }
   Dispose_of_Window(Window);
   DisposeRgn(Updated_Region);
end.
```

In this program a rectangle is framed with an oval drawn inside. Using the mouse, we can position the cursor for scrolling the oval outside of the rectangle. If the cursor is outside the rectangle, no bits are shifted. Depending on the position of the cursor, we can scroll the oval upward, downward, left, right, or diagonally. As the oval is scrolled, the updated region is filled with a white background pattern. Why the three rectangles? The first and second rectangles provide the region for drawing the gray boundary enclosing the oval. The oval is drawn within the third rectangle. This third rectangle is slightly smaller than the second, so that when scrolling occurs, the frame of the second rectangle represented by an edge is not scrolled with the oval.

The second bit-transfer routine in QuickDraw is `CopyBits`. This procedure transfers a bit image between any two bit maps, clipping the result to an area specified by a masking region. The procedure `CopyBits` has the following header:

```
procedure CopyBits( Source_Bits, Destination_Bits : BitMap;
                    Source_Rect, Destination_Rect : Rect;
                    Transfer_Mode : integer;
                    Mask_Region : RgnHandle );
```

Transferring bits can be done with any one of the eight source transfer modes: `srcCopy`, `srcOr`, `srcXor`, `srcBic`, `notSrcCopy`, `notSrcOr`, `notSrcXor`, or `notSrcBic`. The result is always clipped to the region handle `Mask_Region`, and to the boundary rectangle of the destination bit map. If the `portBits` of the current `grafPort` and destination bit map are the same, the result is also clipped to the intersection of the `grafPort`'s `visRgn` and `clipRgn`. If clipping to `Mask_Region` is not desired, the pointer constant `nil` must be passed. Both the coordinates of the destination rectangle and the masking region are in terms of the `Destination_Bits.bounds` coordinate system, while the coordinates of the source rectangle are in terms of the `Source_Bits.bounds` coordinates. During transfer, source pixels can be condensed or stretched to fit the destination rectangle. Source and destination rectangles need not be of the same size.

`Bit_Copies` is a program that demonstrates this by first drawing to a window in the upper right corner of the screen, then copying the bit image to a second window, viewing the bit image from the first window in the lower left portion of the screen. As you can see, `CopyBits` can provide a simple scheme for moving the bit images of several windows about the screen.

```
program Bit_Copies(input, output);
{ Purpose:  Sample program that shows how a bit image in one }
{           window can be copied and then assigned to the bit }
{           image of another window. }
   type
      Port = GrafPtr;
      Rect_Table = array[1..2] of Rect;
   var
      Rectangle : Rect_Table;
      Window : array[1..2] of Port;
      J : integer;
{ ******************************************************** }
   procedure Open_Window (var Viewport : Port);
      begin
         new(Viewport);
         OpenPort(Viewport)
      end;
{ ******************************************************** }
   procedure Initialize_Rectangles (var Box : Rect_Table);
      begin
         SetRect(Box[1], 256, 0, 512, 171);
         SetRect(Box[2], 0, 171, 256, 342)
      end;
{ ******************************************************** }
   procedure Dispose_of_Window (var Viewport : Port);
      begin
         ClosePort(Viewport);
         Dispose(Viewport)
      end;
{ ******************************************************** }
   procedure Pushbutton;
      var
         Time: longint;
   begin
   { Wait for mouse button to be pressed. }
      while not Button do { nothing }
         ;
      Delay(10, Time);
   end;
```

```
{ ******************************************************** }
   procedure Erase_Window (Viewport : Port;  Box : Rect);
      begin
         SetPort(Viewport);
         BackPat(gray);
         EraseRect(Box)
      end;
{ ******************************************************** }
begin { Body of the main program. }
{ Hide all of the Macintosh Pascal windows. }
   HideAll;
{ Initialize the three rectangles. }
   Initialize_Rectangles(Rectangle);
   for J := 1 to 2 do  { Open window and erase screen. }
      Open_Window(Window[J]);
{ Set current port to the first window and erase the screen }
   Erase_Window(Window[1], Window[1]^.visRgn^^.rgnBBox);
{ Adjust the clipping region and draw into this first window. }
   ClipRect(Rectangle[1]);
   PenSize(3, 3);
   FillRect(Rectangle[1], white);
   FrameRect(Rectangle[1]);
   MoveTo(312, 50);
   DrawString('Window 1');
{ Prompt the user to continue. }
   MoveTo(266, 120);
   DrawString(' Press mouse button to continue: ');
   Pushbutton;
{ Set the current port to the second window; then copy the bit }
{ image of the first window to this second window. }
   SetPort(Window[2]);
   CopyBits(Window[1]^.portBits, Window[2]^.portBits,
          Rectangle[1], Rectangle[2], srcCopy, nil);
{ Now erase the contents of the first window. }
   Erase_Window(Window[1], Rectangle[1]);
   Pushbutton;
{ Now erase the contents of the second window. }
   Erase_Window(Window[2], Rectangle[2]);
   for J := 1 to 2 do { Dispose of storage for the window. }
      Dispose_of_Window(Window[J]);
end.
```

14.10 SPECIAL GRAPHICAL ENTITIES: CURSORS AND PATTERNS

Cursors and patterns are special graphical entities in QuickDraw. In Pascal a cursor is a special data type defined as follows:

```
type
   Cursor  =   record
```

```
    data: array[0..15] of integer;
    mask: array[0..15]  of integer;
    hotspot: Point
  end;
```

Each array is a 16-word data field containing the pixel representation for either the cursor or its mask. The data field contains the actual cursor image, the mask field contains information about the appearance of each pixel for the cursor, and the third field represents a point for aligning the cursor with the position of the mouse. Examples showing three different cursors representing faces is given in Figure 14.19a–c. Each element of the data and mask array is specified by a decimal `integer` value and must be in the range –32767 to 32767. We establish a cursor by laying out a 16-by-16-bit array, painting a black dot in a cell representing a black pixel (binary 1), and leaving a cell blank for a white pixel (binary 0). Then the binary value (represented in a hexadecimal format) is converted to an equivalent decimal value. If a mask is used, the following rules apply between corresponding data and masking pixels:

Data Pixel	Mask Pixel	Destination Pixel on Screen
White	Black	White
Black	Black	Black
White	White	Copy of pixel under cursor
Black	White	Inverse of pixel under cursor

Array Position	Smile	Hexadecimal	Decimal
0	XXXXXXXXXXXXXXXX	FFFF	-1
1	X--------------X	8001	-32767
2	X--------------X	8001	-32767
3	X--XX------XX--X	9819	-26599
4	X--XX------XX--X	9819	-26599
5	X--------------X	8001	-32767
6	X--------------X	8001	-32767
7	X------X-------X	8101	-32511
8	X-----XXX------X	8381	-31871
9	X------X-------X	8101	-32511
10	X--X-------X---X	9011	-28655
11	X---X-----X----X	8821	-30687
12	X----X---X-----X	8441	-31679
13	X-----XXX------X	8381	-31871
14	X--------------X	8001	-32767
15	XXXXXXXXXXXXXXXX	FFFF	-1

```
X implies a black pixel (binary 1).
- implies a white pixel (binary 0).
```

Figure 14.19a Pixel representation for the Smile cursor.

Array Position	Frown	Hexadecimal	Decimal
0	XXXXXXXXXXXXXXXX	FFFF	-1
1	X--------------X	8001	-32767
2	X--------------X	8001	-32767
3	X--XX------XX--X	9819	-26599
4	X--XX------XX--X	9819	-26599
5	X--------------X	8001	-32767
6	X--------------X	8001	-32767
7	X------X-------X	8101	-32511
8	X-----XXX------X	8381	-31871
9	X------X-------X	8101	-32511
10	X--------------X	8001	-32767
11	X-----XXX------X	8381	-31871
12	X----X---X-----X	8441	-31679
13	X---X-----X----X	8821	-30687
14	X--------------X	8001	-32767
15	XXXXXXXXXXXXXXXX	FFFF	-1

Figure 14.19b Pixel representation for the Frown cursor.

Array Position	Justso	Hexadecimal	Decimal
0	XXXXXXXXXXXXXXXX	FFFF	-1
1	X--------------X	8001	-32767
2	X--------------X	8001	-32767
3	X--XX------XX--X	9819	-26599
4	X--XX------XX--X	9819	-26599
5	X--------------X	8001	-32767
6	X--------------X	8001	-32767
7	X------X-------X	8101	-32511
8	X-----XXX------X	8381	-31871
9	X------X-------X	8101	-32511
10	X--------------X	8001	-32767
11	X--------------X	8381	-32767
12	X---XXXXXXXX---X	9FF1	-24591
13	X--------------X	8001	-32767
14	X--------------X	8001	-32767
15	XXXXXXXXXXXXXXXX	FFFF	-1

Figure 14.19c Pixel representation for the Justso cursor.

Unless assignments are made to the mask array, all elements of the array are assumed to be zero.

The mouse positions the cursor by means of a special point called the *hotspot*. Coordinates for the hotspot are relative only to the data array, the left corner having the coordinates (0, 0), and the bottom right corner having the coordinates (16, 16). The mouse uses the hotspot to align the cursor on the screen with the mouse position. In Figure 14.19 the hotspot for all three data arrays is chosen as the center point of the data array (8, 8). Figure 14.20 lists several routines for handling cursors. The cursor level is hidden from the programmer. Whenever this level is less than zero, the present cursor is hidden from view. Only when this value becomes zero is the present cursor displayed.

procedure `InitCursor;` This procedure reestablishes the current cursor as the predefined arrow pointing north-northwest and reinitializes the cursor level to 0, making the cursor visible on the screen.

procedure `SetCursor( New_Cursor : Cursor );` This procedure sets the new cursor as the current cursor. If the previous cursor was hidden, the new cursor remains hidden until it becomes uncovered by execution of `ShowCursor`. If the previous cursor was not hidden, this procedure immediately shows the new cursor.

procedure `HideCursor;` This procedure removes the present cursor from the screen by restoring the bit image below the cursor, decrementing the cursor level by 1. A call of `HideCursor` should always be balanced with a call to `ShowCursor`.

procedure `ShowCursor;` This procedure increments the cursor level, and if the value of the cursor level becomes zero (it is never incremented beyond zero), the present cursor is displayed on the screen. The present cursor is always assumed to be the cursor last set by a call to `SetCursor`.

procedure `ObscureCursor;` This procedure hides the present cursor until the mouse has been moved. This procedure has no effect on the cursor level and should never be balanced by calling `ShowCursor`.

Figure 14.20 QuickDraw procedures for handling cursors.

The following program, titled `Showing_Cursors`, demonstrates how to make one of three cursors visible on the screen. Notice that the procedure `ObscureCursor` is used to hide the system cursor until the mouse is moved:

```
program Showing_Cursors(input, output);
{ Purpose:  Sample program for showing how user-defined cursors }
{           can be created and viewed. }
   type
      Port = GrafPtr;
      Rect_Table = array[1..3] of Rect;
   var
```

```
      Rectangle : Rect_Table;
      Window : Port;
      J : integer;
      Area : array[1..3] of RgnHandle;
      Smile, Frown, Justso : Cursor;
      Mousepoint : Point;
{ ********************************************************** }
   procedure Open_Window (var Viewport : Port);
      begin
         new(Viewport);
         OpenPort(Viewport)
      end;
{ ********************************************************** }
   procedure Initialize_Rectangles (var Box : Rect_Table);
      begin
         SetRect(Box[1], 0, 0, 512, 342);
         SetRect(Box[2], 40, 40, 250, 250);
         SetRect(Box[3], 300, 120, 500, 320);
      end;
{ ********************************************************** }
   procedure Dispose_of_Window (var Viewport : Port);
      begin
         ClosePort(Viewport);
         Dispose(Viewport)
      end;
{ ********************************************************** }
   procedure Pushbutton;
      var
         Time: longint;
   begin
   { Wait for mouse button to be pressed. }
      while not Button do { nothing }
         ;
      Delay(10, Time);
   end;
{ ********************************************************** }
begin { Body of the main program. }
{ Hide all of Macintosh Pascal windows and the present cursor. }
   HideAll;
   HideCursor;
{ Initialize the data arrays for the three cursors. }
   Smile.data[0] := -1;
   Smile.data[1] := -32767;
   Smile.data[2] := -32767;
   Smile.data[3] := -26599;
   Smile.data[4] := -26599;
   Smile.data[5] := -32767;
   Smile.data[6] := -32767;
   Smile.data[7] := -32511;
   Smile.data[8] := -31871;
   Smile.data[9] := -32511;
   Smile.data[10] := -28655;
   Smile.data[11] := -30687;
   Smile.data[12] := -31679;
   Smile.data[13] := -31871;
   Smile.data[14] := -32767;
   Smile.data[15] := -1;
   Smile.hotspot.v := 8;
   Smile.hotspot.h := 8;
   Frown.data[0] := -1;
   Frown.data[1] := -32767;
   Frown.data[2] := -32767;
   Frown.data[3] := -26599;
   Frown.data[4] := -26599;
   Frown.data[5] := -32767;
   Frown.data[6] := -32767;
   Frown.data[7] := -32511;
   Frown.data[8] := -31871;
   Frown.data[9] := -32511;
   Frown.data[10] := -32767;
   Frown.data[11] := -31871;
   Frown.data[12] := -31679;
   Frown.data[13] := -30687;
   Frown.data[14] := -32767;
   Frown.data[15] := -1;
   Frown.hotspot.v := 8;
   Frown.hotspot.h := 8;
   Justso.data[0] := -1;
   Justso.data[1] := -32767;
   Justso.data[2] := -32767;
   Justso.data[3] := -26599;
   Justso.data[4] := -26599;
   Justso.data[5] := -32767;
   Justso.data[6] := -32767;
   Justso.data[7] := -32511;
   Justso.data[8] := -31871;
   Justso.data[9] := -32511;
   Justso.data[10] := -32767;
   Justso.data[11] := -32767;
   Justso.data[12] := -24591;
   Justso.data[13] := -32767;
   Justso.data[14] := -32767;
   Justso.data[15] := -1;
   Justso.hotspot.v := 8;
   Justso.hotspot.h := 8;
{ Initialize the three rectangles. }
   Initialize_Rectangles(Rectangle);
{ Establish the data structures for three regions. }
   for J := 1 to 3 do
      Area[J] := NewRgn;
{ Open window for viewing the complete screen. }
   Open_Window(Window);
   PenSize(2, 2);
{ Establish each of the three regions. }
   begin { first region }
      OpenRgn;
      FrameRect(Rectangle[1]);
      CloseRgn(Area[1]);
      FillRgn(Area[1], white);
   end;
   begin { second region }
      OpenRgn;
      FrameRoundRect(Rectangle[2], 90, 90);
      CloseRgn(Area[2]);
      FrameRgn(Area[2]);
      MoveTo(100, 200);
      DrawString('Happy Region');
   end;
   begin { third region }
      OpenRgn;
      FrameOval(Rectangle[3]);
      CloseRgn(Area[3]);
      FrameRgn(Area[3]);
      MoveTo(360, 280);
      DrawString('Sad Region');
    end;
{ Prompt the user to continue. }
   MoveTo(266, 50);
   DrawString(' Press mouse button to stop: ');
{ Establish a new cursor, and then obscure the new cursor until }
{ the mouse is moved. }
   SetCursor(Justso);
   ShowCursor;
   ObscureCursor;
   while not button do
      begin
         GetMouse(Mousepoint.h, Mousepoint.v);
         if PtInRgn(Mousepoint, Area[2]) then
            SetCursor(Smile)
         else
            if PtInRgn(Mousepoint, Area[3]) then
               SetCursor(Frown)
            else
               SetCursor(Justso);
      end;
{ Erase the complete screen with a gray background. }
   BackPat(gray);
   EraseRgn(Area[1]);
{ Dispose of storage for the window and the areas. }
   Dispose_of_Window(Window);
   for J := 1 to 3 do
      DisposeRgn(Area[J]);
end.
```

This is followed by immediately setting the cursor to the data pattern `Justso`. While the mouse button is not pressed, and depending on the position of the mouse, the procedure `SetCursor` is called to change the cursor. The Pascal system returns to using the normal cursor when the program ends execution. Figure 14.21 shows the screen with the Smile cursor.

Although the Macintosh has numerous patterns that can be made available, QuickDraw allows us to define our own patterns using a special data type in Pascal called `Pattern`. This type is defined as follows:

```
type
    Pattern = packed array [0..7] of 0..255;
```

This represents a rectangle composed of eight elements, each element storing an 8-bit row composed of integers in the range 0 through 255. For example, let us change the background pattern in the last example by adding the variable `Mypattern: Pattern`, under the variable declarations and replacing the statement `BackPat(gray)` with the following two statements:

```
StuffHex(@Mypattern, '3C66C30000C3663C');
BackPat(Mypattern);
```

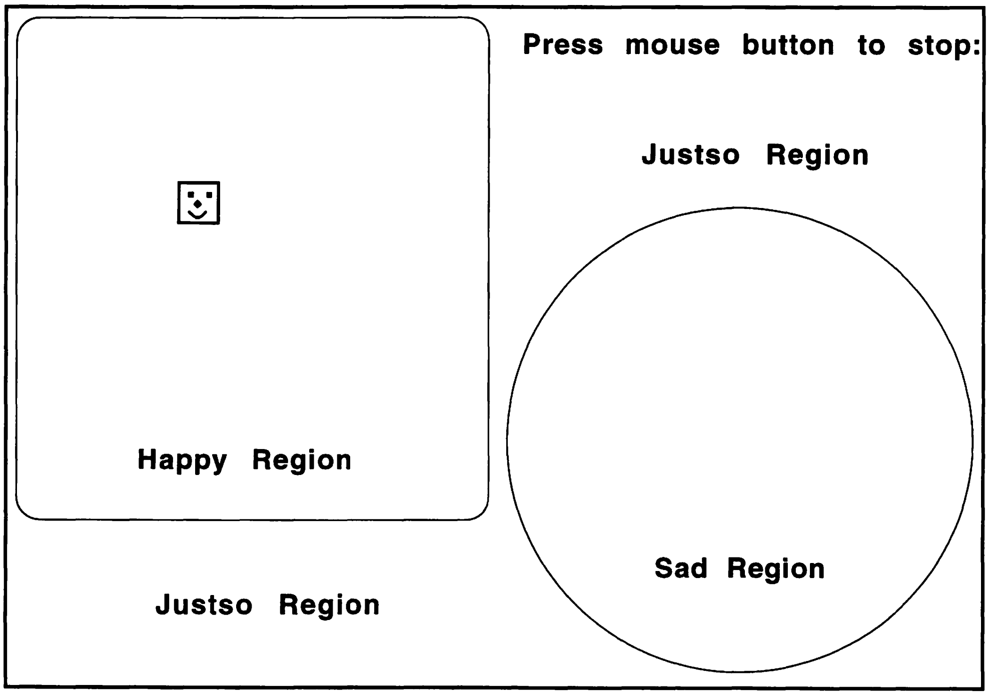


Figure 14.21 An example that applies cursor routines, displaying three different cursors.

The procedure StuffHex takes the hexadecimal pattern on the right and assigns its equivalent numeric value to the elements of the pattern array. The alternative is to draw an 8-by-8 array for representing a pattern, placing into each cell a 1 for a black pixel and a 0 for a white pixel. Each row of the array is then converted from 8-bit binary value to an unsigned integer. Each row requires eight Pascal statements for assigning array elements to a pattern.

14.11 MAPPING AND SCALING POINTS, RECTANGLES, REGIONS, AND POLYGONS

Before ending this chapter, let us examine some of the routines for offsetting, mapping, and rescaling objects and shapes. A list of these is given in Figure 14.22. They are divided into three general categories: conversion between coordinate systems, changing boundaries of objects, and mapping and scaling.

Conversion between Coordinate Systems

procedure LocalToGlobal(**var** Local_Pt : Point); This procedure translates the given point from the local coordinates of the current grafPort into a global coordinate system having an origin at the top left point (0, 0) of the screen. This procedure has no effect on the screen.

procedure GlobalToLocal(**var** Global_Pt : Point); This procedure translates the given point from the global coordinates of the screen into the local coordinates of the current grafPort. This procedure has no effect on screen.

Changing Boundaries of Shapes and Regions

procedure OffsetRect(Rectangle : Rect; Delta_h, Delta_v : integer); This procedure translates the boundary of the given rectangle through a horizontal distance Delta_h and a vertical distance Delta_v in the coordinate system of the current grafPort. Movement is to the right and downward if both Delta_h and Delta_v are positive. For negative values, movement of the rectangle is in the opposite direction. The given rectangle retains its shape and size. The screen remains unchanged, unless a routine is called on to draw within the translated rectangle.

procedure OffsetRgn(Source_Region : RgnHandle; Delta_h, Delta_v : integer); This procedure translates the boundary of the given region through a horizontal distance Delta_h and a vertical distance Delta_v in the coordinate system of the current grafPort. Movement is to the right and downward if both Delta_h and Delta_v are positive. For negative values, movement of the region is in the opposite direction. The given region retains its shape and size. The screen remains unchanged unless a routine is called on to draw within the translated region.

procedure `OffsetPoly(Source_Polygon : PolyHandle; Delta_h, Delta_v : integer);` This procedure translates the boundary of the given polygon through a horizontal distance `Delta_h` and a vertical distance `Delta_v` in the coordinate system of the current `grafPort`. Movement is to the right and downward if both `Delta_h` and `Delta_v` are positive. For negative values, movement of the polygon is in the opposite direction. The given polygon retains its shape and size. The screen remains unchanged, unless a routine is called on to draw within the translated polygon.

procedure `InsetRect(` **var** `Rectangle : Rect; Delta_h, Delta_v : integer );` This procedure rescales the boundary of the given rectangle with respect to the its center by moving its vertical edges through a distance `Delta_h` and its horizontal edges through a distance `Delta_v`. The boundary of the rectangle shrinks for positive values of `Delta_h` and `Delta_v` and expands for negative values. The screen remains unchanged, unless a routine is called on to draw within the rescaled rectangle.

procedure `InsetRgn( Source_Region : RgnHandle; Delta_h, Delta_v : integer );` This procedure rescales the boundary of the given region with respect to its center by moving its boundary coordinates through a horizontal distance `Delta_h` and a vertical distance `Delta_v`. The boundary of the region shrinks for positive values of `Delta_h` and `Delta_v` and expands for negative values. The screen remains unchanged unless a routine is called on to draw within the rescaled region.

Scaling and Mapping

procedure `ScalePt(` **var** `Specified_Pt : Point; Source_Rect, Dest_Rect : Rect );` This procedure will scale a specified point in the source rectangle to the destination rectangle by a ratio of the two rectangles.

procedure `MapPt(` **var** `Specified_Pt : Point ; Source_Rect, Dest_Rect : Rect );` This procedure both translates and scales the source point within the source rectangle to a point within the destination rectangle. The effect of the procedure can only be observed by calling on routines that draw with the specified point.

procedure `MapRect(` **var** `Rectangle : Rect; Source_Rect, Dest_Rect : Rect );` This procedure both translates and scales the specified rectangle within the source rectangle to a rectangle within the destination rectangle. The effect of this procedure can only be observed by calling on routines that draw with respect to the specified rectangle.

procedure `MapRgn( Specified_Region : RgnHandle; Sourec_Rect, Dest_Rect : integer);` This procedure both translates and scales the specified region within the source rectangle to a region within the destination rectangle. The effect of this procedure can only be observed by calling on routines that draw with respect to the specified region.

procedure `MapPoly( Specified_Polygon : PolyHandle ; Source_Rect , Dest_Rect: Rect );` This procedure both translates and scales the specified polygon within the source rectangle to a polygon within the destination rectangle. The effect of this procedure can only be observed by calling on routines that draw with respect to the specified polygon.

Figure 14.22 QuickDraw procedures for offsetting, mapping, and rescaling objects and shapes.

We can convert between coordinate systems with the procedures `LocalToGlobal` and `GlobalToLocal`. For example, assume the existence of an oval drawn within a rectangle called `Oval_Rect` of `grafPort` `Window_1`. What is required is a set of Pascal statements that map this oval from the local coordinates of `Window_1` into the local coordinates of `Window_2` while the local coordinates for each `grafPort` are not identical. The following statements show these steps:

```
SetPort(Window_1);
LocalToGlobal( Oval_Rect.left_top );
LocalToGlobal( Oval_Rect.right_bottom );
SetPort( Window_2 );
GlobalToLocal( Oval_Rect.left_top );
GlobalToLocal( Oval_Rect.right_bottom );
FrameOval( Oval_Rect );
```

The first `grafPort` is set as the current `grafPort`. Since a rectangle is defined in terms of two corner points, top left and bottom right, these two points are copied into global coordinates. Setting the current `grafPort` to `Window_2` maps the two global points to the local coordinates of this second port, and the rectangle `Oval_Rect` is now defined in terms of the local coordinates of the second port. Although these steps allow us to translate an object from one local coordinate system to another, they do not allow us to scale the translated object.

The routines `OffsetRect`, `OffsetRgn`, `OffsetPoly`, `InsetRect`, and `InsetRgn` are useful for translating the boundary of a rectangle, region, or polygon before drawing within that boundary. In the case of the `Inset` routines, they can be used to rescale the boundaries of either a rectangle or a region before drawing within the boundaries. These routines have no effect on points, lines, and shapes previously drawn within the boundaries of these objects.

The mapping procedures `MapPt`, `MapRect`, `MapRgn`, and `MapPoly` affect only drawing with the specified objects such as points, rectangles, regions, and polygons. They are used for both translating and scaling boundaries of objects such as rectangles, regions, and polygons, but not for the points directly within these boundaries.

Consider the following Pascal statements. Figure 14.23 shows the results of executing them.

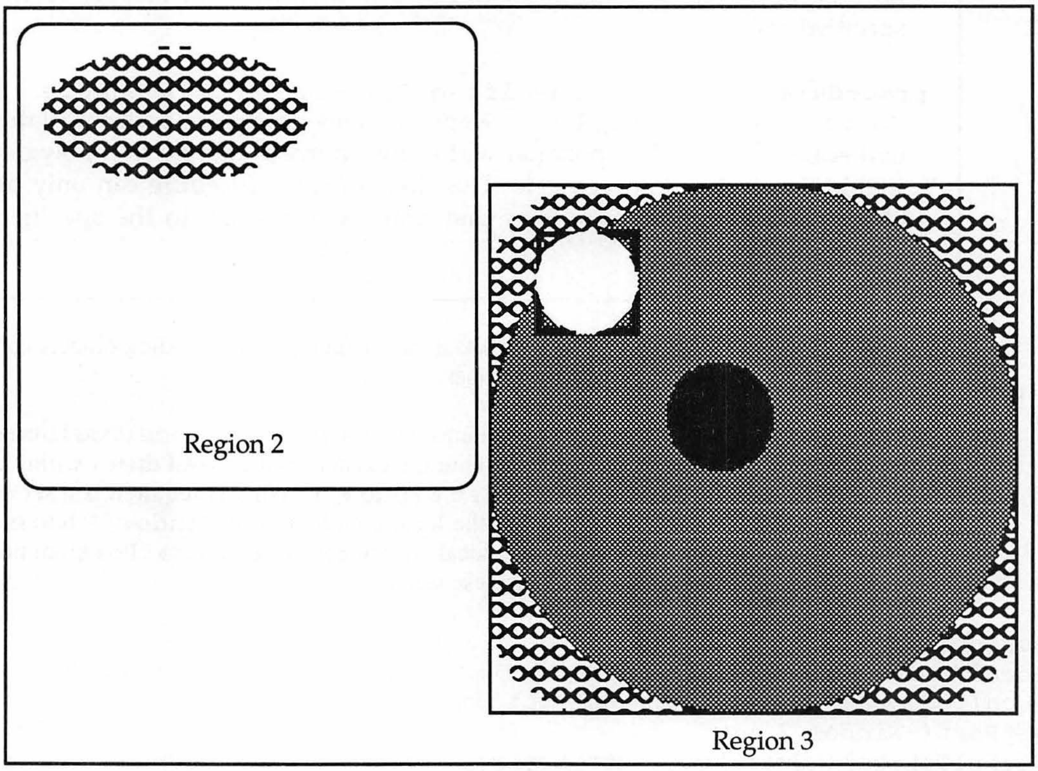


Figure 14.23 Applying the procedures MapRgn and MapRect to map rectangles and ovals from Region2 into Region3.

```
begin
{ Establish boundaries for rectangles. }
   SetRect( Rectangle[2], 40, 40, 250, 250 );
   SetRect( Rectangle[3], 300, 120, 500, 320 );
   SetRect( Rectangle[4], 60, 60, 100, 100 );
   SetRect( Rectangle[5], 50, 50, 150, 100 );
{ Make a copy of the second rectangle. }
   Temp_Rect := Rectangle[2];
{ Map from region 2 in rectangle 2 to a region in rectangle 3. }
   MapRgn( Area[2], Rectangle[2], Rectangle[3]);
   FillRgn( Area[2], Mypattern );
{ Map from within rectangle 2 to rectangle 3, assigning  results }
{  to rectangle 2. }
   MapRect( Rectangle[2], Rectangle[2], Rectangle[3] );
   FrameRect( Rectangle[2] );
   FillOval( Rectangle[2], gray );
{ Map from within the temporary rectangle to third rectangle, }
{ assigning the result to rectangle 4. }
   MapRect( Rectangle[4], Temp_Rect, Rectangle[3] );
   FrameRect(Rectangle[4]);
   FillOval(Rectangle[4], white);
   OffsetRect( Rectangle[4], 50, 50 );
   FillOval( Rectangle[4], black );
{ Map region 3 from within rectangle 3 to rectangle 5. }
   MapRgn( Area[3], Rectangle[3], Rectangle[5] );
   FillRgn( Area[3], Mypattern );
end;
```

What do these statements mean? First, we make a copy of the second rectangle, because once the MapRect procedure is executed, we are no longer looking at the original corner points of this second rectangle. Second, the procedure MapRgn establishes a mapping relationship from region 2 in the second rectangle to a region located within the third rectangle. When the FillRgn routine is called, an oval filled with gray is drawn in the third rectangle, located in region 3. The MapRect procedure that follows now establishes a mapping relationship of the second rectangle to the third rectangle. Only rectangle-drawing routines (such as FrameRect and FillOval) involving the second rectangle can affect the screen. The MapRect procedure that follows establishes a mapping relationship between the fourth rectangle located within the temporary rectangle and a region within the third rectangle. Drawing routines for rectangles specifying the fourth rectangle are viewed as being performed in the third region of the screen. The last MapRgn call establishes the mapping relationship between region 3 within the third rectangle and a region within the fifth rectangle. A call to FillRgn results in an oval being drawn in the second region.

Keep in mind that these routines only translate and scale the boundaries of the basic object being mapped. Lines and text drawn within the region to be mapped are not mapped. The routine CopyBits provides the means for both mapping and scaling all of the shapes, points, and text located within a source rectangle to a destination rectangle in another region.

SUMMARY

Both Macintosh and THINK Pascal provide library procedures for drawing to the Drawing window and also give us access to the QuickDraw library of the Macintosh computer. This special ROM-based library has routines for drawing lines, rectangles, polygons, regions, arcs, wedges, pictures, and text.

By opening and setting grafPorts, we can draw several windows on the screen, one or more of them overlaid. By using the routines to define regions, the programmer can establish clipping regions that go beyond a simple rectangle. The bit-transfer routine for scrolling bits gives us the ability to scroll one window while leaving other windows unaffected. The routine Copy_Bits allows us to translate and scale all of the bits from one region to another region.

The picture routines of QuickDraw allow us to define and save one or more drawings. These saved pictures can later be drawn within a picture frame of a grafPort, with the picture being scaled to the size of a rectangle represented by the picture frame. By using the text routines, we can select different fonts and text styles for drawing text within a `grafPort`. Keep in mind that the routine `PenSize` has no effect on the height or width in drawing text characters. QuickDraw gives the programmer an opportunity to use imagination in drawing to the screen.

Care must be taken when calling on QuickDraw routines. Being careless can cause Macintosh or THINK Pascal to crash and could also damage the binary information stored in sectors on the disk.

REVIEW QUESTIONS

1. What is the function of a computer library?
2. In Macintosh/THINK Pascal the QuickDraw library is divided into what two libraries?
3. What is the difference between RAM and ROM?
4. Where are QuickDraw libraries stored in the Macintosh computer?
5. What QuickDraw routines used by Macintosh Pascal are different in THINK Pascal?
6. Distinguish between a point, a rectangle, a region, and a coordinate plane.
7. How is the data type `Point` defined in Macintosh/THINK Pascal?
8. How is the data type `Rect` defined in Macintosh/THINK Pascal?
9. How is the data type `Region` defined in Macintosh/THINK Pascal?
10. What does a `grafPort` represent?
11. How is a `grafPort` defined in Macintosh/THINK Pascal?
12. In looking at a `grafPort` record, what function is served by the field `portBits`? by the field `portRect`? by the field `visRgn`? by the field `clipRgn`?
13. What fields are related to properties involving the drawing pen?
14. What fields are related to properties involving the drawing of text?
15. Through the intersection of what rectangles and boundaries will drawing always take place?
16. What must be executed before the command `OpenPort` is executed?
17. What are the differences between the commands `OpenPort` and `InitPort`?
18. What are the differences between the routines `GetPort` and `SetPort`?
19. What are the differences between the commands `SetOrigin` and `MovePortTo`?
20. What is the purpose of a clipping region?
21. Which is true: Nothing can be *drawn* outside a clipping region or nothing can be *observed* outside a clipping region?
22. What routines can be used for setting and changing the clipping regions?
23. List the procedures that can affect the drawing pen.
24. When describing the characteristics of the drawing pen, what are the normal values for the pen?
25. If you assign new values to the fields describing the drawing pen, is it true that the characteristics for drawing text will also change? Explain your answer.
26. For Macintosh/THINK Pascal, are the Program and Drawing windows the same `grafPort`? Can you think of a way to check your answer?
27. What are the differences between procedure `SetRect` and `SectRect`?

28. Name the function for testing the existence of a pixel within a specified rectangle.
29. A rectangle is considered empty if the bottom coordinate is equal to or less than the top, or when the right coordinate is equal to or less than the left. What function can test for the existence of an empty rectangle?
30. Do any of the routines described in Figure 14.7 result in drawing to a `grafPort`?
31. If the Macintosh/THINK Pascal routine `HideAll` did not exist, how could windows appearing on the screen be hidden?
32. The routines in Figure 14.8 all require a parameter of type `Rect`. What routine can be used to establish a rectangle before execution of any of these routines?
33. For the routines listed in Figure 14.8, what are the basic differences between the `Rect` routines and the `Oval` routines?
34. What are the purposes of the `RoundRect` routines listed in Figure 14.8?
35. When using `Arc` and `Wedge` routines, what are the differences in the views of angles that are parameters of routines in Figure 14.9 and those that are angles of Pascal trigonometric functions?
36. If you are drawing an arc, what routine can you use to fill the arc with a background pattern?
37. Instead of using the `WriteDraw` command, what command from the QuickDraw library can we use to paint text to a `grafPort` window?
38. If the `StringOf` function did not exist, how could we perform the equivalent actions by using an external file?
39. How can the function `StringWidth` be used for painting text to a `grafPort`?
40. Why are the text-drawing routines given in Figure 14.14 different from the Macintosh/THINK Pascal library routine `WriteDraw`?
41. When drawing characters to a `grafPort`, how can we change the style of the characters?
42. Is it true that all characters will have the same pixel width and pixel height when drawn to a `grafPort`? Can you think of a way to test your answer?
43. Do the routines in Figure 14.14 have any effect on the characters displayed in the Text window? in the Drawing window?
44. What field of a `grafPort` record is used to establish the text size?
45. What effect does the field `pnLoc` of the `grafPort` record have on the text drawing routines?
46. Define the term *region* as it applies to the QuickDraw library.
47. What is represented by the data type `RgnHandle`?
48. What will happen if you execute a region routine specifying a region's handle before executing the command `NewRgn`?
49. What is the purpose of the routines `OpenRgn` and `CloseRgn`? What differences exist between the commands `NewRgn` and `OpenRgn`?
50. What is meant by the term *polygon* ?
51. What data type can we use to store the representation of a polygon?
52. What is the purpose of procedure `OpenPoly`?
53. What are the differences between the procedures `KillPoly` and `ClosePoly` listed in Figure 14.16?

54. Briefly explain the concept of a picture as it applies to the QuickDraw library.
55. What data types are required in QuickDraw for storing a picture?
56. What QuickDraw command is necessary for drawing a picture?
57. What are the purposes of transfer modes and bit-transfer operations?
58. What are the differences between the two bit-transfer routines `ScrollRect` and `CopyBits`?
59. How can the routine `CopyBits` be used to erase the source bits within a `grafPort`?
60. Explain how to create the format for a new cursor?
61. What are the differences between the routines `HideCursor` and `ObscureCursor`?
62. How can a previous cursor be reinstated to a `grafPort`?
63. What is the purpose of the procedure `StuffHex`?
64. What purposes are served by the routines listed in Figure 14.22?
65. What routines are useful for scaling a `grafPort`? What routines are useful for translating the boundary of a `grafPort`?

PROGRAMMING EXERCISES

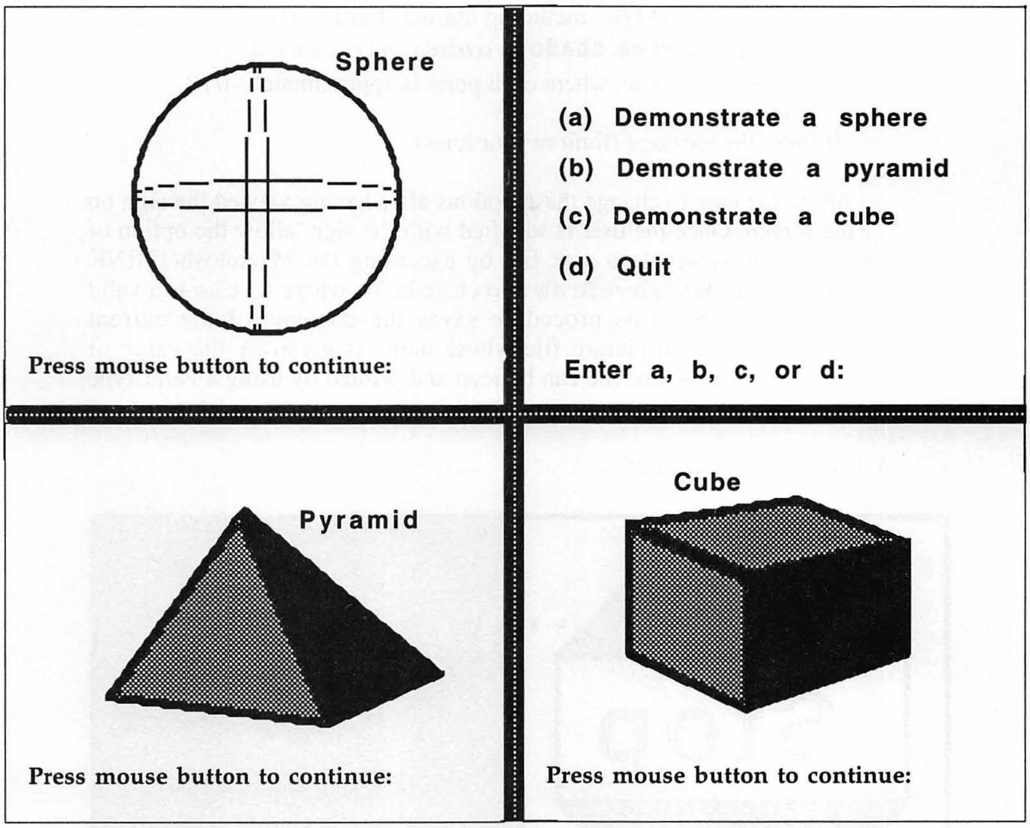
Although not all programming exercises require you to write an algorithm, you may better understand the problem and what is required if you first write an algorithm and then trace it by hand with several examples before you write the Pascal program.

1. Complete the procedures for the programming example `Drawing_Pictures`, and test the program to show that it draws a pie chart.

2. Write a program that divides the screen into four separate windows, with each window representing a `grafPort`. In the upper right window have your program prompt the user to enter one of four possible choices:

   (a) Demonstrate a sphere (upper left window).
   (b) Demonstrate a pyramid (lower left window).
   (c) Demonstrate a cube (lower right window).
   (d) Quit.

   When a character is entered, the present window is to be cleared, the new window opened, and the three-dimensional figure drawn. Upon completion of the figure, the user is prompted with the following instruction: `Press the mouse button to continue.` Pressing the mouse button clears the the present window and makes the upper right window the active window, and presents the user with the menu for choosing one of the four possible choices. Figure 14.24 demonstrates the format for the screen. This exercise requires using the following QuickDraw routines: `OpenPort`, `ClosePort`, `SetPort`, `SetClip`, `TextFace`, `TextSize`, `DrawString`, `SetRect`, `FrameRect`, `PaintRect`, `FillRect`, `FrameOval`, `NewRgn`, `DisposeRgn`, `OpenRgn`, `CloseRgn`, `FillRgn`, `FrameRgn`.



```
Press mouse button to continue:

(a) Demonstrate a sphere
(b) Demonstrate a pyramid
(c) Demonstrate a cube
(d) Quit

Enter a, b, c, or d:

Press mouse button to continue:

Press mouse button to continue:
```

Figure 14.24

3. Modify the program in Exercise 2 by storing each window as a separate picture. Upon choosing an option, have the proper picture drawn to the proper window. This exercise will require the following additional QuickDraw routines: `OpenPicture`, `ClosePicture`, `DrawPicture`, and `KillPicture`. *Hint* : Develop and test one picture at a time.

4. The Macintosh Pascal disk comes with the following fonts and font numbers:

Chicago (System Font)	0
Geneva	1, 3
Monaco	4
New York	2
Venice	5

Write a program that allows the user to make simple signs, given the following options:

(a) Choice of a font, with an example of each font.
(b) Choice of text style, including normal, **bold,** *italic*, underline, outline, shadow, condense, and e x t e n d.
(c) Choice of text size, where each point is approximately 1/72 of an inch.
(d) Enter the message (limit of four lines).

Allow the user to change these options after having viewed the sign on the screen. Once the user is satisfied with the sign, allow the option of saving the screen to a disk file by executing the Macintosh/THINK Pascal procedure `SaveDrawing(title)`, where `title` is a valid `string` type. This procedure saves the contents of the current `grafPort` to a picture file whose name is given by the value of `title`. This picture file can be read and printed by using a Paint-type graphics application.

5. Consider Figure 14.25.

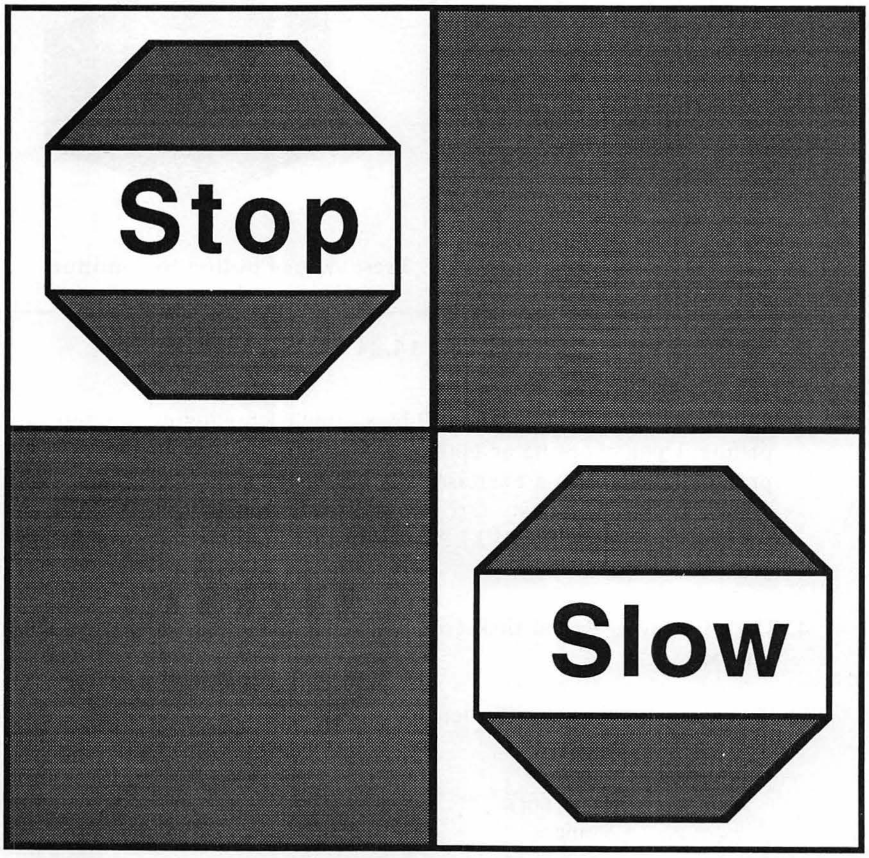


Figure 14.25

Write a program that will require two grafPorts: one for displaying the sign `STOP` and one for displaying the sign `SLOW`. The remainder of the screen must have a gray background. Use the font, font size, and style for displaying the text of each sign.

6. Consider the globe shown in Figure 14.26. Write a program that will draw both the parallel lines representing latitude and the ovals representing the meridians of longitude.

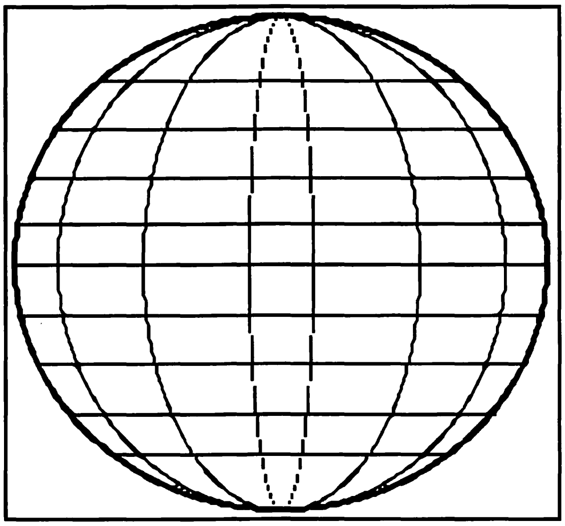


Figure 14.26

7. Write a program using `grafPort` and text routines that perform special printing (sometimes called *pretty printing*) of expressions given by a one-dimensional format. For example, consider the following expressions:

Standard Output	**Special Print**
`A * B**C + 42`	$A*B^C+42$
`A[k] - 42 / A`	$A_k \ -42/A$
`B[i,j]**5 / A[1,1] + N`	$B_{i,j}^{\ 5}/A_{1,1}+N$

8. Modify the programming example from Section 14.9 titled `Scrolling` so that it can be used as a simple test of skill by employing the mouse. Initially the person using the program must see the message `TRY TO FIND THE CENTER OF THE SQUARE` displayed in the upper left corner of the screen. As the mouse is moved and as the cursor moves across the screen, when it comes close to the center of the square, have the program beep once and display the

following message at the upper right of the screen: `YOU ARE GETTING CLOSE`. If the distance increases beyond a limit, this message must be erased. You will have to decide on this limit and how you will test for this distance. If the person locates the center of the square, have the program respond by beeping twice and by displaying the following message at the bottom of the square: `YOU ARE ON TARGET`. If at any time the person moves the mouse away from the center of the square, the message that the user is on target must be erased.

9. Consider the rule for rotating a point Q about an arbitrary point C, as shown in Figure 14.27. The new point (x_2, y_2) is given by the following equations:

$$y_2 = (y_1 - y_c)\cos\emptyset - (x_1 - x_c)\sin\emptyset + y_c$$

$$x_2 = (y_1 - y_c)\sin\emptyset + (x_1 - x_c)\cos\emptyset + x_c$$

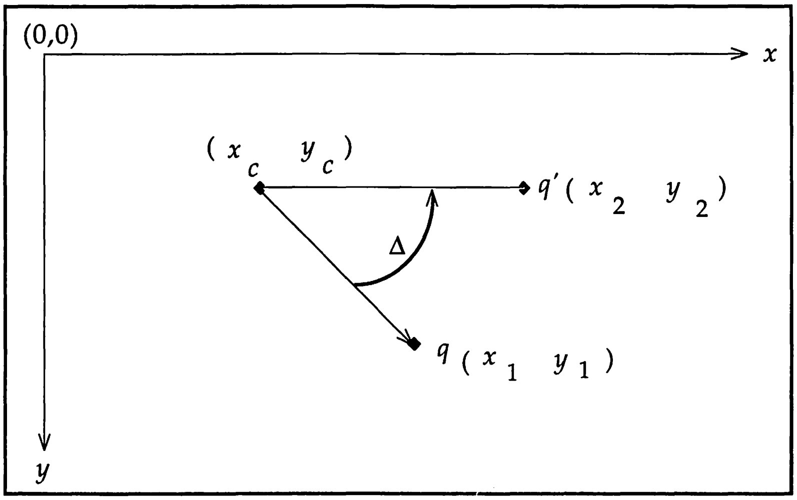


Figure 14.27

Using this principle, write a program that will rotate a rectangle about the center point (x_c, y_c) through an angle Ø. Can you think of a way to define a procedure that will accept three parameters: a point representing (x_c, y_c), angle of rotation, and an object represented by a rectangle?

10. Write a program that will draw bar charts like those shown in Figure 14.28. This program must prompt the user for the following actions:

(a) Title the bar chart.
(b) Add labels for vertical and horizontal axes.
(c) Scale the horizontal axis (maximum and minimum).

(d) Display the bar chart to a window.
(e) Save the bar chart to a picture file.
(f) Exit.

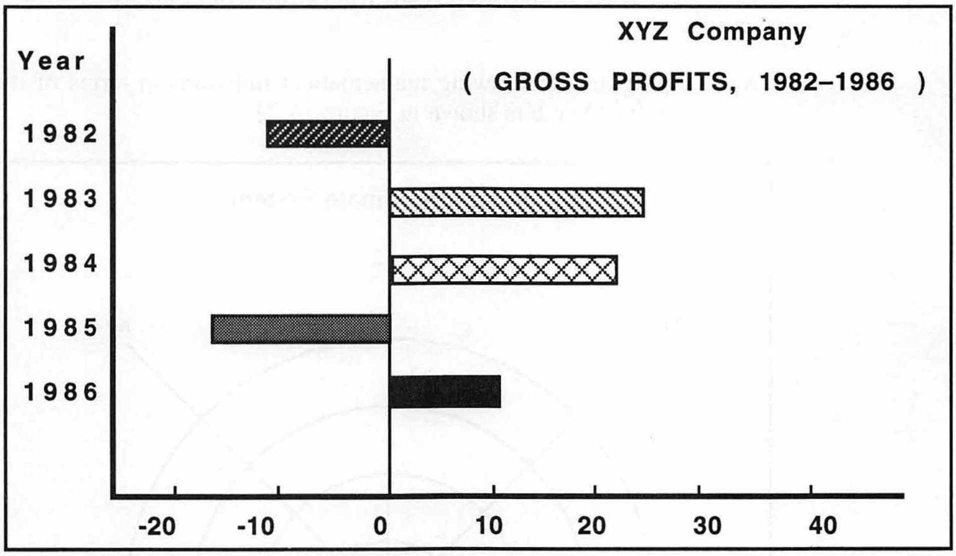


Figure 14.28

11. Using the picture routines of the QuickDraw library, write a program that will offer the user the options of drawing a bar chart using the procedures from Exercise 10 or a pie chart using the procedures in the program `Draw_Piechart`. Can you think of a way to allow the user to modify the text fonts and text style when labeling and titling the drawings?

12. When we use the QuickDraw Library, the routine `CopyBits` provides a means of both mapping and scaling all of the shapes, points, and text located within a source rectangle to a destination rectangle in another region. Can you add an option to Exercise 10 for copying the bit pattern on the screen and moving it to a region in another `grafPort`?

13. Using the region routines of the QuickDraw Library, write a program that will serve as a tutorial for teaching the basic principles of set theory demonstrated through Venn diagrams. In the execution of this program a set is represented as an oval shaded with a background pattern. The following options are offered to the user:

(a) Demonstration of *A* **union** *B*
(b) Demonstration of *A* **intersection** *B*
(c) Demonstration of set difference *A* – *B*
(d) Demonstration of a complement of a set *A*
(e) Exit

Regions representing the sets *A* and *B* should be shown before execution of a demonstration and also after execution. Shading must be used to show the region that results from execution of the demonstrated option.

14. Write a program for drawing mathematical functions in terms of the polar coordinate system shown in Figure 14.29.

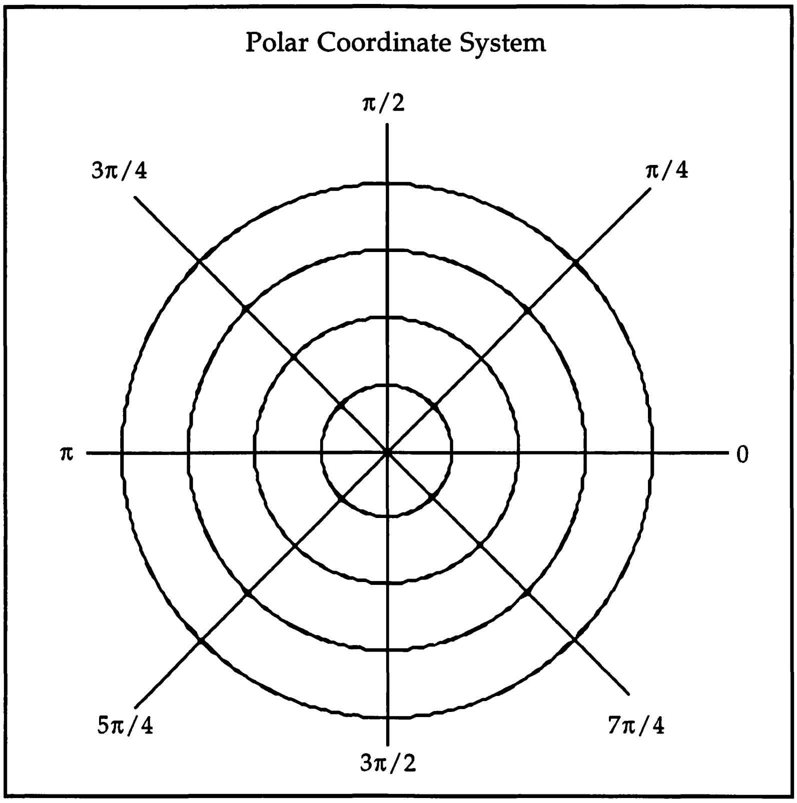


Figure 14.29

Notice that the direction of the angles is opposite to that of Macintosh Pascal as well as the Macintosh drawing plane, being in terms of an *xy* plane instead of in terms of a radius and angle. Try some of the following examples:

```
R = A * cos( N * Ø )                    { N-leaved rose }
R = 2 - cos ( Ø )                       { limacon }
R = 2^Ø   or   4 * R = Ø                { spiral }
R = e^2 * Ø                             { logarithmic spiral }
R = 2 + 2 * sec( Ø )                    { conchoid }
R = A * ( 1 + cos( Ø ) )                { cardioid }
```

Hint : Convert the point given by `(R,Ø)` into one given by `(X, Y)`; then draw a line from a previous point to a new computed point. Use small increments for angle Ø to obtain smooth lines.

15. Write a program that can display five or six windows on the Macintosh screen, as shown in Figure 14.30.

    The small box in the upper left corner of an active window will always be shaded. To make an inactive window active, move the mouse cursor to an area of the inactive window and press the mouse button. This will redraw the inactive window and make it active. When an active window becomes inactive, the small box in the upper left corner of the window must remain unshaded. For deleting an active window, move the cursor to the small shaded box and press the mouse button. This must delete the present active window as a shaded rectangular area on the screen, leaving the last window created before the deleted window as the active window.

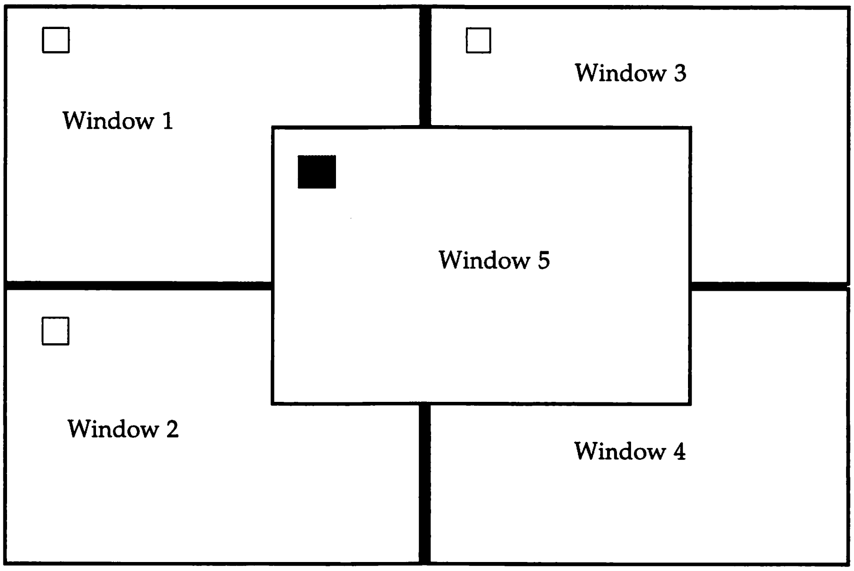


Figure 14.30

Appendix A

THINK and Macintosh Pascal Reserved Words

MACINTOSH PASCAL

The following are reserved words used by Macintosh Pascal. Use these words only in the context in which they are defined:

and	else	label	procedure	type
array	end	mod	program	until
begin	file	nil	record	uses
case	for	not	repeat	var
const	function	of	set	while
div	goto	or	string	with
do	if	otherwise	then	
downto	in	packed	to	

Any reserved word may be written in either uppercase or lowercase letters. The two forms are equivalent symbols in the context of translation. The following single characters are special symbols in Macintosh Pascal:

+ - * / = < > [ ] . , ( ) : ; ^ @ { } $

The following character pairs represent special characters:

<> <= >= := .. (* *) (. .)

The characters (* and *) represent { and }, respectively, and the characters (. and .) are always be displayed as [and] , respectively.

THINK PASCAL

The following are reserved words used by Lightspeed Pascal. The THINK Pascal user's manual refers to them as *word-symbols.*

and	end	interface	packed	type
array	file	label	procedure	unit
begin	for	mod	program	univ
case	function	nil	record	until
const	goto	not	repeat	users
div	if	object	set	var
do	implementation	of	string	while
downto	in	or	then	with
else	inherited	otherwise	to	

Any reserved word may be written in either uppercase or lowercase letters. The two forms are equivalent symbols in the context of translation. The following single characters are special symbols in Macintosh Pascal:

+ - * / = < > [ ] . , ( ) : ; ^ @ { } $

The following character pairs represent special characters:

<> <= >= := .. (* *) (. .)

The characters (* and *) represent { and }, respectively, and the characters (. and .) are always be displayed as [and] , respectively.

Appendix B

Macintosh Character Set

Figure B.1 shows the Macintosh character set. The character set varies by style and selection of font (Times 12 is shown). For example, the Geneva 12 character in the hex 9D position is a rabbit. This character is not available in most other fonts.

Notice that the row and column headings of our table are hexadecimal digits. To determine the equivalent decimal code value, take the column hexadecimal digit and multiply it by 16. Then add the value of the row digit. For example, the character π has the hexadecimal value B9 (column B, row 9). The equivalent decimal value is (11 * 16) + 9 = 185. Remember that the hexadecimal digits A, B, C, D, E, and F have the decimal values 10, 11, 12, 13, 14, and 15, respectively. The first 32 characters (00 through 1F) and the last 38 characters (DA through FF) are nonprinting characters and are often replaced by a small square bracket or box or the character ¸ shown in Figure B.1. Your choice of font set will determine the optional characters that are found in hex locations 80 (column 8, row 0) through D8 (column D, row 8).

Nonprinting characters are shown by ¸ or a blank. The character set found in hex locations 21 (column 2, row 1) through 7F (column 7, row F) and accessible from the keyboard is used for standard text and programming. It contains one blank character, 7F, which is the delete character. The character at hex location 00 is the null character; a character of no length. All characters between hex locations 00 and 21 are nonprinting.

The decimal numbers associated with the characters are called their ASCII values. ASCII is the acronym for American Standard Code for Information Interchange.

The Macintosh character set: Times font

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0		¸		0	@	P	`	p	Ä	ê	†	∞	¿	–	¸	¸
1	¸	¸	!	1	A	Q	a	q	Å	ë	°	±	¡	—	¸	¸
2		¸	"	2	B	R	b	r	Ç	í	¢	≤	¬	“	¸	¸
3		¸	#	3	C	S	c	s	É	ì	£	≥	√	”	¸	¸
4		¸	$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘	¸	¸
5		¸	%	5	E	U	e	u	Ö	ï	•	µ	≈	’	¸	¸
6	¸	¸	&	6	F	V	f	v	Ü	ñ	¶	∂	∆	÷	¸	¸
7	¸	¸	'	7	G	W	g	w	á	ó	ß	∑	«	◊	¸	¸
8	¸	¸	(	8	H	X	h	x	à	ò	®	∏	»	ÿ	¸	¸
9		¸	)	9	I	Y	i	y	â	ô	©	π	…	Ÿ	¸	¸
A	¸	¸	*	:	J	Z	j	z	ä	ö	™	∫		¸	¸	¸
B	¸	¸	+	;	K	[	k	{	ã	õ	´	ª	À	¸	¸	¸
C	¸	¸	,	<	L	\	l	|	å	ú	¨	º	Ã	¸	¸	¸
D		¸	-	=	M	]	m	}	ç	ù	≠	Ω	Õ	¸	¸	¸
E	¸	¸	.	>	N	^	n	~	é	û	Æ	æ	Œ	¸	¸	¸
F	¸	¸	/	?	O	_	o		è	ü	Ø	ø	œ	¸	¸	¸

Figure B.1 The Macintosh Character Set: Times 12

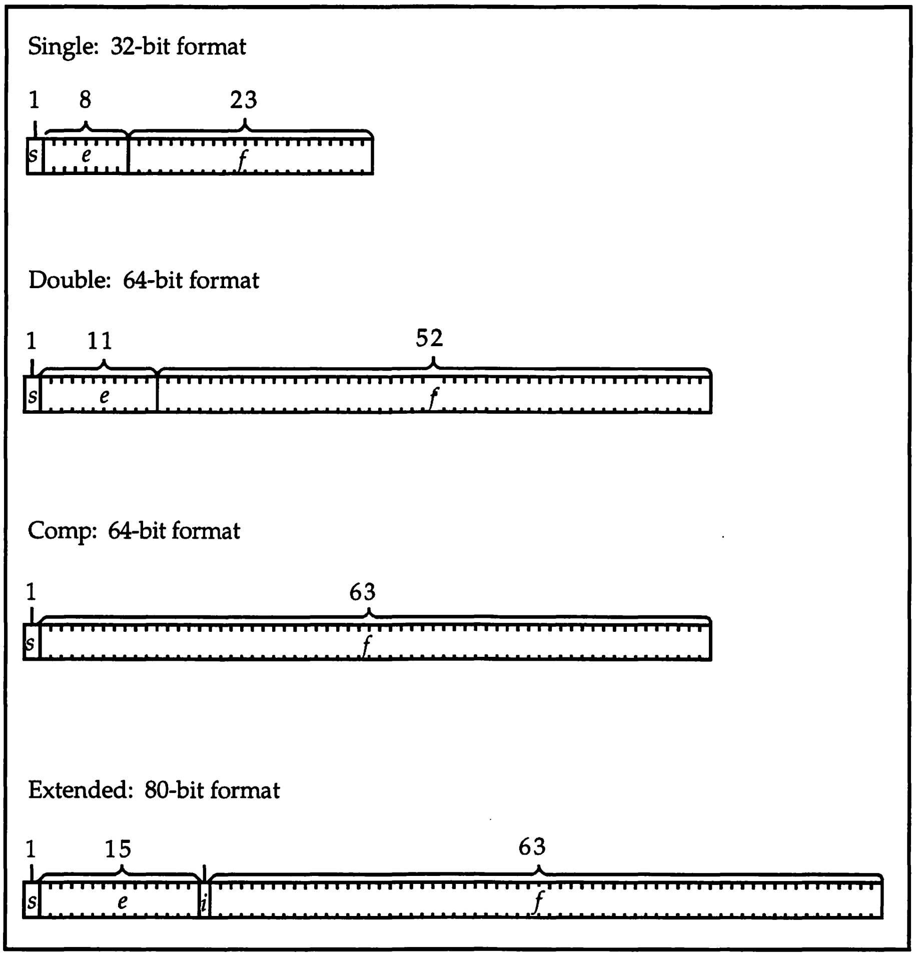
Appendix C

Introduction to the SANE Library

INTRODUCTION TO THE SANE LIBRARY AND SANE DATA TYPES

The SANE Library supports a set of numeric routines for performing both `extended` floating-point and integer computations. The core features of SANE have been taken from Draft 10.0 of Standard 754 for Binary Floating-Point Arithmetic as proposed by the Institute for Electronic and Electrical Engineers (IEEE). The SANE Library supports all of the requirements of the IEEE standard; it also goes beyond this standard by including data types and functions designed for scientific, engineering, and financial computations. The SANE Library provides arithmetic capabilities for both present-day and anticipated computer architectures. It does this without imposing an extra burden on the programmer or the user.

The SANE Library supports three application types and one arithmetic type for dealing with `real` and integer numbers. The application data types are `single`, `double`, and `comp` (computational), and the arithmetic type is referred to as `extended`. The `single`, `double`, and `extended` types store floating-point values, and the comp type stores integer values. Data types in SANE differ from those in other floating-point libraries because they support a format that represents the storage of positive and negative infinity as numeric values. SANE also supports formats for the existence of a non-numeric number (not-a-number, or NaN) and also stores floating-point numbers in a denormalized format. Figure C.1 shows the various internal formats for storing various SANE data types.



Key:

s = sign of the number
e = exponent of the number
f = significand of the number
i = explicit 1-bit

Figure C.1 SANE data types.

Notice that all floating-point formats (`single`, `double`, and `extended`) are composed of a sign bit on the left, followed by an exponent field, and a significand on the right. In general, the value of floating-point numbers is represented by the expression

$$(-1)^s * significand * 2^{exponent}$$

where the character `S` represents the sign bit of the number. If `S` is 0, the number is positive; if `S` is 1 the number is negative. Keep in mind that these floating-point formats are for storing numeric values at the machine level and are not character representations for floating-point numbers. Similar to the decimal point separating the integer and fractional part of a decimal number, the binary point of any floating-point format is implied to exist between the right of the exponent field and the left of the significand field (left of the exponent field and to the right of the explicit 1-bit in `extended`). The binary point is never explicitly stored in these formats; if it were, it would require an additional byte of storage, and would impose additional complications in executing numeric values. SANE supports routines for converting values between string representations and machine formats.

The significand is always assumed to have a single bit position to the left of the binary point. Only in the case of the `extended` format is this bit explicitly represented. For the `single` and `double` formats, storage for this bit is implied; it is not explicitly represented. This results in the significand having the following range:

```
0 <= significand < 2 .
```

In general, floating-point numbers are stored in a normalized form for maximum precision for a given significand width. Maximum precision is achieved when the high-order bit in the significand, the leftmost bit of the significand, is 1. This implies that the significand has the following range:

```
1 <= significand < 2 .
```

A floating-point value is stored as a denormalized number when the leading bit positions of the significand begin with zero. While a denormalized number fails to maximize the resolution of storage for a floating-point number, it does allow a floating-point number to have values beyond the limits of the minimum range for the exponent in a normalized form. For example, the smallest value for a normalized `single` number is approximately $-1.2*10^{-38}$, and the smallest value for a denormalized number is approximately $-1.5*10^{-45}$. Figure C.2 provides a table comparing the precision and range for all SANE data types including normalized and denormalized forms.

The `single` (equivalent to the `real` type discussed in Chapter 3) uses 32 bits or 4 bytes of storage (a byte is equivalent to 8 bits of storage). Eight bits are used to store the value of the exponent as a binary number. In this format the exponent has a value between 0 and 255 and represents a normalized value for the exponent in terms of a binary number. In this format the exponent is always stored as an unsigned binary integer number. The actual binary value for the exponent is the actual stored value minus 127. The remaining 23 bits to the right represent a fractional part (referred to as the *significand*) for storing the magnitude of the floating-point number.

The `single` number as a decimal (base 10) number can be determined from the following set of rules:

1. If $(0 < e)$ and $(e < 255)$, then $value <-- (-1)^s * (1.f)_{10} * 2^{(e-127)}$.

2. If $(e = 0)$ and $(f <> 0)$, then $value <-- (-1)^s * (0.f)_{10} * 2^{(-126)}$.

3. If $(e = 0)$ and $(f = 0)$, then value $<-- (-1)^s * 0$.

4. If $(e = 255)$ and $(f = 0)$, then *value* $<--(-1)^s * infinity$.

5. If $(e = 255)$ and $(f <> 0)$, then *value* is a NaN.

	Data Type			
	Single	**Double**	**Comp**	**Extended**
Machine Size				
Bits	32	64	64	80
Bytes	4	8	8	10
Exponent Range				
Minimum	-126	-1022		-16383
Maximum	127	1023		16384
Precision of the Significand				
Bits	24	53	63	64
Decimal digits	7 to 8	15 to 16	18 to 19	19 to 20
Approximate Decimal Range				
Minimum negative	-3.4E+38	-1.7E+308	-9.2E+18[a]	-1.1E+4932
Max. neg. norm.	-1.2E-38	-2.3E-308		-1.7E-4932
Max. neg. denorm.	-1.5E-45	-5.0E-324		-1.9E-4951
Min. pos. denorm.	1.5E-45	5.0E-324		1.9E-4951
Min. pos. norm.	1.2E-38	2.3E-308		1.7E-4932
Maximum positive	3.4E+38	1.7E+308	9.2E+18[a]	1.1E+4932
Supports				
Infinities	Yes	Yes	No	Yes
NaNs[b]	Yes	Yes	Yes	Yes

[a] Approximation for the magnitude 9,223,372,036,854,775,807.
[b] SANE representation for not-a-number.

Figure C.2 The precision and range of SANE data types.

Notice that infinity is represented by the exponent with the value of 255 (all exponent bit positions are 1) and a significand of zero, while NaN is represented by the exponent with the value of 255 and a significand that is non zero. The values $(1.f)_{10}$ and $(0.f)_{10}$ represent the binary values of $(1.f)$ and $(0.f)$ converted to a decimal number. The `double` type uses 8 bytes (64 bits) of storage with 11 bits for the exponent and 52 bits for the significand. The 3 additional bits in the exponent field increases the range of the exponent from plus or minus 38, for `single` precision to plus or minus 308 for `double` precision. The numeric precision (number of decimal digits) is increased from 7 to 15. This type of format allows the computation and storage of numbers that can be either larger or smaller than a `single` precision number. The value of a `double` precision number as a decimal (base 10) number can be determined from the following set of rules:

1. If $(0 < e)$ and $(e < 2047)$, then *value* $<--(-1)^s * (1.f)_{10} * 2^{(e-1023)}$.

2. If $(e = 0)$ and $(f <> 0)$, then *value* $<--(-1)^s * (0.f)_{10} * 2^{(-1022)}$.

3. If $(e = 0)$ and $(f = 0)$, then *value* $<--(-1)^s * 0$.

4. If $(e = 2047)$ and $(f = 0)$, then *value* $<--(-1)^s *$ *infinity*.

5. If $(e = 2047)$ and $(f <> 0)$, then *value* is a NaN.

Notice that infinity is again represented by an exponent containing all 1-bits and a zero significand, and NaN is represented by an exponent containing all 1-bits and a nonzero significand. Both `single` and `double` numbers have a numeric value of zero when all bit positions of the exponent and significand fields are zero. The `extended` format has 15 bits for storing the exponent and 64 bits for the significand. This allows scientific-notation numbers with very large or small exponent values to be stored and increases the numeric precision to 19 decimal digits. The `extended` mode offers the advantage of greater precision as well as a large exponent range. The value of an `extended` number as a decimal (base 10) number can be determined from the following set of rules:

1. If $(0 <= e)$ and $(e < 32767)$, then *value* $<--(-1)^s * (i.f)_{10} * 2^{(e-16383)}$.

2. If $(e = 32767)$ and $(f = 0)$, then *value* $<--(-1)^s *$ *infinity*, regardless of the value for *i*.

3. If $(e = 32767)$ and $(f <> 0)$, then *value* is a NaN, regardless of the value for *i*.

Notice again that infinity is represented by the exponent containing all 1-bits and a zero significand, and not-a-number is represented by the exponent containing all 1-bits and a nonzero significand, regardless of the value for the explicit 1-bit. The `comp` (computational) type is useful for extending the range of integer values. Its format is a `single` bit position for the sign, followed by 63 bits for the significand. This data type is useful in accounting applications, where results of computations must be both large and exact, and where money is to be represented as integral values for representing either cents or fraction of cents (mils). A `comp` type is capable of storing an integer number in the range from $-(2^{63}-1)$ to $(2^{63}-1)$. The value of a `comp` number as a decimal (base 10) number can be determined from the following rule:

```
If ( s = 1 ) and ( e = 0 ),
    then the value is a unique comp NaN;
else
    the value is a two's-complement number with a 64-bit representation.
```

As you can see, the `comp` type does not support a representation for numeric infinity. All integer numbers, whether they be of type `comp` or of type `integer`, are stored internally in two's-complement form. By using the two's-complement form, an integer zero will always be unique and positive. This is different from one's-complement form, where either positive or negative zero can exist.

INFINITY, NOT-A-NUMBER, AND DENORMALIZED NUMBER

The SANE Library supports two special numeric notations: infinity and not-a-number (NaN). Infinity is a special binary pattern that can be represented in either `single`, `double`, or `extended` mode. For each floating-point type, infinity is represented by an exponent having all 1-bits (a value of 255 for `single`, 2047 for `double`, and 32767 for `extended`) and a significand of zero. Computational types have no representation for infinities. Any attempt to assign a computational type an infinite value forces the computational type to become a computational NaN. Infinities can arise in one of two ways:

1. A SANE operation resulting in mathematical infinity such as a nonzero number divided by zero
2. A SANE operation producing a magnitude too large to be stored in an intended floating-point format

Symbolically, infinity is represented by the characters `INF`. Infinities can be stored as either positive (`+INF`) or negative (`-INF`) numbers. For example, 1 divided by 0 yields `+INF`, and –1 divided by 0 yields `-INF`. Infinities can propagate through SANE operations. For example, if 6.89 is added to `INF`, the result will still be `INF`. Note that 1 divided by `-INF` yields –0. SANE operations can also result in a special value representing not-a-number (`NaN`). `NaNs` can result from undefined operations: 0 divided by 0, `+INF` added to `-INF`, or taking the square root of a negative number. `NaNs` can be stored by any of the SANE data types: `single`, `double`, `extended`, and `comp`. Like infinity, `NaNs` are allowed to propagate through SANE operations. For example, if 6.89 is added to `NaN`, the result is `NaN`. There are two kinds of `NaNs`: quiet `NaNs` and signaling `NaNs`. A signaling `NaN` is one where a `NaN` is used as an operand of an arithmetic operation and an `INVALID` exception is signaled. If execution is not halted, the signaled `NaN` becomes a quiet `NaN`. It can be convenient to use `NaNs` to initialize storage of SANE data types. When doing this, the value must be explicitly written as `NaN(integer)`. For example, the following statement shows how an `extended` type called `Extra` is assigned the value of `NaN`:

```
Extra := NaN(255);
```

Using the value `NaN` or `NaN()` generally produces an error message when an attempt is made to execute an expression containing `NaN`. `NaN` in floating-point form has an associated code indicating the origin of `NaN`. Figure C.3 provides a list of codes and their meanings.

Name[a]	Meaning	Decimal Value	Hex Value[b]
NANSQRT	Unable to compute valid square root	1	$01
NANADD	Unable to compute valid addition	2	$02
NANDIV	Unable to compute valid division	4	$04
NANMUL	Unable to compute valid multiplication	8	$08
NANREM	Unable to compute valid remainder or **mod**	9	$09
NANASCBIN	Unable to convert ASCII character string	17	$11
NANCOMP	Error in converting `comp` NaN to float	20	$14
NANZERO	An attempt to create a NaN with an existing zero code	21	$15
NANTRIG	Invalid argument passed to a trigonomertic routine	33	$21
NANINVTRIG	Invalid argument passed to an inverse trigonomertic routine	34	$22
NANLOG	Invalid argument passed to a logarithmic routine	36	$24
NANPOWER	Invalid argument for an exponentiation routine	37	$25
NANFINAN	Invalid argument for a financial function	38	$26
NANINIT	Failure to initialize storage (signaling NaN)	255	$FF

[a] Names for error codes are in keeping with those given in the Macintosh Pascal Technical Appendix.

[b] Hexadecimal (hex) values are provided for reference.

Figure C.3 SANE NaN error codes.

Notice that the code number 255 represents signaling for uninitialized storage. Programmers are free to use other code values when assigning the value of `NaN` to a variable. Denormalized numbers are nonzero binary floating-point numbers in which the significand has a leading bit of zero. When a floating-point number stored in a normalized representation has to store a value that is too small for its format, it can be stored in a denormalized format. What is the importance of the denormalized number? In a computer system that does not produce a denormalized number, the following could occur if `B` is a variable with a small (less than the smallest normalized number) but not zero value:

$$A + B = A$$

That is, the normalized value of `B` is treated as if it has a value of zero although its denormalized form is nonzero. With the SANE system such numbers are tagged as denormalized, with the result

$$A + B <> A\ .$$

Building a Project That Uses SANE

Fortunately it is not difficult to use the SANE routines in your programs. The following program, which uses SANE to control precision (see Figure C.4), illustrates how this is done. The purpose of the example program is to compute the area under a curve, using a technique known as the *trapezoidal rule.* Numerous techniques exist for numeric integration; this one is used to demonstrate the difference in approximations, using three different precision settings: `RealPrecision`, `DblPrecision`, and `ExtendPrecision`, and in changing the number of points.

Rounding	Precision
Rounding to the nearest (default)	Extended precision
Rounding upward	Double precision
Rounding downward	Real precision
Rounding toward zero	

Figure C.4 Rounding and precision options supported by SANE.

Computing the area under a curve, represented by the function `f(x)` and the *x*-axis, is based on the summation of small trapezoidal areas over a closed interval. Figure C.5 demonstrates this concept, where the *i* th area is represented by the value

$$Delta\_x * [f(x_{i-1}) + f(x_i)] / 2,$$

and the value of `Delta_x` is represented by the term

$$(b-a)/n .$$

If the function `f(x)` is continuous on the closed interval `[a, b]` and if a regular partition of `[a, b]` is determined by the numbers

$$a = x_0, x_1, \ldots, x_n = b,$$

the area under the curve represented by `f(x)` is given by the approximation

$$Area = (b-a)/(2n)[f(x_0) + 2f(x_1) + \ldots + 2f(x_{n-1}) + f(x_n)]$$

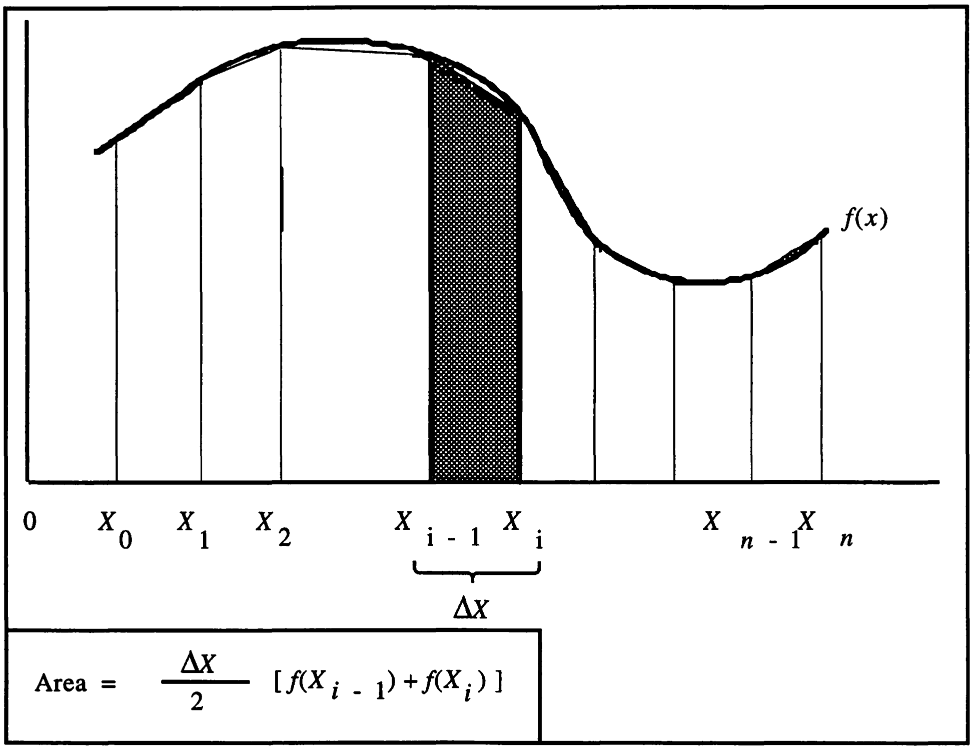
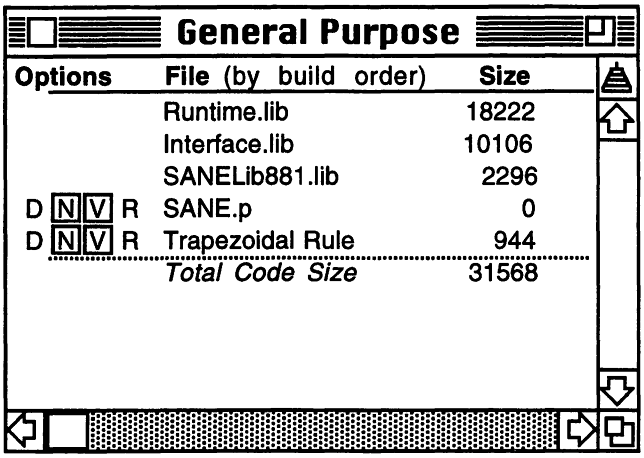


Figure C.5 The trapezoidal rule is based on the summation of individual trapezoids.

Accuracy in computing the area should improve as the value of *n* becomes larger. The more points you have for the curve representing the function `f(x)`, the more accurate your approximation. The computation for the area should be more precise for `extended` precision than for `real` precision. The following program, titled `Trapezoidal_Rule`, demonstrates how to test these expectations. Here the function `f(x)` is represented by the trigonometric function `sin`, with the closed interval for integration being `[0, π]`. The exact area under the sine curve is 2.0. This program computes the relative error given by the rule (`Estimated_Value – True_Value`)/ `True_Value` and displays the type of precision, the value of `n`, the computed value for area, and the relative error.

```
program Trapezoidal_Rule;
{ This program computes the area under the curve f(X) represented
{ by the trigonometric function sin. The closed interval [a, b] }
{ is represented by [ 0, pi ]. }
   uses
      Sane;
   const
      True_Area = 2.0;
   var
      Estimated_Area, Sum, Delta_x, X, Rel_Error: extended;
      C, B, A: extended;
      N, I: integer;
      Precision_Type: RoundPre;
begin
{ Close all windows, then open the Text window for viewing data. }
   HideAll;
   ShowText;
   writeln;
{ Compute a set of area values for each of the three SANE }
{ precisions. }
   for Precision_Type := RealPrecision downto ExtPrecision do
      begin
         SetPrecision(Precision_Type);
      { Initialize the end points of the interval [A, B]. }
         B := pi;
         A := 0;
      { Establish the initial number of intervals. }
         N := 200;
      { Compute the area for 25 different values of N. }
         writeln(Precision_Type);
         while N <= 5000 do
            begin
               Delta_x := (B - A) / N;
               X := A + Delta_x;
               Sum := sin(A);
            { Compute the summation of N-1 points of f(x). }
               for I := 1 to N - 1 do
                  begin
                     Sum := Sum + 2 * sin(X);
                     X := X + Delta_x;
                  end;
               Sum := Sum + sin(B);
            { Compute the area and display the value of N, Area, }
            { and Error. }
               Estimated_Area := (B - A) / (2 * N) * Sum;
               Rel_Error := (Estimated_Area - True_Area) /
                                                   True_Area;
               writeln(N : 4, '  ', Estimated_Area : 10 : 9, '  ',
                          Rel_Error);
               N := N + 200;
            end;
         writeln;
      end;
end.
```

In order to build this program, you must insert the SANE library into your project. You must also insert the interface Sane.p. Figure C.6 shows how the final project should look. Recall that the build order is important as you put this project together.



Options	File (by build order)	Size
	Runtime.lib	18222
	Interface.lib	10106
	SANELib881.lib	2296
D N V R	SANE.p	0
D N V R	Trapezoidal Rule	944
	Total Code Size	31568

Figure C.6 The project for the program `Trapezoidal_Rule`

One final point: as you build this project, you will discover that you actually have two SANE libraries to choose from. The one shown above is called `SANELib881.lib`, and the alternative is called `SANELib.lib`. Your choice of these libraries depends on whether your Macintosh is equipped with the MC68881/882 floating-point coprocessor. If this chip is present on your machine, use the file `SANELib881.lib`. Otherwise use the plain vanilla version.

Unfortunately, space prohibits us from giving a complete discussion of the SANE library and its use. We refer the reader to the first edition of this text for more complete coverage. Another source is the multiple volume set, *Inside Macintosh.* Finally, there is a book devoted specifically to SANE, the *Apple Numerics Manual.* Both of the latter works are published by Addison-Wesley. In the remainder of this appendix we will summarize the SANE functions and procedures that are available to the Pascal programmer.

TRANSFER ROUTINES IN CONVERTING SANE DATA TYPES

function `Num2Integer( E : extended ) : integer;`
This function takes the value as an `extended` type and returns a numeric value of type `integer`.

function `Num2Longint( E : extended ) : longint;`
This function takes the value as an `extended` type and returns a numeric value of type `longint`.

procedure `Num2Dec( Form : DecForm; E : extended;` **var** `D : Decimal );`
This procedure takes an `extended` value and, using a format provided by the value of `Form`, converts the value of `E` into a decimal `record` type. This converted value is returned through the formal parameter `D`.

function `Dec2Num( D : Decimal ) : extended;`
This function takes a value represented by a decimal record and returns a numeric value of type `extended`.

procedure `Num2Str( Form : DecForm; E : extended;` **var** `S : DecStr );`
This procedures takes an `extended` value and, using a format provided by the value of `Form`, converts the value of `E` into a `DecStr` type. The string representation is returned through the formal parameter `S`.

function `Str2Num( S : DecStr ) : extended;`
This function takes a value represented by a `DecStr` called `S` and returns a numeric value of type `extended`.

SANE INQUIRY AND ENVIRONMENT ACCESS ROUTINES

Inquiry Routines

function `ClassReal( R : real ) : NumClass;`
This function takes a real number represented by `R` and returns through the name of the function its corresponding numeric class.

function `ClassDouble( D : double ) : NumClass;`
This function takes a double-precision number represented by `D` and returns through the name of the function its corresponding numeric class.

function `ClassComp( C : comp ) : NumClass;`
This function takes a `computational` number represented by `C` and returns through the name of the function its corresponding numeric class.

function `ClassExtended( E : extended ) : NumClass;`
This function takes an extended-precision number represented by `E` and returns through the name of the function its corresponding numeric class.

function `SignNum( E : extended ) : integer;`
This function takes an extended-precision number represented by `E` and returns through the name of the function an integer value representing the sign of the floating-point number. The value returned is 0 if positive and 1 if negative.

Environment Access Routines

procedure `SetException( E : Exception; B : Boolean );`
This procedure allows one of five exception bits to be either set or cleared. If the Boolean value of `B` is *true* , the exception bit corresponding to `E` is set to 1. This is equivalent to forcing an exception flag to be raised. If the Boolean value of `B` is *false*, the corresponding exception bit is cleared, lowering the corresponding exception flag.

function `TestException( E : Exception ) : Boolean;`
This function interrogates the bit corresponding to the exception `E`. If the bit has been set to 1, the value returned through the name of the function is *true*. If the bit has been cleared, that is, if its value is 0, the value returned through the name of the function is *false.*

procedure `SetHalt( E : Exception; B : Boolean );`
This procedure allows one of five halt bits to be either set or cleared. If the Boolean value of `B` is *true* , the corresponding halt bit for the exception given by `E` is set to 1. If the Boolean value of `B` is *false*, the corresponding halt bit for the exception is cleared. This procedure has no effect on the bit positions for the exception flags.

function `TestHalt( E : Exception ) : Boolean;`
This function interrogates the halt bit corresponding to the exception given by E. If the bit has been set to 1, the value returned through the name of the function is *true*. If the halt bit has been cleared, the value returned through the name of the function is *false*.

procedure `SetRound( R : RoundDir );`
This procedure allows the rounding direction to be set to one of the four possible values: `ToNearest`, `Upward`, `Downward`, and `TowardZero`.

function `GetRound : RoundDir;`
This function returns the current rounding direction through the name of the function.

procedure `SetPrecision( P : RoundPre );`
This procedure allows the rounding precision to be set to one of the three possible values: `ExtPrecision`, `DblPrecision`, and `RealPrecision`.

function `GetPrecision : RoundPre;`
This function returns the current rounding precision through the name of the function.

procedure `SetEnvironment( E : Environment );`
This procedure allows the environment represented by `E` to be set, thereby replacing the current environment settings. The value of `E` can affect the settings of rounding direction, rounding precision, exception flags, and halts.

procedure `GetEnvironment(` **var** `E : Environment );`
This procedure returns the current environment setting represented by the formal parameter `E`.

procedure `ProcEntry(` **var** `E : Environment );`
This procedure saves the current environment by returning a value through the formal parameter `E` and then establishes a default environment. All halts are now disabled.

procedure `ProcExit( E : Environment );`
This procedure restores the environment represented by `E`, including any halts that were disabled. It is possible that previous exceptions either raised or set by the procedure `SetException` will result in program execution being interrupted with an error message displayed in a dialog box.

ARITHMETIC, AUXILIARY, AND ELEMENTARY FUNCTIONS

Comparison Routine

function `Relation( X, Y : extended ) : Relop;`
This function takes two values, `X` and `Y`, and returns through the name of the function one of four possible relationships: `GreaterThan`, `LessThan`, `EqualTo`, and `Unordered`.

Arithmetic and Auxiliary Routines

function `Remainder( X, Y : extended;` **var** `Q : integer ) : extended;`
This function takes two `extended` values, `X` and `Y`, representing the dividend and divisor, respectively, and computes the quotient and remainder according to the IEEE standard. The quotient is returned through the formal parameter `Q` as an integer, and the remainder is returned through the name of the function.

function `Rint( X : extended ) : extended;`
This function takes an `extended` value `X` and rounds to an integer value. The rounded value as an `extended` type is returned through the name of the function. The value returned depends on the current rounding direction set through execution of procedure `SetRound`.

function `Scalb( N : integer; X : extended ) : extended;`
This function efficiently scales the given number `X` by a factor of 2 raised to a given power `N`. The value returned through the name of the function represents the product $X * 2^{N}$.

function `Logb( X : extended ) : extended;`
This function takes the value `X` and returns through the name of the function the binary exponent of `X` as a signed `integer` value. If `X` represents a denormalized number, the exponent is determined as if the argument is normalized. Two special cases exist: (1) If `X` is infinite, the value returned is `+INF`. (2) If `X` is zero, the value returned is `-INF`, with the exception flag `DivByZero` being raised.

function `CopySign( X, Y : extended ) : extended;`
This function takes two values, `X` and `Y`, and returns through the name of the function the magnitude of `Y` attached to the sign of `X`.

Next-After Functions

function `NextReal( X, Y : real ) : real;`

function `NextDouble( X, Y : double ) : double;`

function `NextExtended( X, Y : extended ) : extended;`
Each of these three functions takes the value of `X` and returns through the name of the function the next representable value after `X` in the direction given by `Y`.

Elementary Functions

function `log2( X : extended ) : extended;`
This function returns the base 2 logarithm for the given value `X`.

function `ln1( X : extended ) : extended;`
This function returns the base *e* logarithm for the value `X + 1`.

function `exp2( X : extended ) : extended;`
This function returns the value of 2 raised to a power given by value of `X`.

function `exp1( X : extended ) : extended;`
This function returns the value of the natural number `e` raised to a power given by the value of `X` minus 1 (e^{X-1}).

function `XpwrI( X : extended; N : integer ) : extended;`
This function returns the value of `X` raised to an `integer` value `N`.

function `XpwrY( X, Y : extended ) : extended;`
This function returns the value of `X` raised to the floating-point value of `Y`.

function `Compound( R, N : extended ) : extended;`
This function computes the compounded value $(1 + R)^{N}$ and returns the value through the name of the function.

function `Annuity( R, N : extended ) : extended;`
This function computes the annuity value $(1 - (1 + R)^{(-N)}) / R$ and returns the value through the name of the function.

function `tan( X : extended ) : extended;`
This function returns the value representing the tangent of `X`, where `X` represents an angle in radians.

function `RandomX(` **var** `S : extended ) : extended;`
This function takes the value of `S` as a seed and returns a number from a sequence of pseudorandom numbers having values in the range $1 <= S <= 2^{31} - 2$. If this function is again executed, the next random number in the sequence is returned through the formal parameter `S`. If the seed value `S` is a non integral value or is outside the specified range, the results are unspecified.

function `NaN( N : integer ) : extended;`
This function converts a standard 16-bit integer into `NaN`. The value `NaN` is returned through the name of the function.

Appendix D

Creating a Macintosh Pascal Application

Version 3.0 of Macintosh Pascal provides the capability of converting a Macintosh Pascal program into an application program using the P-Shell. Figure D.1 shows the Macintosh folder with the P-Shell. For an application program to be generated, the P-Shell file must reside either in the Macintosh Pascal folder or at least in the same folder as the application file.

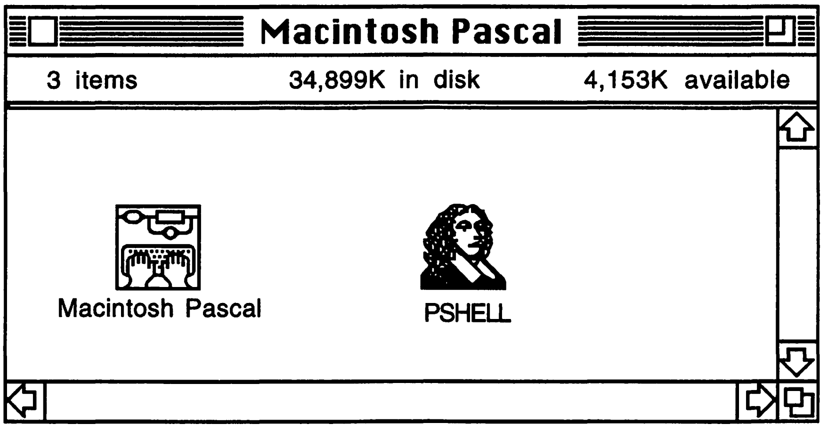
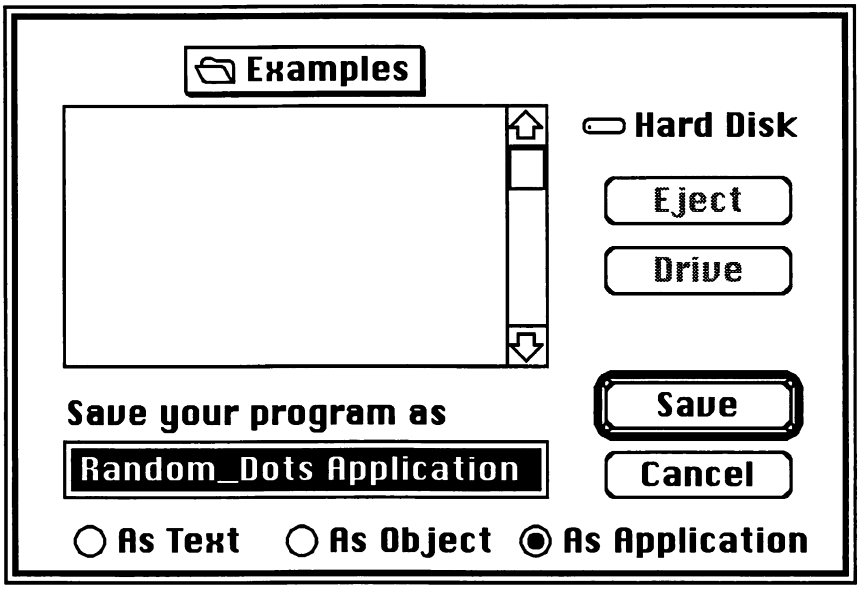


Figure D.1 The PSHELL icon.

Using the P-Shell, we can execute an application program directly by double-clicking its corresponding name or icon from the Macintosh Pascal window rather than having to invoke the Pascal translator directly.

While the program window is active and contains a copy of the program to be transformed into the application, the process of creating a Macintosh Pascal program requires four steps:

1. Pull down the **File** menu and highlight the **Save As...** option.
2. When the dialog box (see Figure D.2) appears, enter a name that is different from the Macintosh Pascal program name.
3. Click the **As Application** option.
4. Click the **Save** option.



Save your program as
Random_Dots Application

○ As Text ○ As Object [✓ As Application]

Figure D.2 The Macintosh Pascal **Save As...** dialog box.

The result is different from other files that have been saved. You will notice that at the end of the operation the new name given to the application file does not appear in the title bar. If you choose to quit the Pascal translator, the name or icon of the application file can be viewed from the Macintosh Pascal window. Figure D.3 shows two files; one is a Macintosh Pascal program, and the other is an application program.

The following program illustrates some of the problems of transforming a Macintosh Pascal program into an application.

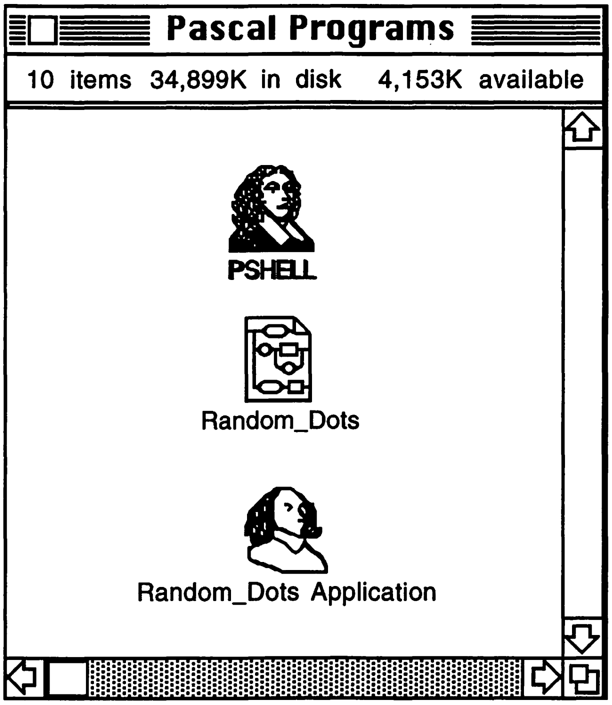


Figure D.3 The Macintosh Pascal application icon.

```
program Random_Dots;
{ Purpose:  This program provides random squares with different }
{           patterns about the Drawing window. }
   var
      Top, Left, Bottom, Right, X : integer;
      Pat : Pattern;
      Background : integer;
begin
{ Use the function Random to choose the corners of the square }
{ and the pattern randomly.  Initialize the counter. }
     X := 1;
{ Repeat the loop while the counter value is less than or equal }
{ to 200. }
   while X <= 200 do
      begin
      { Open Text and Drawing windows for viewing by the user. }
         ShowText;
         writeln('Cycle:   ', X);
         ShowDrawing;
      { ------------------------------------------------------ }
      { The size of each window is set from the Windows menu }
      { when the program is composed and translated. After the }
      { size of each window has been established, it may be }
      { better to execute SetTextRect and SetDrawingRect. }
```

```
        { ---------------------------------------------------------- }
        { Randomly select the rectangles for drawing an oval. }
           Top := abs(random) mod 201;
           Left := abs(random) mod 201;
           Bottom := Top + 30;
           Right := Left + 30;
        { Randomly select the background color. }
           Background := abs(random) mod 5;
           case background of
              0 :  Pat := white;
              1 :  Pat := black;
              2 :  Pat := gray;
              3 :  Pat := ltgray;
              4 :   Pat := dkgray;
           end;
        { Display the oval in a rectangle with a chosen background }
        { pattern. }
           FillOval(Top, Left, Bottom, Right, Pat);
           X := X + 1;
        end;
end.
```

Before converting a program into an application, the commands `ShowText` and `ShowDrawing` must be appropriately placed. Failure to call on these routines results in a blank screen when the application program is executed. Remember that when an application program is being executed, no menu bar is displayed.

Not properly setting the window sizes by executing `SetTextRect` or `SetDrawingRect` can result in either window having the wrong size or being improperly placed on the screen as the application program is executed. During execution it is impossible to change the size or close any window by clicking or moving the mouse. The best rule is to be prepared. Thoroughly test your Macintosh Pascal program before converting it into an application program. Do not attempt to use an application program for the purpose of debugging a Macintosh Pascal program. An application program that has entered into an infinite loop can only be halted by turning off the Macintosh computer.

If you want to execute an application program without going through the Finder, highlight the title or icon of your application and then select the menu **Special**. Next select **Set Startup...** Provided that your diskette contains a System Folder, your application goes into immediate execution when your Macintosh machine is started.

For resetting the startup place to the Finder, quit the present application and highlight the Finder icon. Now choose the option

Set Startup... from the **Special** menu. At this point the Finder will be executed when the Macintosh machine is started.

Appendix E

Using Labels and the Pascal `Goto` Statement

Both Macintosh and THINK Pascal support a special unconditional transfer statement called **goto**. Syntactically this statement appears as follows:

```
goto label
```

where `label` is represented as an unsigned integer whose value must be in the range 0 through 9999. When executed, the program branches to the statement given by the label and continues execution. Like other objects, labels must be declared. For example, consider the following program for displaying the numbers 1 through 100. Rather than using one of the looping constructs, the program uses a one-way branch and a **goto** statement to simulate a **while-do** construct:

```
program Display_Number;
{ This program displays 100 numbers across the Text window. }
   label
      10;
   var
      Number: integer;
begin
{ Show  the Text window to the user. }
   ShowText;
10:
   if Number <= 100 then
```

```
      begin
         writeln(Number);
         Number := Number + 1;
         goto 10
      end;
   writeln;
end.
```

There are several rules for using the **goto** statement:

1. A **goto** statement can be used for transferring either forward or backward within the body of a compound statement.
2. A **goto** statement can be used to exit from within the body of a compound statement, or a **then** clause of an **if-then** statement, or a **then** clause of an **else** clause within an **if-then-else** statement. A **goto** statement cannot be used to transfer execution into the body of a compound statement, a **then** clause, an **else** clause, or the body of a **case** statement. It can be used to transfer execution to the beginning of a compound statement, an **if-then** or an **if-then-else**, **case**, or simple statement.
3. Labels given within the body of a case statement are not the same as those declared through label declaration. Case labels are referred to as *case constants* and are type-associated with the case selector.
4. A **goto** statement can transfer execution from within the body of a procedure or function to a location outside this program unit. For example, the following program will transfer execution from within the body of procedure `Test` to the statement labeled `100`:

```
program Transfer_Out;
{ Sample program for testing an unconditional branch. }
   label
      100;
   procedure Test;
   begin
      writeln('Executing the body of procedure Test.');
      goto 100
   end {Test};

begin
   ShowText;
   Test;
100:
   writeln('Back in the main program.')
end.
```

When **goto** `100` is executed, execution of the procedure `Test` is terminated before the **end** statement of `Test` is executed.

5. Like constants, types, and variables, labels satisfy the scoping rules for being local and global.

When should the **goto** statement be used? That depends on the problem being solved. For example, the following code represents the case of an in-test iteration loop:

```
   Sum := 0.0;
   Counter := 1;
120:
   begin
      Sum := Sum + A[Counter];
      if  Counter = 10 then
         goto 150;
      Counter := Counter + 1
         goto 120
   end;
150:
```

In most cases we can avoid the need for a **goto** statement by using one of the other constructs such as **if-then**, **if-then-else**, **while-do**, **repeat-until**, or **for**. Excessive use of the **goto** statement can lead to poorly structured programs that are difficult to read and debug.

Appendix F

References

Following is a list of references for readers who want to enhance their knowledge of programming and related topics. There are numerous books that discuss Pascal and programming topics relating to Pascal. The list presented here is limited, however. The books listed are those we believe are significantly related to the content of this one, and those that offer additional knowledge beyond its framework.

The reference manual cited in [1] provides a guide for installing and using Macintosh Pascal. It also is an excellent resource on the syntax and semantic rules of Pascal, whether the reader is using Macintosh Pascal or Symantec's THINK Pascal. Appendix B of [1] provides a lengthy review of the QuickDraw library, and Appendix C of [1] reviews the Standard Apple Numeric Environment, SANE.

For those readers using Symantec THINK Pascal with the intention of developing software applications, references [2], [3], and [4] are important. Reference [2] provides information on installing THINK Pascal as well as on editing, establishing, and using projects, running and debugging programs, applying units and libraries, building projects, applying compiler directives, and interaction with the debugger, LightsBug. Later chapters provide information on using resource-description files, and the steps needed when interacting with the resource editors SARez, SADeRez, and SAPostRez.

There are several books that provide lengthy introductions to Computer Science using Pascal. Nance and Naps [5] offers a lengthy introduction on programming, problem solving, and data structures using Standard Pascal. In much of the book, unrefined algorithms are presented using a pseudocode that includes constructs of Standard Pascal. Most of the algorithms are redefined through a listing of full Pascal programs.

For readers wanting an in-depth study of file systems, the book by Miller [6] is good. It is a comprehensive introduction to files, with numerous examples. In many of the chapters, structured algorithms are used for reviewing and reinforcing file concepts and theories. Several of these algorithms are implemented in Pascal.

There are several books that provide an excellent introduction to data structures. The book by Wirth [7] is a classic in the field of Computer Science and offers an excellent introduction to searching and sorting algorithms. Wirth also provides an introduction to tree structures, with algorithms reviewing balanced tree insertions and deletions. For multiway trees, he provides an introduction by describing B-Tree algorithms on insertion and deletion. All of the algorithms are expressed in Standard Pascal. This book requires concentration, and the algorithms must be traced to reinforce understanding of concepts and theories. Horowitz and Sahni [8] have also written a classic book on data structures. This book reviews the concepts of stacks and queues, trees, graphs, internal and external sorting, hashing, files, and advanced tree structures. All algorithms are written in Standard Pascal. A more recent book by Kruse [9] on data structures places emphasis on aspects of software engineering. It provides two lengthy case studies that present many of the concepts in data structures. All algorithms are expressed in Standard Pascal.

Several sources exist for learning more about object-orienting programming, object-oriented analysis, and object-oriented design. First, the reference manual [3] provides a brief introduction to using the object-oriented features of THINK Pascal. Though not a formal text on the subject of object-oriented programming, it provides the necessary information for getting started with objects, installing the THINK class library, reviewing basic programming concepts of object Pascal, and learning the Class Browser. It also has a brief review of all class types and class methods supported by the THINK class library. The book by Sphar [10] offers more specific material on programming with object Pascal. It is divided into three major parts: defining object-oriented programming, object-oriented applications, and polymorphic software components. Throughout the text, examples are given in THINK Pascal.

For a more detailed introduction to object-oriented design (OOD) and object-oriented analysis (OOA), see references [11], [12], [13], [14], [15], and [16]. At present, there is no single approach that best describes OOA and OOD. It is clear, however, that object-oriented programming requires a study of OOA and OOD. Object-oriented programming is more than simply changing the structure of the program code. It requires analysis and design to properly model the application being implemented in object Pascal.

References [17], [18], [19], [20], [21], and [22] provide a more detailed view of the Macintosh tools. Some of these references deal specifically with sound and the QuickDraw Toolbox.

REFERENCES

[1] Symantec, *Macintosh Pascal: Reference Guide*, Symantec Corporation, 1988, Cupertino, CA.

[2] Symantec, *THINK Pascal, The Fastest Way to Finished Software: User Manual,* Symantec Corporation, 1991, Cupertino, CA.

[3] Symantec, *THINK Pascal, The Fastest Way to Finished Software: Object-Oriented Programming Manual,* Symantec Corporation, 1991, Cupertino, CA.

[4] Symantec, *THINK Pascal, The Fastest Way to Finished Software: Resource Utilities Manual,* Symantec Corporation, 1990, Cupertino, CA.

[5] Douglas W. Nance, Thomas L. Naps, *Introduction to Computer Science: Programming, Problem Solving, and Data Structures,* 2d ed., West, 1991, St. Paul, MN.

[6] Nancy E. Miller, *File Structures Using Pascal*, Benjamin/ Cummings Publishing Company, 1987, Menlo Park, CA.

[7] Niklaus Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall, 1976, Englewood Cliffs, NJ.

[8] Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures in Pascal*, Computer Science Press, 1990, New York.

[9] Robert L. Kruse, *Data Structures and Program Design,* 2d. ed., Prentice Hall, 1987, Englewood Cliffs, NJ.

[10] Chuck Sphar, *Object-Oriented Programming Power for Think Pascal Programmers*, Microsoft Press, 1991, Redmond, WA.

[11] Grady Booch, *Object-Oriented Design With Applications*, Benjamin/Cummings, 1991, Redwood City, CA.

[12] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991, Englewood Cliffs, NJ.

[13] Peter Coad and Edward Yourdon, *Object-Oriented Analysis,* 2d ed., Yourdon Press, 1991, Englewood Cliffs, NJ.

[14] Peter Coad and Edward Yourdon, *Object-Oriented Design*, Yourdon Press, 1991, Englewood Cliffs, NJ.

[15] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990, Englewood Cliffs, NJ.

[16] David W. Embley, Barry D. Kurtz, Scott N. Woodfield, *Object-Oriented Systems Analysis: A Model-Driven Approach*, Yourdon Press, 1992, Englewood Cliffs, NJ.

[17] Stephen Chernicoff, *Macintosh Revealed,* Vol.1, *Unlocking the Toolbox,* 2d ed., Hayden Books, 1987, Indianapolis, IN.

[18] Stephen Chernicoff, *Macintosh Revealed,* Vol.2, *Programming with the Toolbox,* 2d ed., Hayden Books, 1987, Indianapolis, IN.

[19] David A. Surovell, Frederick M. Hall, and Konstantin Othmer, *Programming QuickDraw*, Addison-Wesley, 1992, Reading, MA.

[20] Christopher L. Morgan, *Hidden Powers of the Macintosh*, The Waite Group, 1985, New York.

[21] William B. Twitty, *The Magic of Macintosh: Programming Graphics and Sound*, Scott, Foresman, and Company, 1986, Glenview, IL.

[22] Dave Mark and Cartwright Reed, *Macintosh Pascal Programming Primer,* Vol.1,*Inside the Toolbox Using THINK Pascal*, Addison-Wesley, 1991, Reading, MA.

Index

+ (*addition*), 106
- (*subtraction*), 105
/ (*division*), 106
* (*multiplication*), 106
= (*to establish a constant*), 116
:= (*assignment*), 115
π (*see also* pi), 32, 34, 124, 345

- A -

ABS(X), 122
abstract class, 632-34
address, 3, 71, 95, 253-56, 298, 360, 410-12, 471, 501, 575, 581-82, 645
algorithm, 60-64, 79-81, 110, 113, 117-19, 138-39, 142-43, 147-48, 150-51, 155-57, 165-69, 211, 213, 215, 218-19, 230-32, 251-52, 257, 278-81, 306, 308-10, 312, 322-23, 374, 393-94, 396, 410, 412-13, 419, 421-27, 440, 445, 475, 480, 483-85, 491-96, 502, 511, 545, 553, 556, 559, 572, 585, 603, 609-14, 660, 662, 666
    bubble sort, 412-13, 452-53
    linear search, 425, 427, 453, 502
    search, 391, 421, 425
    Shellsort, 421-23
    sorting, 421, 425
    straight insertion sort, 421-22
    quicksort, 421, 423-25
APPL, 369
application (*see also* program), 2, 13, 38, 42, 286, 307, 323-24, 343-44, 359, 372, 374, 379, 501, 549, 573, 667-68
    program, 4, 31
arithmetic
    functions, 106, 120-21, 123-24, 129
    operations, 2, 84, 86, 106, 113, 127, 350
artificial language, 2, 62
ASCII, 88, 93-94, 582
assignment (*see also* :=), 71, 92, 107, 115, 118-19, 141-43, 145, 147-48, 159, 276, 281, 290, 324, 393, 437-38, 448, 467, 505, 582-83, 614, 617, 645, 700

- B -

BackColor, 199, 234-35
BackPat, 683, 721, 727-28
Beanpicker, 165-66, 168-69, 171, 221, 224-26, 228
begin-end, 144, 255
Bic, 716-17

binary
    digits (*bits*), 3
    search algorithm, 426-27, 451, 483, 485
    trees, 573, 602
        growing and pruning, 573
        leaf node, 602, 608
        parent node, 602
BitMap, 680, 719
bkColor, 680-81
bkPat, 680-81, 692-94
BlockMove, 617-618, 621
body of the main program, 252, 254-55, 277, 283-85, 288, 294, 343, 408, 431, 482-83, 590
Bold, 7-8, 39, 190, 704-05
boolean, 83, 86, 88-90
    expressions, 139, 160, 173-74
    operators, 138, 146, 176, 454, 533
boot, 3
branching statement, 143, 147
bug, 11, 36, 48, 51, 171
Build, 42-43, 341, 345-46, 368-69, 379, 634
Build Library, 42, 50, 354
building THINK Pascal libraries, 349
built-in constants, 70
button, 5, 12, 119, 206-9, 217, 219-21, 647, 656, 698, 721
byte, 3, 20, 446, 448, 533, 545, 621

- C -

case statement, 154-55, 158, 171, 180, 204, 437, 503, 701
central processing unit, (*see also CPU*) 2
char type, 102, 119, 155, 254, 446, 448, 533, 550
character data, 466
CharWidth, 705
Check, 11-12, 43, 48, 109, 118, 156-57, 167-71, 189, 206, 226, 278-79, 303, 345-46, 351, 358-59, 404, 418, 476, 484, 487, 494, 496, 499, 545-47, 591, 597, 602, 613, 634, 675
class, 46, 50-51, 61, 90-91, 628-34, 636-42, 644-47, 650, 653, 656-60
Class Browser, 30, 46, 51, 628, 656-57, 668-70
Clear, 15, 19, 39, 50, 79, 150, 171, 229, 257, 367, 504, 575
Clipboard, 50
ClipRect, 681, 683, 686-87, 699, 702, 721
clipRgn, 680-83, 686, 688, 717, 719
Clone, 645-46, 668-69
Close, 12, 14, 38, 41, 50, 59, 109, 209, 214, 268-69, 343, 428, 467-69, 475, 482, 484-90, 499, 522-23
close box, 14, 209
ClosePicture, 714-16

ClosePoly, 712-13
ClosePort, 682, 697, 710-11, 718, 720, 725
CloseRgn, 708, 710-11, 727
colrBit, 680-81
command
  branching, 147
  for, 74, 139, 163-65
  input, 60, 74, 78, 467
  that provides action, 2
  Type, 96
comments, 54, 66, 76, 79, 118, 173, 252, 399, 405, 614, 701
compile, 36, 41-43, 46, 50, 189, 263, 265, 280, 345-46, 357, 370, 379, 531
compound
  conditional, 138, 160, 174-76
  statement, 143-44, 147-48, 175, 328
computer
  automated machine, 19
  languages, 2, 325
  organization, 2
  program, 1-2, 8, 20, 61-62, 138, 178, 180, 187
Concat, 538-39, 544, 548-49, 684, 701-3, 706
concrete class 632, 669
condense, 684, 687, 704-5, 738
conditional expressions, 138-39, 145, 161
const heading, 116
constant, 66-71, 85, 109-10, 115-16, 123-24, 154, 221, 251-52, 259, 264, 277, 359, 392-93, 441, 616, 701, 720
  equated with the name, 68
  predeclared, 68, 90-92, 101, 105
  real, 70
control structures, 139, 161
  for branching, 147
coordinate plane, 675-77, 679-81, 695, 701, 734
Copy, 14-15, 17, 38-39, 48, 50, 194, 268, 288, 294, 345-47, 429-30, 468, 481, 491, 493-96, 538-40, 579, 612-14, 617, 645, 683, 708, 716-17, 721
CopyBits, 717, 719-21, 733
CopyRgn, 708
CPU, 2, 374, 376
counters, 139, 163
Cursor, 45, 50, 119, 206, 209, 678, 721, 728
cursors and patterns, 721
Cut, 14-15, 17, 19-20, 39, 50, 54, 194, 375

- D -

data objects, 2, 61, 70, 75, 98, 116, 165, 465, 573-75, 593, 598
  nonnumeric, 99, 532, 550
  special, 573
data type
  casting, 127
  computational, 60, 83-85, 98-99, 127, 524, 550
  declarations, 96, 633
  integer, 108, 195-96, 199, 210
  longint, 86
  ordinal, 60, 83, 86, 88, 90-94, 101, 391
    nonstandard ordinal, 83, 99
  real , 71, 83-84, 195
  simple, 83, 98-99
  string, 119, 465, 467, 533, 535, 537-38, 550, 573-74
data structures, 585
  array (*homogeneous structure*), 391
    multidimensional, 391, 408
    of arrays, 416
    one-dimensional, 527
  record (*inhomogeneous structure*), 427-28
DateTimeRec, 428-30, 453, 457, 531, 635
Debug and Names, 370, 374
Debug Menu, 119, 45, 51-52, 259, 374-75, 377, 379, 382, 389-90
debugging, 9, 120, 30, 43, 50, 348, 657
decimal, 70, 77-78, 83-84, 88, 111-12, 706, 722-23
declarations, 66, 68, 70-71, 92, 95-96, 116, 250-52, 277, 285-86, 288, 290, 324, 338-41, 343, 351, 356, 358, 433, 437, 575, 632-34, 641, 656, 659-60, 675, 679, 713, 728
defining a port, 682
definiteness, 61, 79, 165
degrees to radians, 124
delete, 15, 8, 50, 59, 535, 540-44, 549, 574, 591-93, 607, 706
desktop folder, 209
development folder, 31-32
device, 5, 12, 251, 360, 500, 680-81
dialog box, 11-13, 16-18, 32, 35, 37-39, 42-45, 48-50
dictionary, 602-07, 621
DiffRgn, 28, 708, 711
Digits, 3, 65, 77, 84-86, 98, 112, 117-18
direct recursion, 386
disk drive, 4-5, 31
dispose, 573, 575, 584, 602, 610-611, 614, 616, 637-38, 642, 645, 664-65, 684-87, 710-11, 715, 718-21
DisposeHandle, 617-18, 621
DisposeRgn, 707, 710-13, 719, 727, 736

div, 106, 108, 110-12, 116-18, 127, 156-57, 171, 484
Do It button, 16, 45, 109, 194, 379
double, 83-85, 98-99, 127, 349, 370, 374, 441, 542-44, 617, 656
dragging the mouse, 189-90, 206, 209
DrawChar, 705-6
DrawCircle, 675
drawing pen, 195-96, 198, 205, 215, 240, 441, 688-89
drawing
- lines, 192, 194, 219-20, 688-90, 716
- pictures, 674, 713-14
- rectangles, 211, 236
- simple geometric patterns, 196
- simple lines, 188, 192

Drawing window, 7-8, 11, 12, 16, 46, 66, 72, 109, 119, 188, 370, 646-47, 715
drawing with
- arcs and wedges, 693
- points, lines, and rectangles, 688
- regions and polygons, 707

DrawLine, 194-96, 214, 217, 220-21, 236, 675, 693
DrawPicture, 714-16
DrawString, 684, 699-705, 707, 721

- E -

Edit, 14-16, 34-48, 194, 344-49, 353, 377, 656
effectiveness, 61, 165
else clause, 148-49, 159, 166-67, 494
EmptyRect, 691
EmptyRgn, 709
end-of-file marker, 465, 467, 489, 525
eof, 465, 489, 523
eoln, 443, 447, 449, 498-99, 706
equal sign (*see also* =), 68, 97, 100
EqualRect, 691
EqualRgn, 709
EraseArc, 694
EraseOval, 203-04, 232-33, 236, 692
ErasePoly, 712-13
EraseRect, 198-99, 201, 205, 236, 317, 320, 691, 715, 718, 721
EraseRgn, 709-10, 727
EraseRoundRect, 693
error, 11-12, 16, 34, 41, 45, 47-48, 50, 64-65, 69, 76-78, 87, 90, 94-95, 112, 114, 118, 146, 154-55, 164, 171, 189, 195, 205, 211, 227, 263, 280, 285-86, 288, 318-19, 321, 324, 359, 368, 374, 393, 404-5, 410, 433, 437, 448, 465, 467, 500, 544, 576-77, 618, 633, 645
exceptions, 98, 189, 574-75, 594

Exp, 124-26, 130, 133, 155
expression, 47, 68-70, 75, 90, 106-8, 113-16, 118-21, 127, 145, 154, 161, 164-72, 259, 264, 278, 294, 324, 393, 437-38, 448, 468-70, 481, 494, 497, 533, 539, 578-79, 617, 700
extend, 40, 251, 323, 379, 604
extended, 43, 83-85, 98-99, 108, 121, 125, 127, 428, 450, 540, 550, 614

- F -

fgColor, 680-81
Fibonacci series, 303-4, 333
file, 4, 18, 30, 32, 36, 38-39, 43, 45-46, 87, 171, 251, 338-39, 341-49, 351, 353-54, 356, 360-61, 369-371, 464, 522-85, 612, 617
  advantages of using, 464, 525
  actual, 268
  buffer, 467-84, 515
  component, 465-99
  devices, 499
  library, 32, 34, 361
  logical, 466-67, 501, 526
  nontext, 464
  object, 13, 42, 369
  opening and closing, 464
  pointer, 464, 516
  random access, 480
  text, 464-66, 472, 475, 496-500, 523
filepos, 464, 483, 526
FillArc, 694, 703
filling with a pattern, 198
FillOval, 203-5, 231-33, 236, 651, 692, 718
FillPat, 680-81
FillPoly, 713
FillRect, 198-217, 323-24, 435, 442, 652, 684, 686-87, 692, 699-700, 702, 718, 721
FillRgn, 709-11, 727, 732-33
FillRoundRect, 693
Find, 12, 16, 31, 39-41, 49-50, 63, 66, 118
finiteness, 61, 165
firmware, 188-89
fixed and variant fields, 391
folder name, 499
font, 17-18, 34, 39, 44, 47, 196, 217-18, 500, 680-81, 704-5
  Control, 17-18, 20, 196, 217, 500
  size, 17, 39, 44, 218, 705,
for-to, 163
ForeColor, 199, 234-36, 241

formal parameter, 253-78, 285, 288-92, 298, 303, 324, 340, 360, 396, 404-5, 411, 413, 417, 428, 471, 507, 545, 579, 583, 590, 637
  list, 252, 254, 264, 270, 276-78, 285, 288, 409
  declared as array types, 404
forward declaration, 286, 288, 306
FrameArc, 694
FrameOval, 203-7, 236, 692, 703, 711, 718, 727, 731
FramePoly, 712-13
FrameRect, 190, 198-01, 205, 214, 235-36, 435, 442, 445, 678, 684, 686-87, 691, 699, 718, 721, 727
FrameRgn, 28, 709-11, 727
FrameRoundRect, 692, 727
framing, 198, 710-11

- G -

Get, 31, 43, 46, 49-50, 219-21, 223, 472-73, 475, 481, 490, 630, 632-33, 637, 640, 642-44, 648
GetClip, 683
GetDrawingRect, 189, 211, 241, 675
GetHandleSize, 617, 621
GetMouse, 189, 206-08, 219-21, 314, 507, 648, 718, 727
GetPen, 688
GetPenState, 688, 690
GetPort, 682, 734
GetTextRect, 189, 211, 241
GetTime, 428, 431, 457, 635
global identifiers, 284-85, 340
global variables, 285, 322, 374, 396, 457
GlobalToLocal, 729, 731
Go, 8-12, 20-21, 23, 32, 35-36, 42-43, 45, 61, 63, 109, 119-20, 166, 206, 229, 348, 376-77, 379, 613
Goto, 767
Go-Go, 11-12, 20, 36, 43, 51, 53, 119-20
GrafPort, 192, 234, 680-92, 694, 704-5, 715-17, 729-31
GrafPort routines, 682
GrafProcs, 680, 682
GrafPtr, 680, 683, 686, 697, 710, 714, 717, 720, 724
graphical system, 188

- H-

Halt, 12, 45, 172-73, 217, 219-21, 377, 653
halts, 265, 370, 372, 522
hand, 8, 9, 11, 63
handles, 573, 616, 637, 645, 658, 666, 682, 686, 714

hard copy, 17, 48
hard disk, 4, 6, 31, 465
heap storage, 579, 581, 592, 619
HideAll, 189, 209, 213, 236, 243, 247, 253, 255-57, 262-63, 272-73, 279-80, 290-91, 310, 312, 342, 367, 373, 394, 400, 418, 429, 434-35, 442-43, 546, 654, 711, 715, 718, 721, 725
HideCursor, 724-25
HidePen, 688, 690
hotspot, 722, 724, 726

- I -

I-beam, 6, 14-15
icons, 44, 374, 376-77, 380-81
identifier, 7, 34, 64-65, 67-68, 70, 88, 95, 251-52, 261, 278, 282, 285-86, 324, 339, 341, 359, 393, 405, 417, 427, 466-67, 472, 533, 581-82, 617, 632,
if-then, 159, 208, 423-24
if-then-else, 138-39, 147-48, 150, 158, 162, 494
implementation, 98, 321, 323, 339-41, 351-52, 357-59, 363-65, 368, 575, 584, 633-34, 636, 659-60, 667
Implementation Section, 357, 381
in, 439
indentation, 8, 39, 53, 57
Include, 42-44, 66, 70, 74, 77, 83, 86, 98, 172
indirect recursion, 305-06
infinity, 184
information, 1-5, 17, 35, 43-44, 61-62, 66, 74, 102, 134, 138-39, 157, 166
information to the computer (*see also program*), 2
inheritance, 628, 667, 669-70
  multiple, 667, 670
inherited, 628-29, 642-44, 650-52, 656
InitCursor, 724
initial size, 239, 345
InitPort, 682, 734
inorder traversal, 605
input unit, 5
Insert, 10, 14, 45, 109, 190, 195-96, 199, 288, 345-46, 376-77, 473, 535, 542-44, 549, 574, 593-95, 598, 601, 603-7
insertion sort, 391, 421-23, 453, 457
InsetRgn, 730-731
instance variables, 628, 636, 643, 650, 668-72
Instant
  project, 50, 52, 344-45
  window, 16, 20, 45, 109, 194, 240, 377

interface, 32, 34, 36, 323, 338, 345-46, 349-51, 370, 632-34, 643
Interface.lib, 32, 34, 36, 345, 353, 381
invalid, 108, 115, 277, 467-68, 523
InvertArc, 694
InvertCircle, 196-98, 223-24, 236, 261-62, 675, 693
inverting, 196, 198
InvertOval, 203-5, 651, 692
InvertPoly, 712
InvertRect, 198-99, 201, 205, 236, 652, 692
InvertRgn, 709
InvertRoundRect, 693
italic, 704-5, 711, 738
iteration statement, 138

- K -

keyboard, 3, 5, 32, 65, 71, 78, 98, 112, 117-18
KillPoly, 712-13, 735

- L -

Law of Cosines, 123-24, 132
lazy input, 391, 440, 448
Length, 76, 78, 85, 87, 95, 98, 119, 124, 146, 214, 405, 421, 441, 446-49, 466, 476, 480, 490, 522, 533-34, 537-40, 542, 544-45, 547-49, 574, 702-3, 706
lexicographic ordering, 533
library, 32, 34, 42, 50, 66, 120, 123-24, 141, 153, 188-89, 209, 211, 229, 234, 250-51, 257-58, 324, 338-39, 341, 349-51, 360-61, 379, 428, 584, 674-75, 680-81, 693-94
  add a, 30
  predefined, 338, 360, 381
LightsBug, 44-45, 51, 338, 374-77, 380-81, 387-90, 628, 656-57, 669-70
Line, 2, 7-12, 20-21, 34, 43, 45, 66, 71, 75-79, 86, 95, 98, 119, 253, 264, 374, 440-41, 498-501, 523-25, 529, 531, 566, 568-71, 646-47, 675-76, 689-90, 695-96, 705-6, 712-13
LineTo, 192-96, 217, 220-21, 224, 236, 240, 242-44, 257-59, 689-90, 712-13
linked lists, 585, 602, 619
list head, 596, 600
ln, 124-26, 130, 133, 334

local
  identifiers, 282, 379
  variables, 285, 298, 315, 320-21, 331, 431, 476, 484, 579, 701
LocalToGlobal, 729, 731
log, 121
logical, 145, 148, 162, 173, 323, 339-40, 466-67, 716
longint, 83, 86-88, 98-99, 126-27, 154, 173-74, 200, 234, 265, 284, 296, 326, 366, 427, 446, 483, 535, 537, 550, 680, 684, 718, 720, 725
loop, 127, 138-45, 159-61, 163-65, 171, 203, 206, 223-24, 229, 285, 324, 421-23, 449, 470, 473, 582, 612, 678
  for, 254-55, 417, 470, 473
  in-test iteration, 139
  post-test iteration, 138, 144, 174-75
  nested, 159, 171

- M -

Macintosh
  character set, 533, 746
  file system, 13
  memory manager, 616, 619
  operating system, 314, 619, 674
  Pascal, 2, 4-9, 11-14, 16, 19-21
  Pascal program, 7, 9, 13-14, 19-20, 24,
  Pascal String Types, 532
main memory, 2-3, 5, 20, 203, 60, 465
Make utility, 30
mapping and scaling, 674, 729
mapping and scaling points, 729
MapPoly, 731
MapPt, 730
MapRect, 730
MapRgn, 731
maxint, 70, 101
maxlongint, 70, 101
memory, 2-5, 45, 71, 74, 95, 107, 203, 259, 264, 298, 360, 369, 372, 374, 421, 446-48, 465, 475, 573, 575-77, 583-85, 612, 614, 616, 637, 644-45, 657, 666, 679, 686, 707, 710, 712-14
  secondary, 3-5, 20
menu
  File, 13-14, 17, 32, 34, 48-50, 345-46, 349, 351
  options, 1, 3, 5, 8-9, 14, 19, 30, 37, 43-44, 50-52, 674
  Pascal, 8-9, 37, 51-52
  Run, 8-9, 11, 35-36, 41, 43-44, 48-51, 118-19, 206
  Windows, 16-18, 45, 119, 209, 217
messages, 62, 71, 75, 78, 211, 411, 419, 630, 638, 643, 647

Microsoft Word, 3
mod (*see also* modulo), 69, 108, 110, 112-13, 117-18, 127-54, 166, 170-71
modem, 499-500
modulo (*see also* mod), 112
monitor, 45, 51, 58, 119, 134, 189, 234, 249, 676
mouse, 5-7, 9-10, 12, 14-15, 20, 45, 188-90, 206-9, 211, 217, 219-21, 235-36, 677-78, 684, 687, 698, 700, 718-22, 724-25, 727-28
Move, 32, 38, 45, 348, 377, 396, 616-17, 637, 684, 689, 703, 713
MovePortTo, 688, 734
MoveTo, 192-236, 441, 443-44, 523, 684, 687, 689-721, 727

- N -

NewFileName, 501, 505, 526
NewHandle, 617-18, 621
NewRgn, 707, 710-11, 718, 726
nil, 575, 586, 596-97, 660, 720-21
nonrelocatable blocks, 617
null
  character, 76, 446, 449, 462
  string, 446, 534-35, 539-40, 557
number
  real, 83, 85-86, 126, 465, 577, 579
  integer, 108, 195-96, 199, 210
  ordinal, 83

- O -

Object List, 658-60, 662, 664-66, 671
object-oriented programming (*see also* OOP), 46, 628-29, 637, 668-69
ObscureCursor, 724, 727, 736
ODD(X), 121
OffsetPoly, 730-31
OffsetRect, 729, 731
OffsetRgn, 729, 731
OldFileName, 501, 506, 526
Omit, 541-42, 566-67, 644
OOP (*see also* object-oriented programming), 628, 634
open, 19, 22-23, 25, 32, 37-38, 41, 45-46, 119, 127, 189-90, 209, 213, 257, 259, 343, 345, 349, 370, 377, 399, 403, 408, 440, 466, 474, 484, 498, 522, 547, 675, 683-87, 696-97, 708, 710
OpenPicture, 713-16
OpenPoly, 712-13
OpenPort, 682-83, 688, 710, 717, 720, 725

OpenRgn, 708, 710-711, 713, 727, 735-36
operand, 106-07, 114, 127-28, 146, 173, 581, 641
operator, 106-16, 124, 127-30, 146-47, 155, 161, 171, 173, 278, 280, 574, 581, 583-84, 595, 641
  binary, 107
  precedence, 106, 113, 115, 128, 146-47, 161, 175-76, 454
  set, 438
  unary, 107-8, 581
ord, 131, 134, 556, 558, 562, 565, 706
ordered set of instructions (*see also* program), 2
otherwise, 148-49, 154-57, 160, 171-72, 278, 359, 372, 376, 393, 427, 489, 545, 603, 642, 709
outline, 7, 34, 47, 64, 198, 200, 314-15, 336, 344, 684, 687, 691-92, 704-5
output
  commands, 60, 74, 81, 99, 101, 467
  device, 5, 681
ovals, 188, 196, 203, 688-89, 691, 693, 707
overflow, 43, 87, 184, 594-96, 657
override, 253, 628, 630, 633-34, 642, 644-45, 647, 649-651, 669, 671

- P -

packed array of characters, 99, 391, 446-49, 535
Page, 24, 34, 153, 226, 231-32, 335, 394, 426, 457, 498, 501, 519, 527, 547, 649
page range, 17, 48
Page Setup, 14, 17, 19, 38, 50
PaintArc, 694
PaintCircle, 66, 72-74, 103, 196-98, 201-3, 223, 226, 236, 435, 693
PaintOval, 203-05, 236, 692
PaintPoly, 712-13
PaintRect, 198-99, 201, 205, 235-36, 691
PaintRgn, 709-10
PaintRoundRect, 692
palindrome, 571
paper copy, 17, 48
paper feed, 17, 48
parameters
  actual, 276, 327, 393
  formal, 266, 276-77, 405
  functional, 288
parentheses, 106-8, 114-15, 121, 161, 451, 632

Pascal
- operators, 106
- procedure, 250-53, 277, 371, 548, 625
- program, 1-2, 6-7, 9, 13-14, 17, 19-24
- menus, 8-9, 37, 51-52
- record, 427
- set, 391, 437, 500

Paste, 14-15, 19, 39, 50
patCopy, 681, 689, 716
patStretch, 680-681
pattern, 198-200, 205, 215, 234, 236, 294, 323-24, 442, 532, 540-42, 544-49, 680-81, 683, 688-94, 701-3, 709, 712-13, 716-17, 719, 728-729
Pause, 12-13, 22, 43-44, 78, 83, 141, 204, 217, 219-20
pen, 190, 192-96, 198, 205, 215, 258-59, 441, 681, 684, 688-94, 705, 708-9, 711-14
PenMode, 689-90
PenNormal, 689-90
PenPat, 689
PenSize, 190, 192, 195-96, 235-36, 435, 507, 684, 687-89, 699, 705, 711, 718, 721, 727
PenState, 688, 690
Pi (*see also* π), 70, 100, 124, 130-31, 290, 292
picFrame, 714
PicHandle, 713-15
PicPtr, 714
picSave, 680-81
picSize, 714
Picture, 189, 232-33, 681, 702-3, 713-16
picture handle, 714
pictures, 189-90, 192, 215, 441, 676, 680, 691, 709, 712, 716, 722-23, 729
pnLoc, 680-81, 688, 705, 735
pnMode, 680-81, 689-90, 692-94, 705
pnPat, 680-81, 689-94, 705
pnSize, 680-81, 689-90, 705
pnVis, 680-81, 688, 690
Point, 70-71, 74, 77-78, 83-84, 112, 158, 163, 192-94, 196-98, 203, 205, 207-09, 214-15, 217-21, 224, 227, 229, 231, 233, 251-52, 257-62, 264, 295, 298, 305, 324, 348-49, 351, 353, 359, 364, 372, 376-77, 405, 413, 424, 433-36, 441-42, 670, 675-78, 680, 686, 688-91, 695-96, 701-3, 705, 707, 709, 712-13, 717-19, 722, 724-25, 729-30
pointer, 464, 466-67, 469-73, 475, 482-84, 486, 489-90, 494, 498-501, 508, 516, 523-25, 545, 554-55, 559, 561-63, 554, 557, 561, 573, 575-83, 585-90, 593-96, 599-601, 603-7, 609-12, 614, 616-17
polyBBox, 712-713

polygon, 674, 681, 707, 712-13, 714, 729, 730-731
PolyHandle, 712-13, 730-31
polymorphism, 632, 639, 669
polyPoints, 713
PolyPtr, 713
polySave, 680-81
polySize, 713
pop, 579, 595-98, 601-2, 656, 660, 662-63
port, 680-87, 697, 699, 710-11, 714-18, 720-25, 731
portBits, 680-82, 688, 717, 719, 721
portBits.bounds, 681, 688, 717
portRect, 680-82, 688, 717
PortSize, 682
Pos, 537-38, 540-44, 549, 566-67
postorder traversal, 625
precedence level, 120, 146, 175
pred, 134, 164, 595
Preferences, 18, 20
preorder traversal, 625-26
Print, 7-8, 14, 17, 19, 30, 38-39, 48, 153, 254, 256, 264, 532, 552-53, 559-60, 563, 565-66, 569-72, 605-7
Print Using procedure, 532
Print Using Statement, 552
Printer, 3, 5, 17-18, 20, 22, 44, 48-49, 370, 464, 499-502, 510, 517-19, 674, 682
printing, 2, 17, 22, 38, 49, 182, 464
problem analysis, 61, 138, 145, 165, 167, 177
problem solving, 60, 307
Profiler, 338, 370-74, 379-82, 385-86
program
    executable portion, 66, 71
    library, 129, 188
    parameters, 65, 522
    statement, 66, 189, 251
    title. 7, 64-65
    unit, 253, 282, 285, 338, 341, 345-47, 349, 356, 370, 377, 382, 575, 594, 639, 641, 669-72
    window, 6-7, 9, 13-17, 19-21, 30, 34, 43, 50, 52, 66, 119, 189-90, 194, 196, 346, 375-76
project, 30-32, 34-36, 38-39, 41-43, 45-47, 49-57, 59, 87, 109, 171, 338-41, 344-49, 351, 353-54, 357, 360-61, 368-71, 374-75, 380-81, 383, 656-57, 668-69
    as an application, 42, 338, 368-69
    manager, 30, 50
    new, 41, 50, 344, 354
prompts, 75, 88, 95, 110, 234, 253, 268-71, 342-343, 410-12, 418, 471, 475, 546, 590, 607, 696, 700
PtInRect, 507, 678, 691, 719
PtInRgn, 709, 727
PtToAngle, 695
push, 594-95, 601, 660

put, 468-70, 475, 481, 483, 490, 581

- Q -

quality of print, 17, 48
queue, 573, 585, 598-602, 619, 671
QuickDraw, 257, 674-731
  libraries, 675
  mathematical foundation, 674-75
Quit, 14, 38, 50,
quotient, 108-9

- R -

RAM, 3-4, 20, 203, 369, 465, 579, 585, 674
Random, 126, 204-5, 324, 360, 425, 437, 467, 480-83, 522-24
range of values, 84, 94, 127
ReadString, 550-51
Rect, 199, 209, 235, 279, 290-91, 304, 342, 399, 405, 418, 429, 546, 559, 677-725
rectangle, 199, 647, 650-54
rectangles, 190-215, 674-721
RectInRgn, 709
RectRgn, 708
recursion, 294-98, 305-6, 605, 613
recursive functions, 294-96
reference, 64, 66, 70, 190, 264, 285-86, 359, 393, 409, 417, 428, 481, 575-76, 581, 591, 617, 637, 642, 644-45, 659, 662-66, 695, 707
reference count technique, 581
region, 678-729
relational
  expression, 139, 145-47, 533, 700
  operators, 146-47, 160-61, 438
remainder, 77, 108-10, 112-13, 117-20, 155, 545-47, 549, 586, 612
repeat-until, 138-39, 144-45, 152, 159-60, 164, 171, 229, 261
Replace, 16, 38, 41, 50, 158, 233, 254, 280, 290, 404, 417, 432-33, 482, 567, 574, 609, 707
reserved words, 64, 66, 340, 379, 744
Reset, 11-12, 43, 45, 223, 464, 466-501
Revert, 14, 19, 38, 50
rewrite, 268-69, 466-72, 475-79, 481, 522
rgnBBox, 679, 710, 718
RgnHandle, 680, 683, 707-11, 717, 719, 725
RgnPtr, 710
rgnSave, 680-81, 708
rgnSize, 679, 710
ROM, 3, 45, 360, 693
Root Class, 630, 633, 644-45
round, 126, 130, 214-16, 701, 703

Run, 8-9, 11, 109, 118-19, 206, 251
Runtime.lib, 32, 34-36, 345, 353, 360

- S -

SANE, 66, 97, 381, 559, 748-61
Save, 1, 13-14, 34-36, 38, 42, 44, 46, 251, 339, 345-46, 351, 464-65
Save a Copy As, 38
Save As, 34, 36
SaveDrawing, 713
ScalePt, 730
scientific notation, 77, 83
screen, 10-11, 13, 18, 32, 45, 48-49, 65, 188-92, 195-97, 201, 207-11, 221, 223, 229, 231, 233-36, 251, 259, 274, 310-15, 317, 321, 324, 327, 335, 676-77, 679-83, 685-86, 688-90, 693, 695, 700, 703, 706, 710, 714-18, 720-21, 724, 726-30
ScrollRect, 717, 719
Search, 14, 16, 37, 39-41, 50-51, 391, 421-22, 425-27, 433, 471, 481, 483-85, 488-89, 544-45, 547-48, 574, 591, 666
SectRect, 690, 734
SectRgn, 708
seek, 464-67, 481-85, 487-88, 490, 502, 509, 515-16, 523-24, 526-27
self, 628, 642, 644, 649-52, 659-60, 662, 664, 669
semicolon, 34, 39, 44, 66, 148, 154, 253-54, 328, 340, 405
set, 120, 127, 436-43, 445-46
Set Project Type, 42, 50
SetClip, 683
SetCursor, 724, 727-28
SetDrawingRect, 189, 209-10, 213, 216, 218, 220, 223, 226, 233, 235
SetEmptyRgn, 708
SetHandleSize, 617, 621
SetOrigin, 682, 686-88, 691
SetPenState, 688, 690
SetPort, 682-84, 686, 699, 702, 711, 715, 721, 731
SetPt, 687
SetRect, 210-11, 213, 215, 218, 220, 222, 224-25, 231-33, 678, 681, 684, 686, 690, 699-700, 702-3, 711, 715, 717, 720, 725, 732
SetRectRgn, 708
SetTextRect, 209, 211, 213, 216, 222, 225-26, 232
shadow, 704-5
Show Finger, 45
ShowCursor, 724, 727
ShowDrawing, 189-90, 207-35, 241, 675, 678
ShowPen, 688, 690

ShowText, 32, 36, 69, 74, 76-77, 81, 189, 209, 211, 213, 216, 222, 224-25, 227, 232, 470, 474, 477, 481, 485, 537, 546, 551, 559, 590
significant digits, 84
SIN(X), 121, 289, 291
single, 16, 30, 39, 47, 50, 77-78, 92, 95, 108, 127, 192, 207, 341, 349, 362, 376, 391, 436, 446, 448-49, 475-76, 534, 542-44, 568, 580-81, 632, 667
size box, 209, 211
Source Options, 39
Special, 66, 72, 108, 112
spExtra, 680-81, 704-5
srcOr, 681, 705, 716-17, 719
stack, 297-98, 303, 360, 418,573, 579, 583, 585, 592-94, 593-98, 602, 619
Step, 9, 12, 20, 31-32, 34-36, 43-45, 61-63
Step-Step, 9, 12, 20, 43-44
StillDown, 206-7
Stops In, 10-12, 17, 45
Stops Out, 11
storage
  space, 616
  free, 579
string, 16, 39-41, 69-70, 84-85, 95, 465-67, 497-98, 532-63, 573-74, 585, 588, 591
  functions, 542, 544, 549
  maximum length, 95
  miscellaneous routines, 550
  procedures, 535, 537, 544
StringOf, 550-51, 566-67, 699-702
StringWidth, 702-3, 705
structure chart, 267-68, 275
stubs, 310-11, 315
StuffHe, 728-29
Style, 217, 559, 680-81, 704-5
StyleItem, 704
subclass, 629-33, 637-39, 641-45, 657
subordinate modules, 370, 372, 410
subrange type, 83, 94, 154, 163, 165
subscript, 393-94, 403, 410, 416, 427, 440
subscripted variable, 393-94, 403-4, 409, 416-17
succ, 90-92, 101, 163-64, 397-98, 401-2, 406-7, 444, 447, 449, 495, 595
superclass, 630-32, 642, 647, 649, 657
superordinate module, 309, 372
symbol table, 285, 288, 359
syntax
  ambiguity, 158
  errors, 11, 79, 109, 203, 345-46, 351, 353
  grammar, 2
  of a program, 19

SysBeep, 485, 487, 490, 678, 700

- T -

TextFace, 684, 687, 705, 711
TextFont, 317, 705
TextMode, 705
TextSize, 317
TextWindow, 501
TMON, 45
TObject, 644-45, 647, 653, 664
trace a program, 119
trace table, 118, 139, 142-43, 422, 425
transfer bits, 674
Transfer Modes, 690, 716, 719
translator, 164, 171, 253, 285-86, 288, 359, 446, 617
trigonometric functions, 121, 695, 701
trunc, 126, 231-33, 236, 556, 562-63, 701-3
truth table, 88
TUTOR SYSTEM, 362, 369-370
txFace, 680-81, 704-5
txFont, 680-81, 704-5
txMode, 680-81, 704-5
txSize, 680-81, 704-5

- U -

underflow, 595-96, 598, 602, 663
underline, 704-5, 711
UnionRect, 690
UnionRgn, 708
User Interface Toolbox, 674
uses clause, 43, 64, 66, 189, 197, 338-41, 343, 355-61, 370, 372

- V -

valid statement, 115
value parameters, 259, 415
variable parameters, 264, 266, 583
variables, 45, 60-61, 66, 70-72, 75-79, 85, 88-98, 108, 111, 114-15, 118-120, 128, 188, 214, 250, 252-54, 259, 264, 276, 278, 282, 284-85, 298, 338-40, 343, 349, 379, 391, 393, 395-96, 408, 410, 428, 431-33, 437, 446, 545, 547, 550-51, 564, 575-76, 579-80, 594, 628-34, 675, 697, 701-2
variables, values given to, 71
variant records, 436
VHSelect, 436
Views Options..., 348

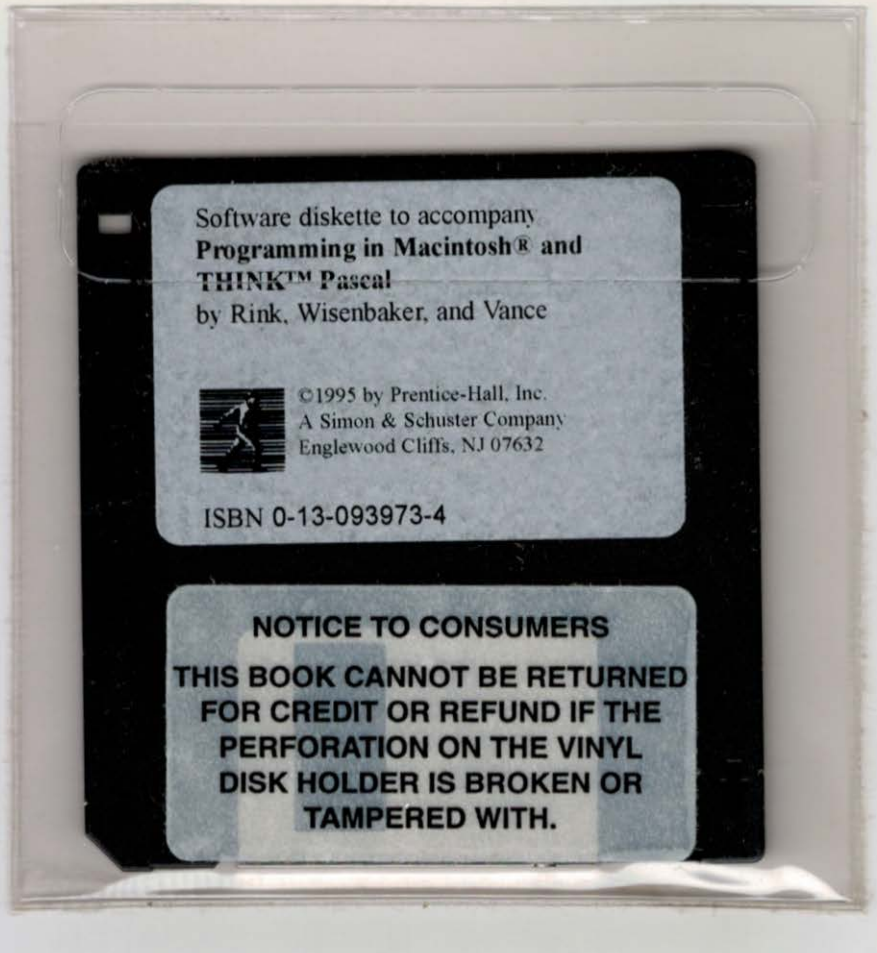
visRgn, 680-82, 686, 688, 717-19, 721
volume name, 349, 467, 499
von Neumann machine, 2

- W -

What to Find, 16, 19
while-do, 507, 517, 588, 590, 698
window, 3, 6, 12, 16-19, 30, 36-39, 44-46, 49-50, 109, 119, 188-89, 194, 196, 206, 209-11, 213, 215, 217-18, 220-24, 226, 231-32, 342-43, 370, 374, 377, 646, 650, 653-54, 656, 674-75, 683-85, 696, 711, 715, 718, 720-21, 725
  Observe, 16, 20, 45, 51, 54, 106, 119-20, 129, 134, 136, 139, 141, 331
  Text, 7-8, 10, 16-18, 35, 44, 71, 74-76, 78-79, 82, 439, 469, 471-74, 480, 482, 488, 546-47, 578, 582-83, 638
with statement, 431-33
word processing, 3, 5, 532
WriteDraw, 192, 196-97, 206-7, 214, 216-220, 231, 233-36, 441, 444, 675, 700
writeln, 8, 11, 32, 36, 47, 52-54, 56, 60, 69, 71, 74-79, 81-82, 85, 87, 89-90, 99, 107, 109-116, 124, 126-28, 464, 466-67, 471, 473-75, 480, 482, 485, 487-490, 497-501, 523-24
writeln( ), 76
writing to Pascal files, 464

- X -

XorRgn, 709, 716-17



Software diskette to accompany
Programming in Macintosh® and THINK™ Pascal
by Rink, Wisenbaker, and Vance

ISBN 0-13-093973-4

NOTICE TO CONSUMERS

THIS BOOK CANNOT BE RETURNED FOR CREDIT OR REFUND IF THE PERFORATION ON THE VINYL DISK HOLDER IS BROKEN OR TAMPERED WITH.

DISK INCLUDED

ISBN 0-13-093873-4
90000
9 780130 938732