

Complete Macintosh Turbo Pascal

Joseph Kelly

Foreword by Philippe Kahn

**Scott,
Foresman
Macintosh Computer
Books**



MACS

Complete Macintosh Turbo Pascal

Complete Macintosh Turbo Pascal

Joseph Kelly

Scott, Foresman and Company
Glenview, Illinois London

Cover photo courtesy of Apple Computer, Inc.

Apple® is registered to Apple Computer, Inc.

Macintosh™ is a trademark licensed to Apple Computer, Inc.

Turbo Pascal® Macintosh is a trademark of Borland International, Inc.

Turbo Pascal® Tutor Macintosh™, Turbo Pascal® Toolbox Numerical Methods Macintosh™, and Turbo Pascal® Database Toolbox Macintosh™ are trademarks of Borland International, Inc.

TMON is a trademark of ICOM Simulations, Inc.

Library of Congress Cataloging-in-Publication Data

Kelly, Joseph.

Complete Macintosh Turbo Pascal / Joseph Kelly.

p. cm.

Bibliography: p.

Includes index.

ISBN 0-673-38456-X

1. Macintosh (Computer)—Programming. 2. Pascal (Computer program language) 3. Turbo Pascal (Computer program) I. Title.

QA76.8.M3K45 1989

005.265—dc19

88-26876

CIP

1 2 3 4 5 6 KPF 94 93 92 91 90 89

ISBN 0-673-38456-X

Copyright © 1989 Scott, Foresman and Company.

All Rights Reserved.

Printed in the United States of America.

Notice of Liability

The information in this book is distributed on an “As Is” basis, without warranty. Neither the author nor Scott, Foresman and Company shall have any liability to customer or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by the programs contained herein. This includes, but is not limited to, interruption of service, loss of data, loss of business or anticipatory profits, or consequential damages from the use of the programs.

Scott, Foresman professional books are available for bulk sales at quantity discounts. For information, please contact Marketing Manager, Professional Books Group, Scott, Foresman and Company, 1900 East Lake Avenue, Glenview, IL 60025.

This book is dedicated to my daughter, Sarah Catherine

Foreword

Philippe Kahn

Turbo Pascal for the Macintosh has been a very successful product for Borland International. The better to support this popular compiler, Borland has teamed up with Scott, Foresman and Company to provide a complete and concise Turbo Pascal tutorial that bears the names of both companies.

Complete Macintosh Turbo Pascal is the result of hundreds of hours of work, not only by the author, but also by our technical support personnel at Borland. Our technicians have critiqued every page of the text and have tested each of the programs presented.

This book is an excellent tutorial for all users of Macintosh Turbo Pascal. The book's organization allows the beginner to read it from cover to cover, while more knowledgeable users can start with Part II or III for coverage of advanced topics. Each chapter concludes with a quiz that reinforces the items covered.

All Macintosh-style programming topics are discussed, including menus, dialog boxes, mouse programming, graphics, sound, event-handling, resources, and more. An entire chapter is devoted to debugging strategies to help users at all levels solve their programming problems. The author presents many complete application programs that are useful in day-to-day programming activities, including an album database and a date/occasion reminder program.

When you have completed reading *Complete Macintosh Turbo Pascal*, you will have enough knowledge of our powerful compiler to create sophisticated programs which utilize the Macintosh environment. Borland is pleased to endorse *Complete Macintosh Turbo Pascal* as the "official" book on Turbo Pascal for the Macintosh.

A handwritten signature in black ink, appearing to read 'PKahn', with a long horizontal line extending to the right.

Contents

	Introduction	xv
Part I:	AN INTRODUCTION	1
Chapter 1	The Fundamentals	3
	What Is Programming?	3
	What Is Pascal?	4
	What Is Syntax?	6
	What Is Programming Technique?	6
	Flowchart Symbols	8
	How Turbo Pascal Communicates with Macintosh	12
	Machine Language	14
	Macintosh Storage	14
	Getting Ready to Use Turbo Pascal	18
	Starting Turbo Pascal	20
	Review Summary	23
	Quiz	23
Chapter 2	Getting Acquainted with Turbo Pascal	24
	The Turbo Pascal Screen	25
	Menu Discussions	25
	Your First Turbo Pascal Program	44
	Running Your First Program	47
	Editing Your Program	48
	Saving Your Program	50
	Printing Your Program	52

	Review Summary	54
	Quiz	55
Part II:	THE TURBO PASCAL LANGUAGE	57
Chapter 3	Turbo Pascal Revealed: Structure and Syntax	59
	The Right Stuff: Syntax	60
	The Elements of a Turbo Pascal Program	60
	Data Types: What to Declare	66
	What to Declare: Variables	72
	Assigning Variables	74
	Constants	75
	A Word on Punctuation: The Semicolon	76
	Expressing Yourself	77
	Mathematical Order of Operation	78
	Simple Turbo Pascal Arithmetic	79
	More on Data Types	82
	Program Formatting	86
	Review Summary	86
	Quiz	87
Chapter 4	Turbo Pascal Statements	88
	The Write and Writeln Statements	89
	Assigning Data Values with Read and Readln	97
	Statement of Choice: The Conditionals	103
	Decisions, Decisions . . . The Thinking Mac	103
	A Couple of Common Errors with If Statements	106
	Nesting Your If Statements	108
	Boolean Operators	110
	Another Case to Consider	110
	Review Summary	112
	Quiz	112
Chapter 5	More Statements: The Looping Structures	113
	Programs That Repeat	114
	The While Statement	114
	Counting Your Loops	116
	Loops That Sum	118
	The For Statement	119
	Backward Looping with Downto	125
	Summing up For Loops	126
	The Repeat Statement	126
	“While” versus “Repeat . . . Until”	129
	Nested Loops	129
	One More Loop: The Goto Statement	130
	Review Summary	132
	Quiz	132

Chapter 6	Procedures, Parameters, and Units	134
	Structured Design	135
	What Is a Procedure?	135
	What Does a Procedure Consist Of?	136
	Where Is a Procedure Placed?	137
	How Are Procedures Used?	137
	Using Variables with Procedures	139
	Global versus Local Variables	140
	Passing Information with Parameters	141
	Variable Parameters	141
	Value Parameters	145
	Introducing Functions	148
	Compiler Directives	151
	The Turbo Pascal Unit	157
	The Uses Clause	158
	PasConsole Information	159
	Using UnitMover	160
	Review Summary	162
	Quiz	164
Chapter 7	Turbo Pascal Library Features	165
	How to Avoid Reinventing the Wheel	165
	The Central Library	166
	Library Routines	167
	Review Summary	178
	Quiz	178
Part III:	APPLICATIONS AND ADVANCED CONCEPTS	179
Chapter 8	Programmer's Corner	181
	The Entire Picture: Complete Pascal Programs	181
	Metric Conversion Program	181
	Guess-a-Number Program	186
	Decimal-to-Hexadecimal Conversion Program	191
	Tape Counter Program	199
	Review Summary	201
	Quiz	201
Chapter 9	Advanced Data Structures	202
	Advanced Numeric Types: Long Integers and Extended Real Numbers	202
	What's in a Type?	204
	Simple Arrays	206
	Parallel Arrays	208
	Records and Files	210
	Sets	211
	Using Arrays, Records, and Sets in an Application	213

	Review Summary	216
	Quiz	217
Chapter 10	Introduction to Advanced Concepts	218
	Recursion; or, Can I Call Myself?!	218
	The Pointer and Handle Data Types	221
	Almost Everything in Life Has a Point(er)!	225
	Further Study—Stacks, Queues, and Trees	230
	Review Summary	234
	Quiz	234
Chapter 11	More on Records and Files	235
	Fields, Records, and Files	235
	Files versus Arrays of Records	236
	File-Handling Library Procedures	237
	Using a File in a Phone-Book Program	240
	Sorting and Merging Records	245
	Variant Records	251
	Review Summary	253
	Quiz	254
Chapter 12	Debugging and File Analysis	255
	Debugging	255
	File Analysis	256
	Encipher/Decipher Program	263
	MacsBug	266
	TMON	268
	Review Summary	269
	Quiz	269
Chapter 13	Graphics, Sound, and Resources	270
	Graphics	270
	Turtle Graphics	271
	Standard Macintosh Graphics	275
	Fun with the Mouse	281
	Making Music in Turbo Pascal	285
	Resources	290
	Using RMaker	293
	Event-Handling Programming with a Resource File	293
	Review Summary	299
	Quiz	300
Chapter 14	A Few Programs for the Road	301
	The Date-Minder Program	301
	Batting-Average Program	308
	Record Album Database Program	314
	Summing Up	323

Review Summary	323
Quiz	323
Appendix A: The Borland Toolboxes and Turbo Tutor	325
Appendix B: Reserved Words	329
Appendix C: Quiz Answers	330
Appendix D: Bibliography	336
Index	337

Introduction

Welcome to the world of Turbo Pascal programming on the Macintosh! We're glad you chose this book to learn about this powerful language. This book is structured so that the complete programming novice can use it to learn how to write sophisticated programs on the Macintosh. However, *Complete Macintosh Turbo Pascal* is not for beginners only! Experienced programmers will find a great deal of information in this book including coverage of resource files, units, event-handling, graphics, sound, etc. For the complete beginner to programming, we recommend that you start from the beginning of the book and carefully study each section. For those of you familiar with programming in another language (e.g. BASIC), we recommend that you start with Part II: The Turbo Pascal Language. Finally, for those of you who have used Turbo Pascal on the IBM PC or just want to know about Macintosh-specific areas of Turbo, we recommend that you skip to Part III: Applications and Advanced Concepts.

No matter what your current level of programming is, we sincerely hope you will come to enjoy the full power of Borland International's Turbo Pascal package for the Macintosh and that you will keep *Complete Macintosh Turbo Pascal* handy as a reference guide when you have become a Turbo "master."



AN INTRODUCTION



The Fundamentals

What Is Programming?

What Is Pascal?

What Is Syntax?

What Is Programming Technique?

Flowchart Symbols

How Turbo Pascal Communicates with Macintosh

Machine Language

Macintosh Storage

Getting Ready to Use Turbo Pascal

Starting Turbo Pascal

Review Summary

Quiz

This chapter is an overview of the fundamentals required for effective use of your Macintosh with Turbo Pascal. You will learn many computer buzzwords and the meanings of this computer jargon, referred to as *computerese*. Basic computing concepts are introduced, defined, and illustrated. As we progress through this chapter, we define the concept of computer programming in the language Pascal.

What Is Programming?

Your Macintosh is a wonderful machine. However, it is just a machine. Computers, like other machines, must be told what to do. This task is accomplished by giving the computer specific instructions. A set of specific instructions, sequentially arranged, is called a *program*.

A program is also referred to as *software*. Software is something that you cannot touch. It manipulates the information, or data, within the computer. Your Macintosh computer, or the physical components that you can touch, is known as *hardware*.

But just what is programming? Programming is creating a set of specific instructions to perform a specific task. The answer to the aforementioned question is academic; however, the concept of programming has been around for a long time. Think about it for a minute. Have you ever followed a recipe

or assembled a bicycle using a set of instructions? Or perhaps solved a long mathematical problem with a step-by-step approach? These are all examples of sequenced instruction.

You may have noticed that many sets of instructions, such as those for assembling stereo components, are presented in both English and foreign languages. This is done for the sake of those who do not understand English. Your Macintosh does not understand English, but it must be able to understand instructions. You, as its programmer, can write these instructions, or *programs*, for the computer. This communication is accomplished much the same way people communicate: by working with a language. A language is a systematic means of communicating with symbols. A computer program is a systematic set of symbols that are understood by the computer, and a computer language is a written set of instructions, a program itself.

What Is Pascal?

People may use various languages for communication, such as English, German, French, or Spanish. Computers, too, use various languages for communication. Some of the languages used in computing are BASIC, FORTRAN, COBOL, and Pascal. Like many languages available to people, there are additional languages used by various computers. The use of grammar and the depth of vocabulary are the principle differences between a computing language and a natural language such as English. This book is written with the purpose of using your Macintosh with one language—Pascal.

Pascal was developed in 1968 by Niklaus Wirth as a teaching tool for programming concepts. Pascal is simple yet complete enough to illustrate good programming techniques.

A Look at Programming Languages

Early programming languages were either too limited or too difficult to learn. Like Pascal, many such languages were designed for a specific purpose. An early language for scientific computation is called *FORTRAN* (FORmula TRANslator). FORTRAN is still in wide use today, but it is too complex for the beginner.

BASIC (Beginner's All-purpose Symbolic Instruction Code) is the result of an attempt to develop a language that was easier to learn than its predecessors. Developed in 1964 at Dartmouth College, BASIC is the most symbolic of the natural languages. Its instructions are analogous to English. BASIC has become the most popular language among microcomputer users because it is simple and easy to use. BASIC, however, has its limitations and is not recommended for long, complex programs.

C, another very popular language because of its flexibility, power, and portability, was written by Dennis Ritchie at Bell Labs. Because it allows the programmer to get more involved with lower-level machine operations, it is the choice of many software development organizations. C allows a programmer to write and run a program on one computer, take the same code to another, and execute it with only minor modifications, if any.

COBOL (COmmon Business Oriented Language) was developed in 1959. COBOL is used widely in the business community because of its ability to handle large, complex problems. A useful language that fulfills the need of manipulating large amounts of information, COBOL lacks the simplicity required of a first language.

Turbo Pascal: A Short History

Pascal, named after the French mathematician Blaise Pascal, is often classified as a flexible and simple language for learning good programming techniques. Actually, a number of implementations of Pascal exist today. The language developed by Wirth, sometimes called ANSI (American National Standard Institute) standard Pascal, has been enhanced with new features and modified to take advantage of the environments from which these features operate. One version, UCSD Pascal, developed at the University of California at San Diego, is widely used with microcomputers. Another popular version is Apple II Pascal, a dialect similar to UCSD Pascal.

If you have shopped around, you have undoubtedly discovered that most high-level compilers for microcomputers are in the hundreds of dollars. These “serious” systems offer packages with everything you need to develop just about any application you could imagine. Fortunately, our good friends at Borland International Inc. saw the need for good, solid compilers at more reasonable prices and have set the standard with such products as Turbo Pascal, Turbo BASIC, and Turbo C for the IBM PC. These products are all excellent development packages for only a fraction of the prices of other similar compilers. In addition, these compilers are so fast and efficient that the name Turbo is truly appropriate.

Borland has taken its tremendously popular IBM PC-based Pascal compiler package and developed a similar product for the Macintosh: Turbo Pascal Macintosh. Along with all the speed and flexibility of the MS-DOS version, the Macintosh product offers all the Mac-specific user interfaces a Macintosh programmer would expect. As you may know, it is common for microcomputers to use a specific dialect of a language, much the same way as people use different dialects of the same languages. The differences between dialects are generally small, and the programmer is able to convert programs from one dialect to another as long as the rules of syntax are followed.

What Is Syntax?

An English teacher drills students on proper sentence structure, spelling, punctuation, and other rules. The teacher emphasizes the rules that are required for proper grammar. These rules, taken collectively, are known as *syntax*.

People can communicate with one another without using proper syntax. A friend may find it amusing if you use an improper statement during a discussion. Your friend may even insult your intelligence but will most likely understand your meaning. Your Macintosh, on the other hand, will not interpret your intentions; it won't laugh or even ask you what you meant. It displays an error message like the one shown in Figure 1.1.

Fig. 1.1.

Error 1: ';' expected.

and continues to display the message until the error is properly corrected.

Therefore, communicating with your Macintosh requires language instructions that are syntactically and logically correct. Many syntax rules in Turbo Pascal are a simple extension of symbols that you are already familiar with. For example, you use the plus sign (+) for addition and the minus sign (−) for subtraction. A hand calculator and an adding machine use the same symbols for simple mathematical calculations. A few symbols, however, are not customarily used for similar calculations. Unlike the calculator, TURBO Pascal uses the asterisk (*) for multiplication instead of the commonly used times symbol (×).

Most programmers don't memorize the entire list of syntax rules for a specific language, as this can be a very taxing chore. Don't be afraid to use a table or reference as a programming aid. With practice you will learn to use proper syntax and programming technique.

What Is Programming Technique?

A buzzword associated with Pascal is *structure*. What is a structured language? Let's answer this question by defining a programming scenario for cooking a pot of chili.

Suppose you just finished reading the great American novel about the Old West and have developed a craving for something representative of the era. Embarking upon the fulfillment of your desire, you decide to make some chili. To start you must purchase the appropriate foodstuffs (meat, tomatoes, kidney beans, onions, chili peppers, and so on) from your local grocery. Next,

select the proper cooking utensils. You mix the ingredients and cook them over a slow heat. Finally you get to eat.

I have just defined a problem. Although the task defined is not a difficult one and is more likely to be solved in the kitchen than in the computer room, it serves to make several points. First the task is defined: to prepare chili. Next the problem is broken down into a set of subproblems: purchasing food, selecting pots and pans, mixing the ingredients, and cooking them over slow heat. The process of taking a large problem and breaking it down into a set of related subproblems is called *top-down design*. A program using top-down design is called *structured*.

In this example I can further subdivide the individual problems, such as cooking beef, adding tomatoes, stirring, adding beans, and stirring again, until the individual tasks represent the basic building blocks of our subprogramming modules. In Turbo Pascal these basic blocks can be solved individually, accomplishing a single task or subproblem (mixing, cooking, eating) until the entire macro problem is solved, which is chili prepared and hunger removed. This structured nature of problem solving (top-down design) and flexibility (working on smaller subproblems independently) is what makes Turbo Pascal so appealing as a language to teach good programming techniques. The best advice that we can give is don't try to solve the macro problem all at once. First define the problem. Break the problem down into its basic building blocks and then solve them individually. When you are done, the entire problem is solved.

The entire process of problem solving using basic building blocks of Pascal instruction enables the programmer to define a complex problem and then to develop the step-by-step solution to the defined problem. The acquired solution is called an *algorithm*. Thus, the loosely defined example of preparing chili results in an algorithm for satisfying the craving for Western food. More important, we have defined several important concepts.

The Implementation Phase versus the Problem-Solving Phase

When writing an algorithm, start with a little planning, called the problem-solving phase. In this phase the problem is analyzed and a general solution is designed. Approach a problem by breaking it down into separate but related tasks. To aid in this process you may use a pictorial graphic design called a *flowchart*.

There are many applications of the flowchart. For example, diagrams and graphs are used in business to depict the flow of information. A road map is

used to chart a traveler's destination from point A to point B. The mapped route a traveler selects is a flowchart.

Flowcharts are often used by programmers to develop very complex programs or a set of related programs. A set of related programs, called a *system*, together performs a complex task. An inventory control program and a payroll program used together may function as an accounting system. Flowcharts are generally helpful for complex problems but are not necessary to solve simple problems. However, a novice programmer will find many simple tasks difficult at first. In addition, a programmer develops good techniques and an understanding of programming logic by using the flowchart as a road map to the desired task.

Flowchart Symbols

This discussion introduces the symbols (or road signs) of a flowchart. The symbols are standard and are used for problem solving across several programming languages. With practice you will become familiar with the flowchart road map and its use.

The following is an explanation of common flowchart symbols and their use. If the terminology is difficult to understand, don't worry. The concepts will be explained in detail in the remaining chapters. This book uses the standard flowchart symbols shown in Figure 1.2.

The flowchart symbols and their processes are summarized as follows:

The *terminal* symbol is used to start or end a program.

The *process* symbol indicates the processing of information, such as a mathematical calculation.

The parallelogram represents an *input/output (I/O)* symbol. Input is the data or instruction that you enter via the keyboard or mouse. Output is the result of the processing of that information; it may be in the form of characters on a screen or perhaps *hard copy*, a printout. Therefore, input/output is the process of putting data into and getting information out of the computer.

The I/O Symbol

The I/O symbol may take either of two common forms. A common input form is the Read function. *Read* copies information from the storage medium (disk) to main memory (the internal memory of your Macintosh) much as a cassette recorder reads audio tones from a prerecorded cassette tape. (If you read something from memory, you are accessing or loading prerecorded data.) The Read function is used to put data into a program. To output information

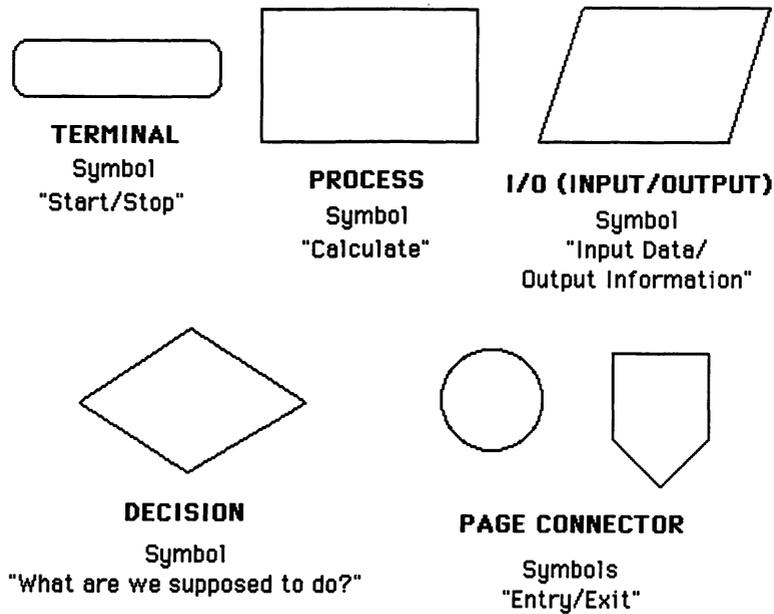


Fig. 1.2. Flowchart symbols used to graphically depict programs.

to a screen or printer, we use the second I/O symbol, called *Write*. (In other languages, such as C, the output command may be different, for example `Printf`). Let's look at an example.

A mathematical operation that adds two numbers and then prints the sum is broken down into steps by a flowchart. Figure 1.3 illustrates this calculation.

The flowchart shown in Figure 1.3 depicts a simple operation to add two numbers and then write the result. The problem could have been easily solved without the aid of a flowchart. However, the flowcharting principles will become apparent in more complex examples as they are introduced in later chapters. Remember, it's important to develop an understanding of programming logic through good programming practice. As a beginning programmer, you should use flowcharts to help you reach your goals.

It should be noted that there may be more than one solution to a programming problem. Therefore, there may be more than one correct flowchart—one for each corresponding solution, or algorithm. People often have the opportunity to select from alternative solutions to problems they encounter every day. Similarly, a programmer may select a solution that he or she considers the optimum answer. The flowcharts in this book are representative of several possible solutions. You are encouraged to experiment with the programs and develop alternative solutions.

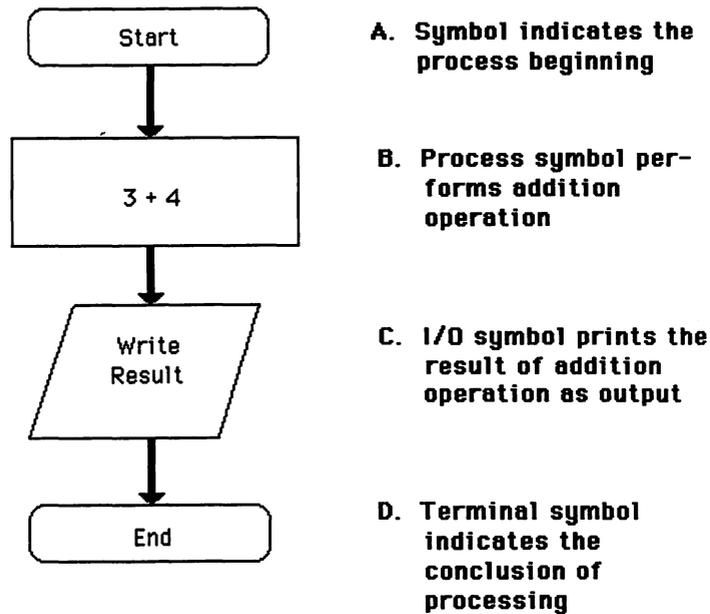


Fig. 1.3.

When the I/O symbol is used for entering information into a program, the program reads data stored in memory and then performs the processing instruction. This process is analogous to the brain reading data stored by the memory.

For example, you may want to perform a mathematical calculation using the data stored in memory and then write the result. The flowchart in Figure 1.4 depicts this example.

The Decision Symbol

The decision symbol permits flexibility when using a flowchart. The computer can make logical decisions, such as whether the first number is larger than the second, whether this is the last number to read, whether two pieces of data are equal. The diamond-shaped symbol indicates this decision and requires the computer to choose yes or no. A simple decision symbol is shown in Figure 1.5.

The exits from a decision symbol represent a *conditional*. These conditional branches flow to another area of the flowchart according to the decision, like the if-then equations you learned in beginning geometry. The decision symbol can use different exit points, as depicted in Figure 1.6.

The arrows of the flowchart generally run top to bottom or left to right. Crossing flowlines is not acceptable. To avoid crossing flowlines, use the

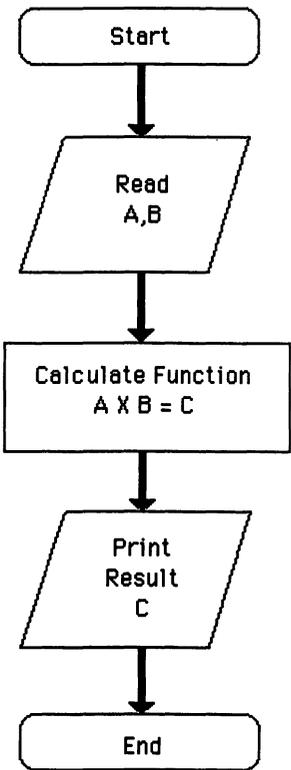


Fig. 1.4.

connector symbol. A second connector symbol, the off-page connector, is used for flowcharts covering more than one page. These symbols indicate the transfer of control from one area of the flow to another. This principle is applied in modular programming, in which the problem is split into subprograms, or *units*. The majority of flowchart examples used in this book will require a single page.

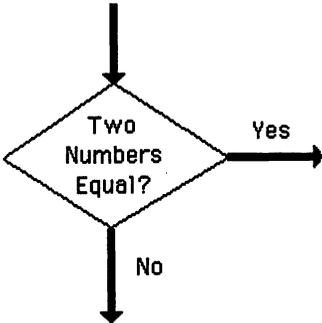


Fig. 1.5. The Decision Symbol.

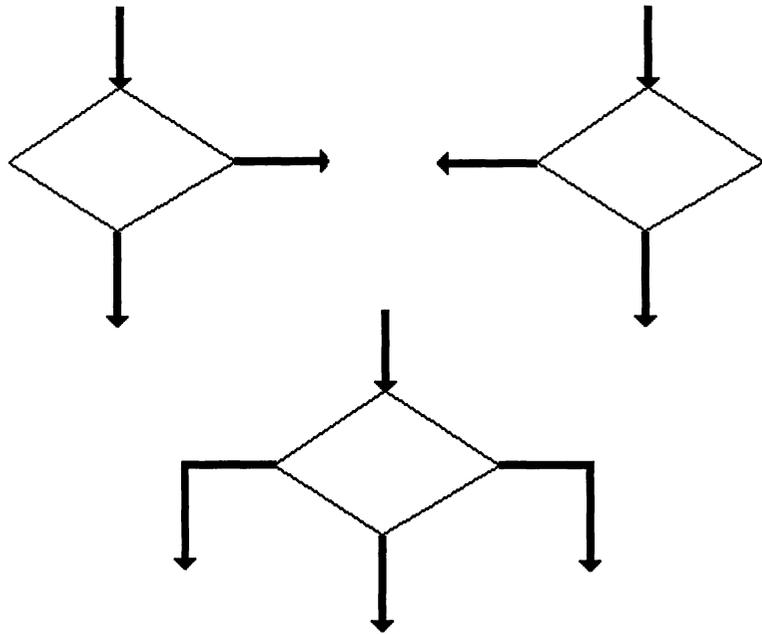


Fig. 1.6. Exit points of the decision symbol.

Remember, memorizing the symbols of a flowchart won't make you a good programmer, nor will memorizing the road signs make you a good driver. The flowchart is simply a tool to be used in pictorially defining programming logic, thus the first step in understanding programming logic.

How Turbo Pascal Communicates with Macintosh

This section introduces you to a few more computerese terms. Many people who purchase the Macintosh are not interested in how the computer understands commands or how it works internally. If you are one of these people, skip to the next section.

Do you need to understand Mac's technology to use it? The answer is no; however, the knowledge you gain will enable you to converse more intelligently with other Macintosh enthusiasts about such topics as compilers versus interpreters, memory, bits, bytes, K, RAM, and ROM. We invite you to speak a little computerese.

Compilers versus Interpreters

Your computer doesn't understand Turbo Pascal. Pascal is a foreign language to Macintosh and requires a translation into a language that Macintosh can understand. Much as a foreign language interpreter translates German to an Englishman, a Pascal interpreter makes available the translation of Pascal into Mac's language—machine language.

The Turbo Pascal compiler is a program that is fixed and that resides in external storage on the Turbo disk. When you enter Pascal code and select to compile it, Turbo reads your code and translates it into Macintosh machine language. The Turbo Pascal compiler is much more than a messenger service, though; it is also an editor as well as a linker. As an editor, Turbo allows you to change or modify your programs much the same way you would change a MacWrite document. As a linker, Turbo allows you to pull in other subprograms or units so that you can develop your applications in a modular form. This linker then combines all the code needed into the final executable program.

Most implementations of Pascal are translated using a compiler instead of an interpreter. An interpreter translates your Pascal code as it is being executed and unlike a compiler is not capable of producing stand-alone double-clickable applications. (In the Macintosh environment, a stand-alone application—one that does not need an interpreter or similar program to execute—is said to be *double-clickable* because its icon may be double-clicked—two rapid clicks with the mouse button—and the program will be started.) Another difference between a compiler and an interpreter involves the execution speed of a program. A compiler translates the entire program into machine language before any execution begins, whereas the interpreter must interpret each statement before it is executed. The advantage of compiler execution is that it is much faster than interpreter execution. However, the disadvantage is that the programmer must wait until the entire program is compiled before any error is identified regardless how small the error may be. For example, in the earlier versions of MacPascal, an interpreter from THINK Technologies, if the programmer entered in a misspelled reserved word, an error was signaled before the program was interpreted, whereas in Turbo Pascal, you wouldn't find this error until you tried to compile it. However, with the very fast compilation speed of Turbo Pascal, you can enjoy the best of both worlds; many other Pascal compilers cannot compete for speed with Turbo.

Turbo is well suited for both the novice and advanced programmer. Borland boasts the ability to compile more than 12,000 lines per minute, although these timings aren't realized with small programs because of the overhead in

starting up the compiler. Nevertheless, the Turbo Pascal package is easy to use yet powerful enough to develop the most sophisticated programs for the Macintosh.

Machine Language

What is machine language? Machine language is the only language a computer understands without translation. Machine-language programming is programming at its lowest level. Instructions and data are sets of numbers in binary form (binary numerals are 0 and 1). The binary digit is the fundamental unit used by your Macintosh computer; a single binary digit is called a *bit* (BInary digiT). Programs in binary form consist of sets of zeros and ones, very tedious and difficult to understand.

Machine language may be translated into a more readable format using a *disassembler*. A disassembler translates binary numbers into *assembly language* instructions. Assembly languages differ from one machine to another depending upon the *microprocessor* used in the computer. The Macintosh uses the Motorola-based 68000 family of microprocessors, and so the assembly language it uses is called “68000 assembly language.” The IBM PC family uses the Intel-based 8088/8086 microprocessors, hence “8088 or 8086 assembly language.”

Even this level of instructions is very difficult for the novice programmer to understand. In general, many assembly language instructions are required to make up just one Pascal statement. This is primarily because the assembly language instructions are very limited in operation (adding two numbers or moving data from one location in memory to another), but when several are combined, just about anything can be accomplished.

Macintosh Storage

One of the most powerful features of your Macintosh is its ability to store information. Computer memory may appear to be an abstract term, since many of us associate memory with the human brain. Unlike the human brain, however, the memory of your Macintosh comes on little silicon chips.

The Macintosh memory is composed of a series of electrical switches. When a switch is on, it is a 1, and when off, a 0. The pattern of 1s and 0s expresses the data. The status of each binary location is either 1 or 0, on or off, yes or no. Let's say you just wrote a program that allows the user to enter the names of all the members of his or her family. You need a *flag* to indicate when you are finished entering names. A flag is a location in memory that

in the simplest case indicates whether something is true or false. This flag is represented by a slot in memory that contains either a 1 or a 0. If the memory slot is set to 1, it means the user is finished; a 0 means you have more names to enter.

The use of bits is not limited to representing flags. Several bits can be grouped so that other values are represented. The use of bits to represent characters such as the letter 'A' requires a standard coding scheme known as the American Standard Code for Information Interchange, or ASCII.

<u>Character</u>	<u>Bit Pattern</u>	<u>Character</u>	<u>Bit Pattern</u>
space	00100000	?	00111111
!	00100001	@	01000000
“	00100010	A	01000001
#	00100011	B	01000010
\$	00100100	C	01000011
%	00100101	D	01000100
&	00100110	E	01000101
‘	00100111	F	01000110
(00101000	G	01000111
)	00101001	H	01001000
*	00101010	I	01001001
+	00101011	J	01001010
,	00101100	K	01001011
-	00101101	L	01001100
.	00101110	M	01001101
/	00101111	N	01001110
0	00110000	O	01001111
1	00110001	P	01010000
2	00110010	Q	01010001
3	00110011	R	01010010
4	00110100	S	01010011
5	00110101	T	01010100
6	00110110	U	01010101
7	00110111	V	01010110
8	00111000	W	01010111
9	00111001	X	01011000
:	00111010	Y	01011001
;	00111011	Z	01011010
<	00111100	[01011011
=	00111101	\	01011100
>	00111110]	01011101

Continued

Fig. 1.7. American Standard Code for Information Interchange, or ASCII.

<u>Character</u>	<u>Bit Pattern</u>	<u>Character</u>	<u>Bit Pattern</u>
^	01011110	o	01101111
—	01011111	p	01110000
‘	01100000	q	01110001
a	01100001	r	01110010
b	01100010	s	01110011
c	01100011	t	01110100
d	01100100	u	01110101
e	01100101	v	01110110
f	01100110	w	01110111
g	01100111	x	01111000
h	01101000	y	01111001
i	01101001	z	01111010
j	01101010	{	01111011
k	01101011		01111100
l	01101100	}	01111101
m	01101101	~	01111110
n	01101110	(DEL)	01111111

Fig. 1.7., cont'd. American Standard Code for Information Interchange, or ASCII.

The coding scheme symbolizes sets of electrified bits. This principle is also applied in Morse code, where a series of dots and dashes represent letters.

In ASCII 8 bits make a byte, the unit of memory for a single letter, number, or special character. A group of bytes makes a word such as “house,” which requires 5 bytes or 40 bits (8 bits \times 5 letters).

To visualize the concept, picture a box with 8 electrical wires attached. Each wire will either be on (1) or off (0). Inside the box is a character. The character is determined by the combination of on and off wires. For example, the combination 01000001 represents the letter A.

It should be noted that some computers, like the Macintosh, incorporate several nonstandard ASCII characters. These characters also are represented by binary numbers. To check on the complete Macintosh character set, refer to Appendix E of your Macintosh Turbo Pascal manual.

Mac Memory

Your computer contains thousands of bits of memory. The standard configuration of memory is divided into two types—random-access memory (RAM) and read-only memory (ROM). The unit of measurement for memory is a *kilobyte*, or *K*. One K of memory is equal to 1024 or 2^{10} bytes. Therefore 64K

is a little more than 64,000 bytes, and since one character occupies one byte, 64K can hold approximately 64,000 characters, or 35 to 40 double-spaced pages.

What Are RAM and ROM?

RAM is volatile memory; it can be changed. RAM holds the code for your programs and does most of your calculations, such as simple scratchpad work. This feature works much the same way that a pocket calculator does. It's important to remember that when you turn your computer off, the data in RAM will be lost.

The first version of the Macintosh contained 128K of RAM. Since that initial release, memory upgrades and new Macintoshes (Fat Mac, Macintosh Plus, Macintosh SE, and Macintosh II) allow anywhere from 512K to several megabytes. A megabyte is 1024K or approximately one million bytes. There will be further memory upgrades available in the future, since the 68000 can work with as much as 16 meg of memory.

Information in RAM can be stored externally on a microfloppy diskette or a hard disk before you turn the computer off. It is sometimes confusing when the memory of your computer is discussed in the same context with the data stored on a microfloppy diskette. Just remember that the memory of your Macintosh computer is *internal storage*, while the memory capacity of a microfloppy diskette is *external storage*.

The concept of external storage is used in the music industry. The cassette tape stores musical data on a medium external to the cassette recorder itself. The process used to store your data, external to the computer, will be explained in detail in Chapter 2.

Your Macintosh also contains 64K, 128K, or 256K of ROM (read-only memory), internal memory that cannot be changed. We say 64K, 128K, or 256K because the early Macs were made with 64K, but from the Macintosh Plus on, Apple has been releasing new products with 128K and 256K of ROM. If you have one of the older Macs with 64K of ROM, you can have your computer upgraded to 128K ROM.

ROM can be read from but not written to or used for storage. ROM is used for programs and code segments that must never change and that are not lost when your computer is turned off. Two of the more important portions of the ROM in your Macintosh are the operating system and the QuickDraw routines used to do fancy graphics. It's not important to know how these areas function at this point. Simply recognize that the programs contained in ROM remain there whether the computer is on or off.

Getting Ready to Use Turbo Pascal

Your Macintosh Turbo Pascal package comes with two disks: the Program Disk and the Utilities & Sample Programs Disk. Before you do anything else with these disks, you should copy them, put the originals in a safe place, and use the backup copies only. The contents of each of these disks is shown below:

PROGRAM DISK FILES

Turbo X.X	(X.X is the version of Turbo Pascal you have)
Read Me	Program that provides additional Turbo information
Read.file	Text file used by Read Me above
UnPack	Program used to unpack the Mac II Interfaces file below
Mac II Interfaces	Interfaces necessary to program for Mac II

NOTES

UTILITIES AND SAMPLE PROGRAMS FILES

	<u>NOTES</u>
RMaker	Program to create resource (RSRC) files
Font/DA Mover	Utility to move fonts and desk accessories
Unit Mover	Utility used to move Turbo Pascal's units to and from files
MyDemo Folder	Demo example
MyDemo.pas	Pascal source code
MyDemo.R	Resource text file
MyDemo.Rsrc	RMaker-compiled version of MyDemo.R
MyDA Folder	Example of a desk accessory
MyDA.pas	Pascal source code
MyDA.R	Resource text file
MyDA.Rsrc	RMaker-compiled version of MyDA.R
Turtle Folder	Example of Turtle Graphics
Dragon.pas	Pascal source code for graphics
C_Curve.pas	Pascal source code for graphics
TurtleTest.pas	Pascal source code for graphics
TurtleUnit.pas	Movable unit for Turtle Graphics
Clock Folder	
Clock.pas	Pascal source code
Lister Folder	
Lister.pas	Pascal source code
Lister.R	Resource text file
Lister.Rsrc	RMaker-compiled version of Lister.R

UTILITIES AND SAMPLE PROGRAMS FILESNOTES**Macintalk Folder**

Macintalk

Macintalk utility

Speak.pas

Pascal source code for speech example

Speak.R

Resource text file

Speak.Rsrc

RMaker-compiled version of Speak.R

Sample Speech

Auxiliary file for Speak program

Sound

Noise.pas

Pascal source code for sound program

Noise.R

Resource text file

Noise.Rsrc

RMaker-compiled version of Noise.R

Other Demos

Numerous.pas, .R and .Rsrc files

Misc Folder

ATalk/ABPackage

68000-based debugger package

MacsBug

Compat Unit

Compat.inc

Include file for compatibility

Compat.doc

In order to use Macintosh Turbo Pascal, you need a Macintosh with at least 512K of RAM and one single-sided 400K floppy disk drive. If you are working with a more powerful machine (Mac Plus, SE, II) or have a double-sided drive, external drive or hard disk drive, you should have plenty of flexibility in working Turbo Pascal. Since Borland does not copy-protect Turbo Pascal, you may copy any of the files to another floppy disk or to a hard disk. If, however, you are using the minimum configuration, you will probably run into memory problems when trying to work with larger programs, especially those that call in additional units. You should note that neither Turbo Pascal disk is bootable; that is, neither of them has the Finder or System files. Because of this you must either boot off another disk and then work with the Turbo disk or set up the Turbo disk with only the essential files. For example, you could remove the Read Me, Read.File, UnPack and Mac II Interfaces files and be able to put the Finder and System files on the program disk. Because the Turbo Pascal program is so large (over 200K), you may find it necessary to conserve memory by removing some of the units you won't be needing. To see how this is done, refer to the UnitMover section in Chapter 7. If you are using a 512K Mac with one single-sided drive, we suggest that you acquire a copy of a RAM disk package to use with Turbo. A RAM disk package allows you to set up your system so that the Macintosh thinks you have two disk drives. The RAM disk program reserves a portion

of RAM and tells the Mac to refer to this area of RAM as another disk. This is suggested because if you put the full Turbo program on a 400K disk with the Finder and System files, there's not much room left to store your Pascal source files. When you set up your RAM disk, do it so that the Finder, System and Imagewriter files are the only ones on the RAM disk. You can then use a separate disk to hold just the Turbo program and your necessary source files. Because this "disk" resides in RAM, when you turn your Macintosh off, its contents disappear. But with the configuration described above, your source files will always be written to the diskette with the Turbo program, so there is no need to save the contents of the RAM disk before you turn the Mac off.

Starting Turbo Pascal

The following is a quick start for getting ready to use the Macintosh with Turbo Pascal. The only assumption is that your Macintosh is completely set up (all peripherals plugged in and the computer's power cord plugged into the appropriate AC outlet).

Step 1: Turn on your Macintosh and adjust the brightness of the display screen. The power switch is labeled 1 for on and 0 for off.

Step 2: Insert your backup copy of the Turbo Pascal Program Disk into the disk drive with the label facing up and the metal slide switch away from you. The disk can be inserted only one way; if it doesn't fit, don't force it. You can, of course, insert the Program Disk before you turn on the computer. The desktop should display the Turbo Pascal disk icon as in Figure 1.8.

Step 3: Open the Turbo Pascal disk icon by double-clicking the mouse unit (press the mouse button rapidly twice and release) or by selecting Open from the File menu (see Figure 1.9).

Once opened, the Turbo Pascal Program Disk will display icons for each of the files described above (see Figure 1.10).

Once the Pascal disk is booted, or loaded into your computer, a built-in applications program called Finder takes control of the operations that control the system. Similar to a central nervous system (or operating system), Finder allows you to move from one application to another, create disk windows, start an application from an icon, and so on. For now we will let Finder help us select the icon labeled Turbo X.X (again, X.X refers to the version number) and activate its contents.

Position the pointer directly over the Turbo X.X icon with the checkered flag and then perform a double-click with the mouse (or select Open from the File menu). Within a few seconds the Turbo Pascal screen will appear (see Figure 1.11).

At this point you are ready to begin using Turbo Pascal and realize the full power of your Macintosh computer. Let's continue by turning to Chapter 2.

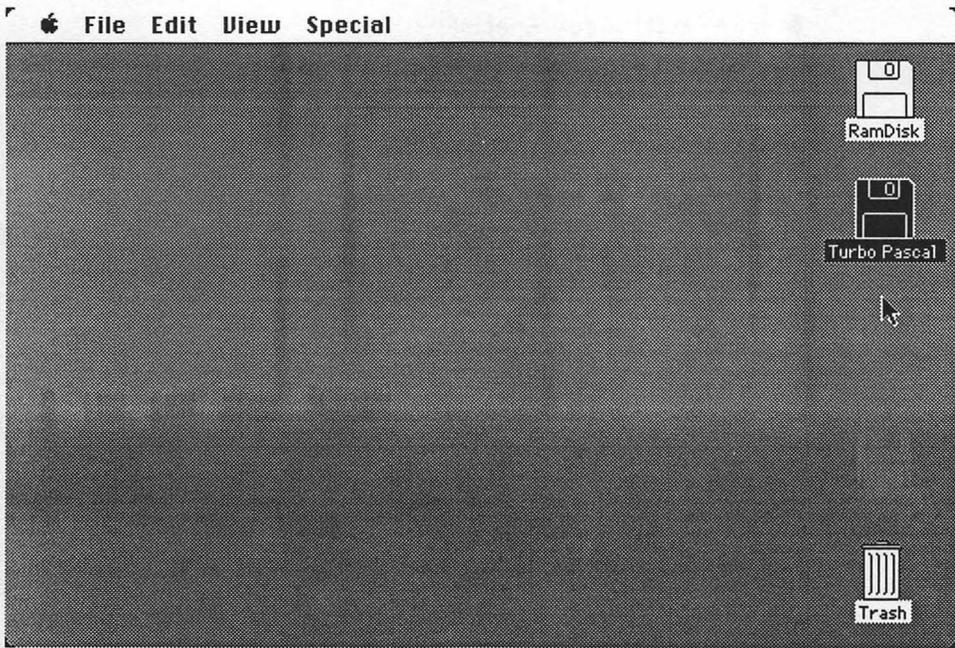


Fig. 1.8.

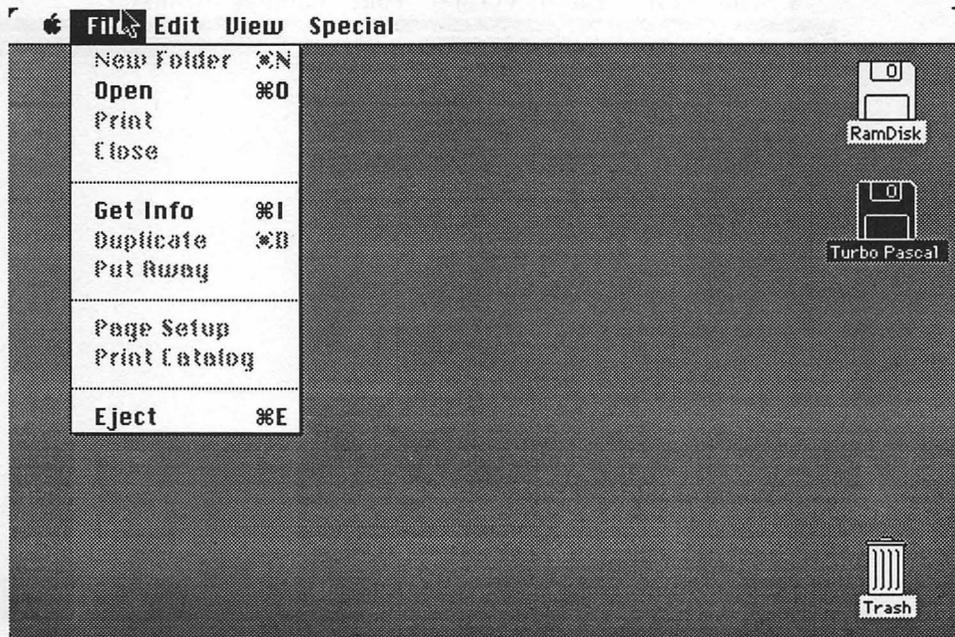


Fig. 1.9. Open the Turbo Pascal Icon.

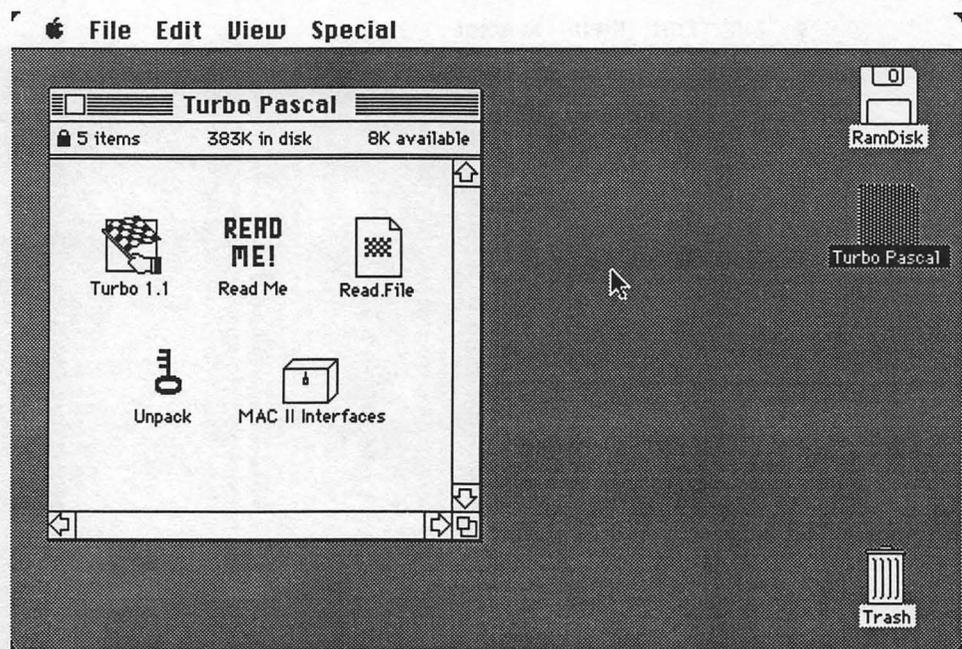


Fig. 1.10. The Turbo Pascal Desktop.

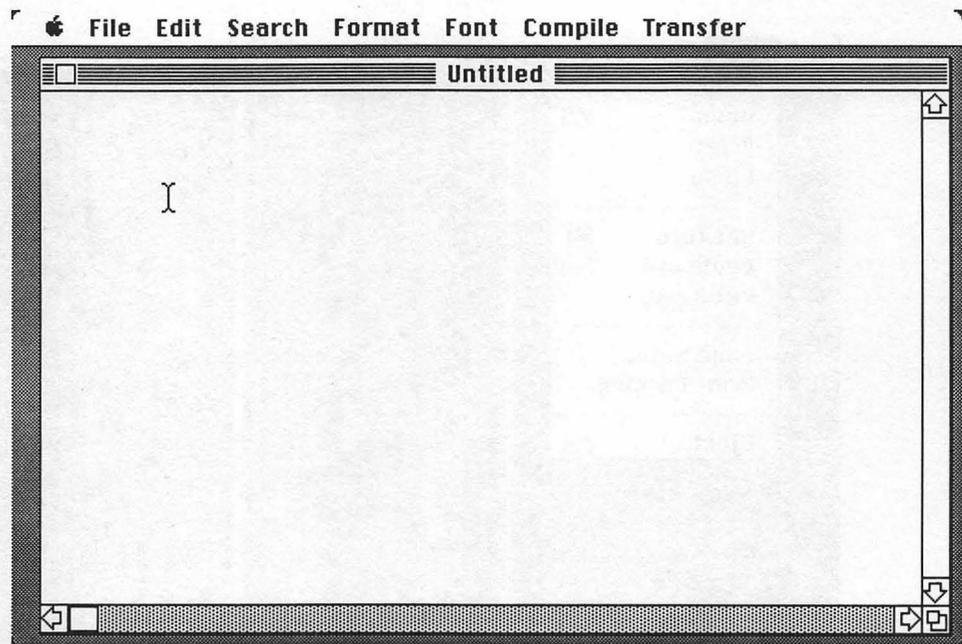


Fig. 1.11. The Turbo Pascal Screen.

Review Summary

1. A set of sequential instructions is called a program. Programming is creating a set of specific instructions to perform a specific task.
2. Hardware is the nuts and bolts of your computer, the part you can physically touch. Software is the program instruction.
3. The Macintosh doesn't understand Pascal commands; they must first be translated either by an interpreter or by a compiler such as Turbo Pascal.
4. Programming logic can be depicted by using flowcharts. Flowcharts provide a road map in structured form to a programming problem.
5. The Turbo Pascal compiler translates your Pascal statements into machine-language instructions. Machine language is represented in binary form. Binary instruction is written in sets of 0s and 1s.
6. The binary digit, or bit, is the fundamental unit used by your computer. A byte represents the memory storage for a single number, letter, or special character.
7. The internal storage of your Macintosh consists of two kinds of memory, RAM and ROM. Storage is often referred to as the external storage of your computer system. The external storage medium is the microfloppy diskette or hard disk.
8. Memory refers to the information that your Macintosh can store and retrieve. The unit of measure is the kilobyte, or K.
9. RAM is volatile and can be changed. ROM is that portion of memory that cannot be changed. RAM can be used to do scratchpad work or run various application programs. When the computer is turned off, the data in RAM is lost. ROM can be read from but not written to or used for storage. When the computer is turned off, the programs in ROM remain intact.

Quiz

1. How much memory does one K equal? 128K equal? 512K equal? one meg equal? four meg equal? What acronym does ASCII represent?
2. What is the primary difference between a compiler and an interpreter?
3. What happens in the problem-solving phase of programming? In the implementation phase? What is the solution to a programming problem called?
4. Design the necessary steps to boil an egg using a simple flowchart.

Getting Acquainted with Turbo Pascal

The Turbo Pascal Screen
Menu Discussions
Your First Turbo Pascal Program
Running Your First Program
Editing Your Program
Saving Your Program
Printing Your Program
Review Summary
Quiz

In this chapter you will learn:

- How to work within the Turbo Pascal environment.
- All of the options available to you in the Turbo menus.
- How to enter your first Pascal program.
- How to run or execute a program.
- How to edit, save, and print a program.

In this chapter we introduce you to the Turbo Pascal screen and acquaint you with the special features of Turbo that make it a joy to work with on the Macintosh. In addition you will learn how to enter, edit, run, save, and print your first Turbo Pascal program. We often refer to Macintosh Turbo Pascal as Turbo Pascal or just Turbo or Pascal. The official name is Macintosh Turbo Pascal, but any of these names is appropriate.

It's important to note that many of the operations discussed in this chapter are simple extensions of the ones learned while using MacWrite or MacPaint. That's the beauty of Macintosh software. All software is designed to take advantage of the Mac interface and thus provide a common ground for learning. For example, the operation to print a hard copy of a Turbo Pascal program is similar to printing a letter using MacWrite. Just in case you haven't mastered these operations, we will review them as our discussion continues.

The Turbo Pascal Screen

The Turbo Pascal screen provides the user with a window in which he or she can immediately start entering Pascal code (see Figure 2.1). Along with this active code entry window, the regular Mac-style menu bar is displayed with the following menus: Apple, File, Edit, Search, Format, Font, Compile, and Transfer. Before going any further, take a close look at each of these menus so that you will know exactly what they offer you.

Menu Discussions

The Apple Menu

The leftmost menu is the standard Apple menu, which displays options for running any installed Desk Accessories as well as an option for learning more about Turbo... (see Figure 2.2).

If this option is selected, a small box is displayed. It explains that Turbo Pascal is a Borland International product and shows the version number (see Figure 2.3).

Any other options displayed with this menu are dependent upon how you have your Desk Accessories configured.

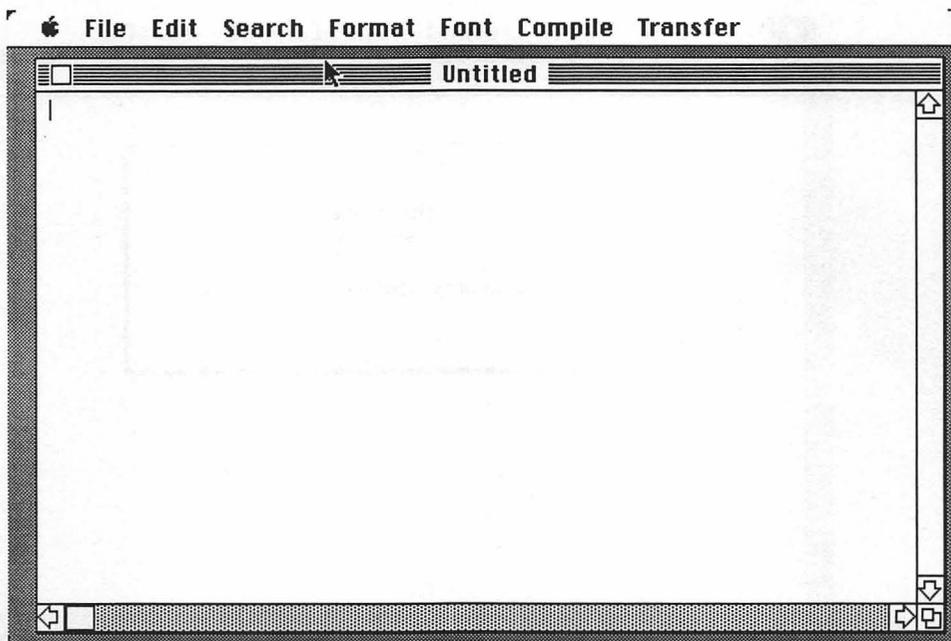


Fig. 2.1.

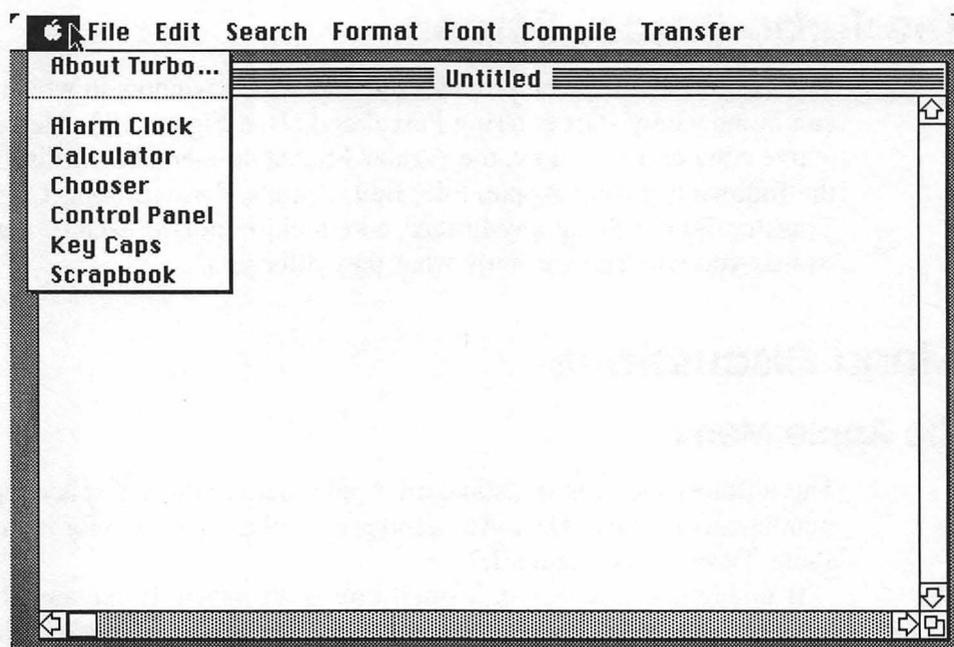


Fig. 2.2.

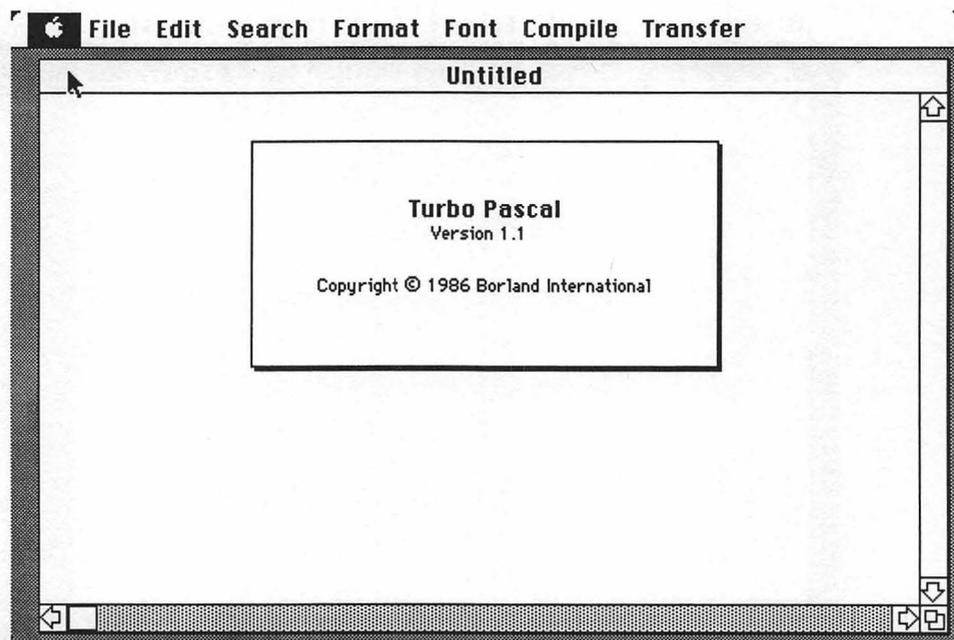


Fig. 2.3.

The File Menu

The File menu permits you to manipulate files, print files, and perform other miscellaneous tasks (see Figure 2.4).

When the first option, New, is selected, a new editing window will open and become the active window. You can have up to eight windows open at any one time in Turbo, so you can edit multiple files without having to open and close them every time you want to switch.

The next option, Open, allows you to open a file that already exists on a disk. When you select this option, a dialog box like the one displayed in Figure 2.5 is displayed. This dialog box allows you to scroll among the available files on the active disk as well as change drives or eject a disk so that a file may be opened.

Open Selection appears next in the File menu, and it allows the user to open the file named by the selected text in the active window. This option can be quite handy when you start dealing with what are known as *include files*. Include files contain information about a program that may be needed for multiple programs; breaking the information out into its own file lets multiple programs refer to it without having to copy its contents into their own source files. If you don't understand this, that's OK. This option will be used only by fairly experienced programmers. The concept of include files is

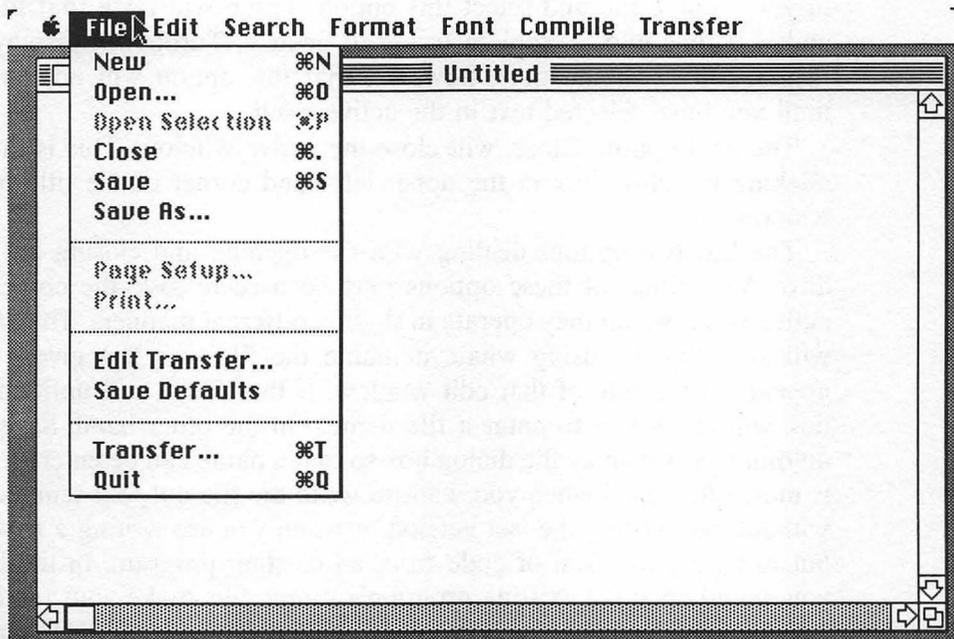


Fig. 2.4.

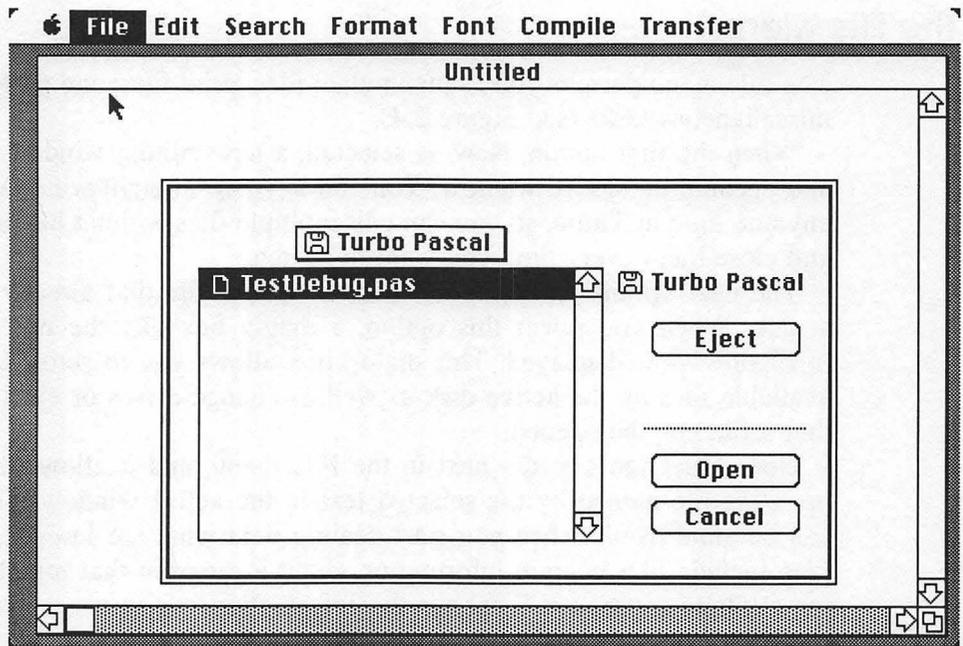


Fig. 2.5.

further explained in Chapter 6. For those who are familiar with include files, all you must do is select the include file by dragging the mouse over its name in your source file and select this option. Turbo will open that file for you, and if it runs into a problem trying to open it, Turbo will display an error message. You should note, however, that this option will not be available until you have selected text in the active window.

The next option, Close, will close the active window. This is the same as clicking the close box in the upper left-hand corner of the title bar of that window.

The last two options dealing with file opening and closing are Save and Save As... Either of these options may be used to save the contents of the active window, but they operate in slightly different manners. The Save option will save the file using whatever name the file was last given; the name appears in the title of that edit window. If that file is still untitled, a dialog box will allow you to enter a file name. On the other hand, Save As... will automatically display the dialog box so that a name can be entered. Save As... is most often used when you want to write the file out to a temporary name without overwriting the last version or when you are writing a new program but using a good deal of code from an existing program. In the latter case you would open the existing program's source file, make your modifications, and then use Save As... to write the modified file with the new name.

The next two options deal with printing files: Page Setup... and Print... When the Page Setup... option is selected, the dialog box shown in Figure 2.6 is displayed. This dialog box allows you to select several options like paper style, horizontal/vertical printing, and a few special effects.

When the Print... option is selected, the dialog box shown in Figure 2.7 is displayed. This dialog box permits you to select the output quality, what pages to print, and how many copies as well as how the paper will be fed into the printer.

Once you have made your selections in this box and clicked the OK button, the printing process will begin. At this point you will have the option of terminating the print command by clicking the cancel button when the printing begins.

Edit Transfer... appears next and allows you to edit the file used to transfer from one application to another. This file is used by the Transfer menu, which is the rightmost menu on the Turbo menu bar. When you select this option, a dialog box is displayed (see Figure 2.8) in which you can enter the names of applications you would like to have in the Transfer menu. There are no applications initially installed in Transfer and as a result no options are initially available in the Transfer menu. To install an application enter its

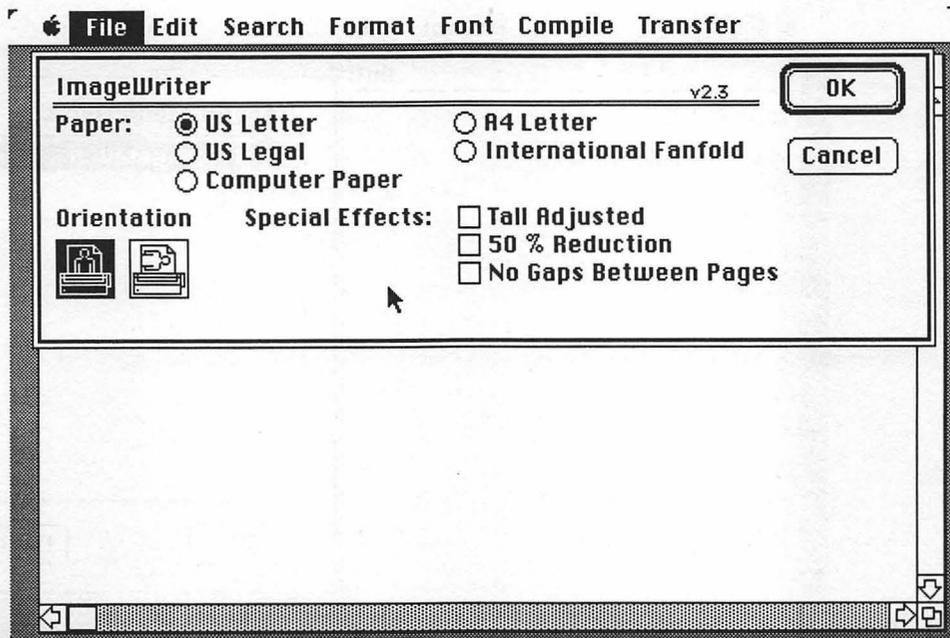


Fig. 2.6.

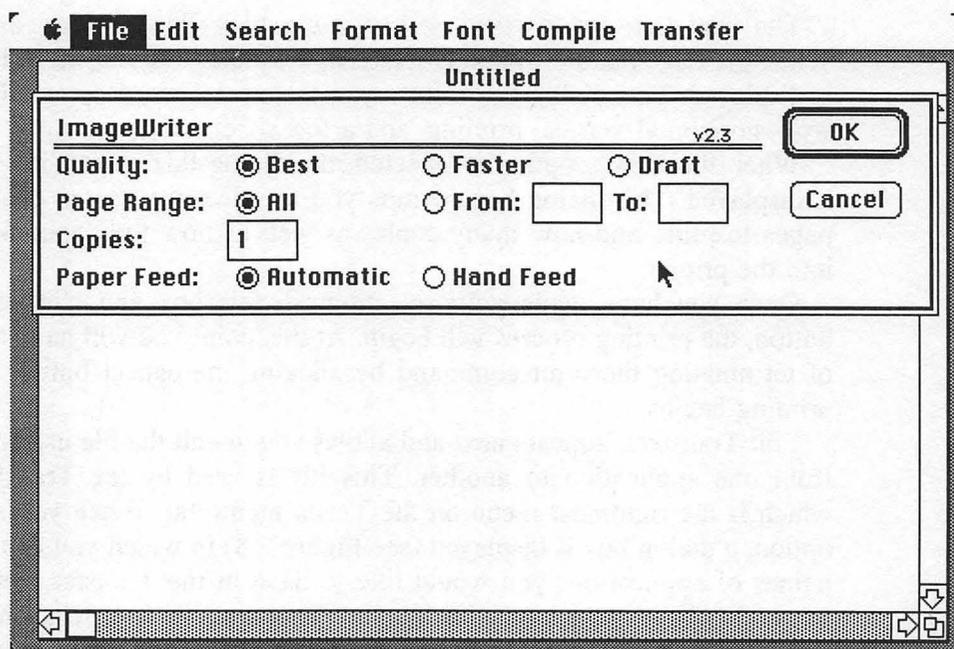


Fig. 2.7.

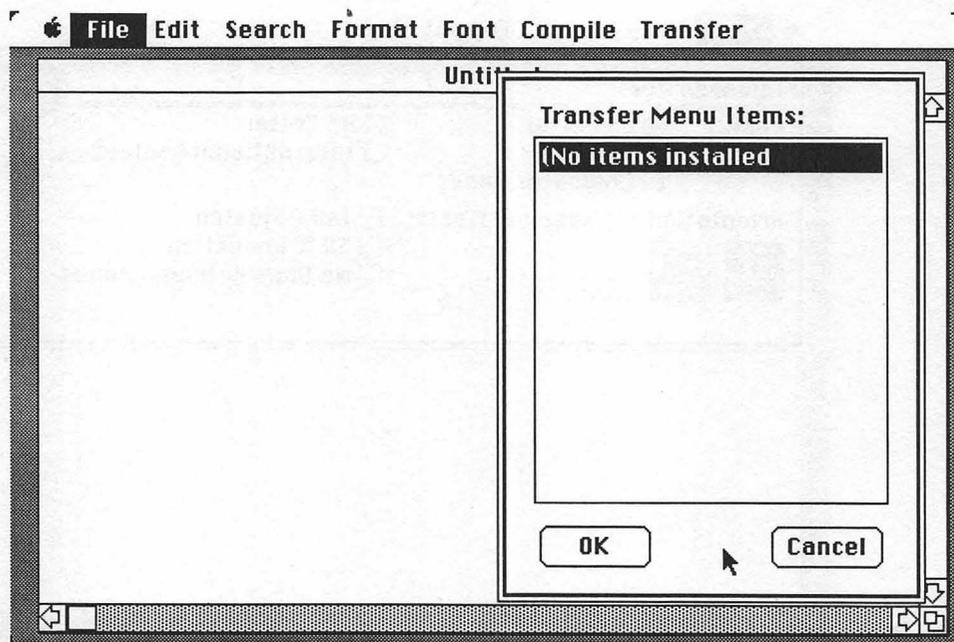


Fig. 2.8.

name in the Transfer menu Items box and click the OK button. If you look at the Transfer menu, you should see an option available for that application. You can select that option and bypass the Finder desktop. The applications you install with Edit Transfer... will remain intact until you quit the Turbo program. To save them for future Turbo sessions, select Save Defaults in the File menu. This menu option also saves any compiler options set up in the Compile menu and edit options set up in the Edit menu (both are discussed below).

The Transfer menu should not be confused with the next item in the File menu: the Transfer option. When this item is selected, the familiar file open dialog box is displayed (see Figure 2.9).

The items displayed in this box are all the double-clickable applications (or files containing applications) found on the active disk. You can, of course, select to eject the active disk or change disk drives from this dialog box; however, the only items displayed on subsequent disk selections will be double-clickable applications found on that disk.

The last option in the File menu is Quit. As you may have guessed, this option allows you to leave the Turbo program. Before you leave, you are asked whether you would like to update any open files.

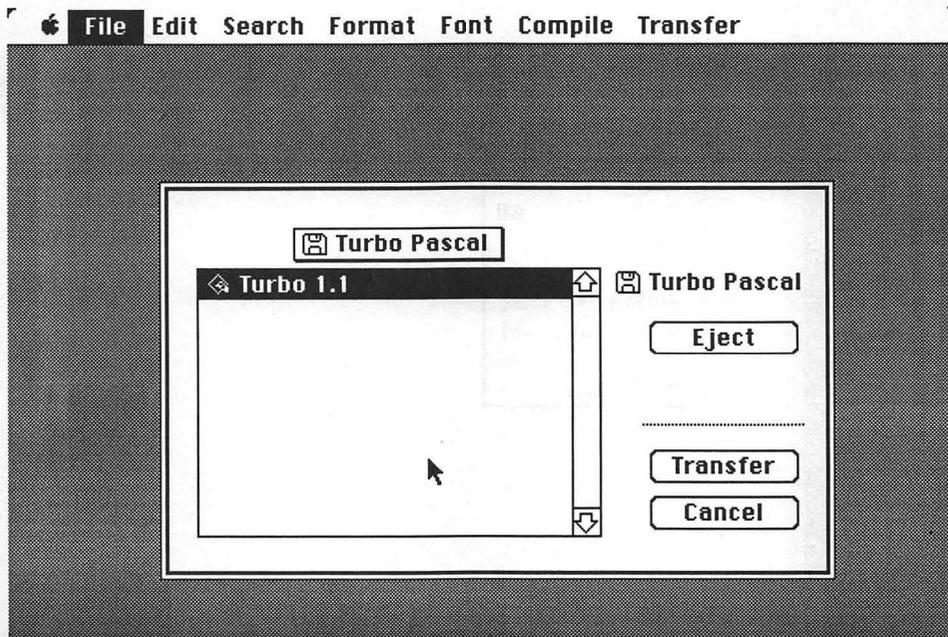


Fig. 2.9.

The Edit Menu

The Edit menu should be familiar to anyone who has worked with any other Macintosh applications. In the Turbo environment it offers the same basic functions you would find in another editor or word processor (see Figure 2.10).

The first option, Undo, does exactly what it says; it reverses the most recent editing operation performed. For instance, if you have just deleted a couple of blocks of code and now decide you shouldn't have, select Undo and the code is magically back in place. This works only with the last operation performed, however. If you deleted those blocks, then deleted a few more lines somewhere else, the original blocks you deleted cannot be brought back via Undo.

Cut may be used to delete the currently selected text. This text is automatically copied to the Clipboard, a temporary storage area, so that it may be restored or copied elsewhere via Undo or Paste. On the other hand, the Copy option will copy the selected text to the Clipboard so that it may be copied elsewhere. Paste will copy the text in the Clipboard to the location of the cursor.

The Clear option deletes the selected text and does not copy it to the Clipboard. This is identical in function to selecting text and pressing the backspace key.

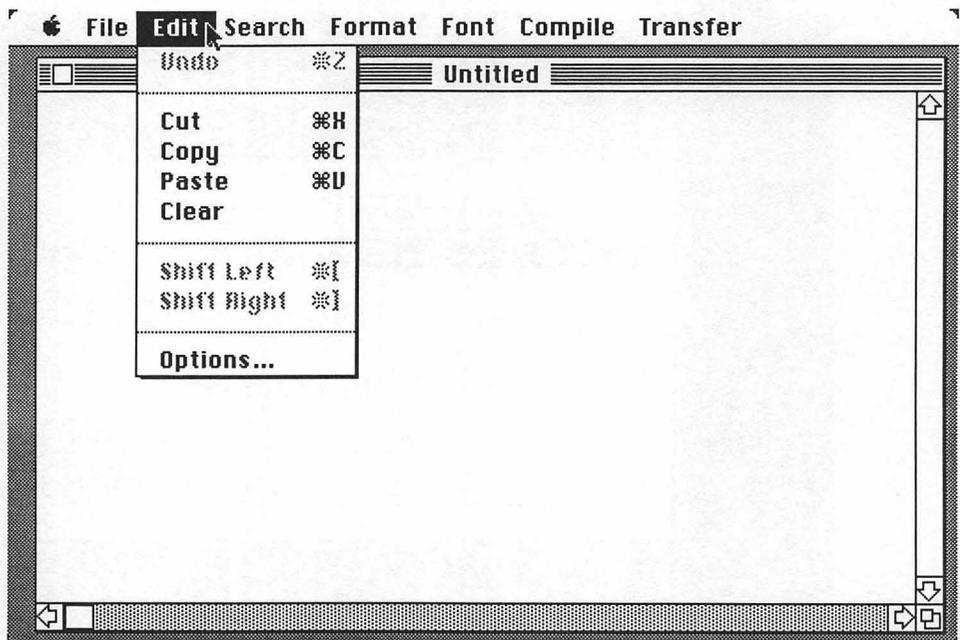


Fig. 2.10.

The next two options, Shift Left and Shift Right, are very useful for indenting and aligning blocks of code. To use either one, select an entire line of text (or several lines) and select the option to shift the text either left or right one space. If you have trouble selecting either of these options because they are disabled (light gray), make sure you have selected entire lines. The easiest way to ensure this is to move the mouse all the way to the left of the edit window and slide it down until you have the entire block selected.

The last option in the Edit menu is entitled Options... When this item is selected, the dialog box shown in Figure 2.11 is displayed.

As you can see, this box allows you to enter options for the tab width (how many spaces are skipped with the tab key), and whether Auto Indent and the startup window are active. Auto Indent is a very handy option. It starts each new line at the same horizontal position of the line above. This enables the programmer to keep blocks of code aligned without having to count the number of spaces to indent with each new line. When the startup window is selected, each time Turbo is started up, an untitled window will be displayed for immediate editing. This option should be based upon your own personal preference, but we recommend that you keep the Auto Indent option enabled. Once you have made any option changes, they will remain in effect until you leave Turbo. To save them for later use, select the Save Defaults command from the File menu.

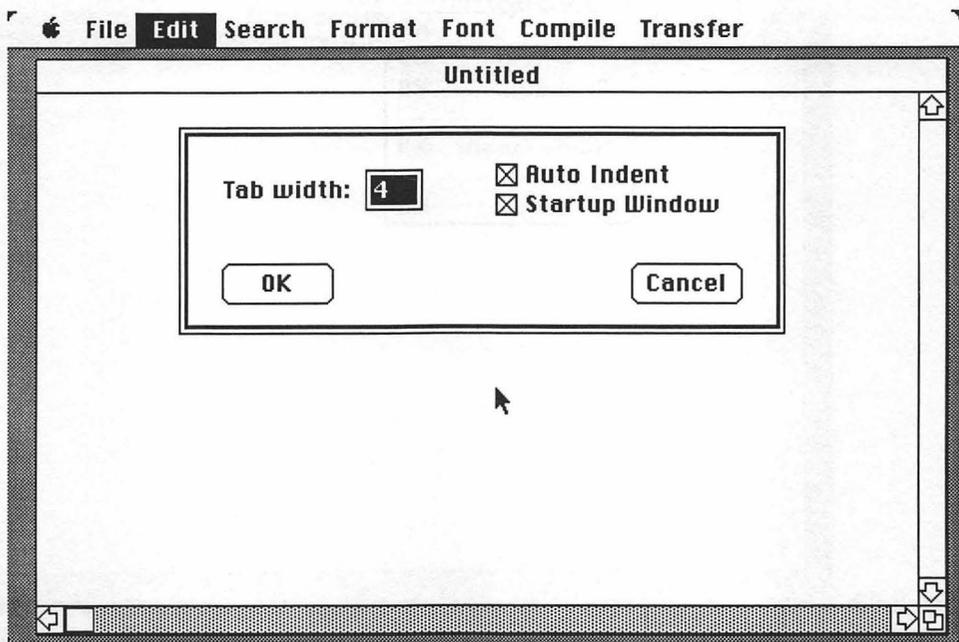


Fig. 2.11.

The Search Menu

The Search menu provides you with the basic functions of search, search and replace and a few other handy features (see Figure 2.12). When the Find... option is selected, the dialog box shown in Figure 2.13 is displayed.

To find a particular string—a sequence of characters, like “computer”—enter the string in the Find What: field and select whether your selection is for complete *words only* or if it is *case-sensitive*. For example, if you want to find the next occurrence of the word “for,” enter that string in the Find What: field and select Words Only. If you click the OK button, Turbo will search through your file until it finds the string “for.” By selecting Words Only, you are preventing Turbo from stopping if it finds the string “form” or “forest” before it sees a “for.” If you leave that option off, any word with the substring “for” is a candidate for the search. In addition, in words-only mode the strings “FOR,” “FoR,” “For,” and so forth are treated as the equivalent of “for.” If you want only the exact upper- or lower-case format that you have entered, select Case Sensitive. Note that the search begins at the cursor and *does not* wrap around to the beginning of the file. If you have the cursor positioned at the end of the file, you’ll never find anything with the Find... option. When Turbo has located the desired string, it will move to that portion of the file and the text will be shown in reverse video. If Turbo is unable to

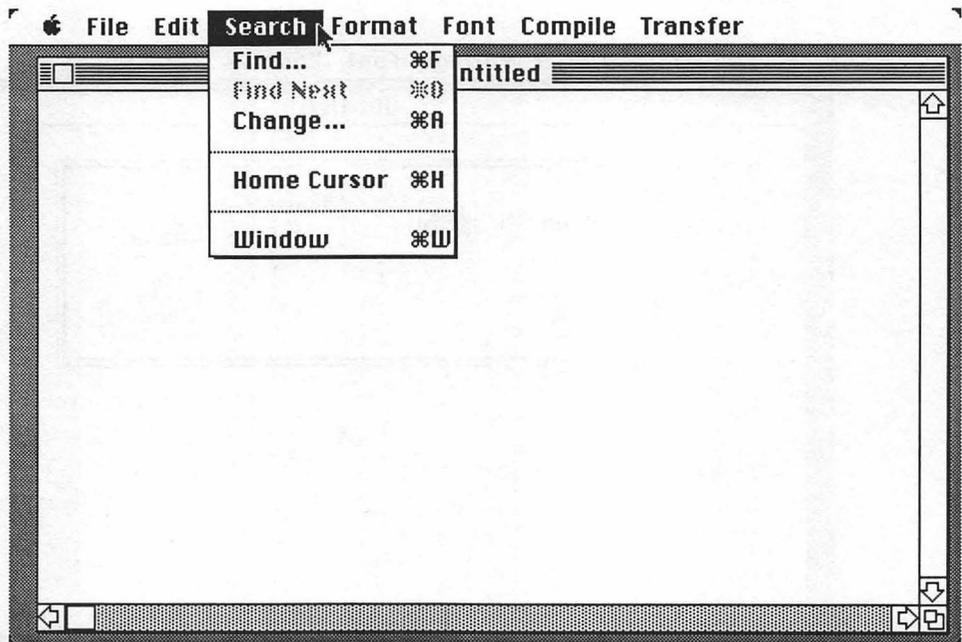


Fig. 2.12.

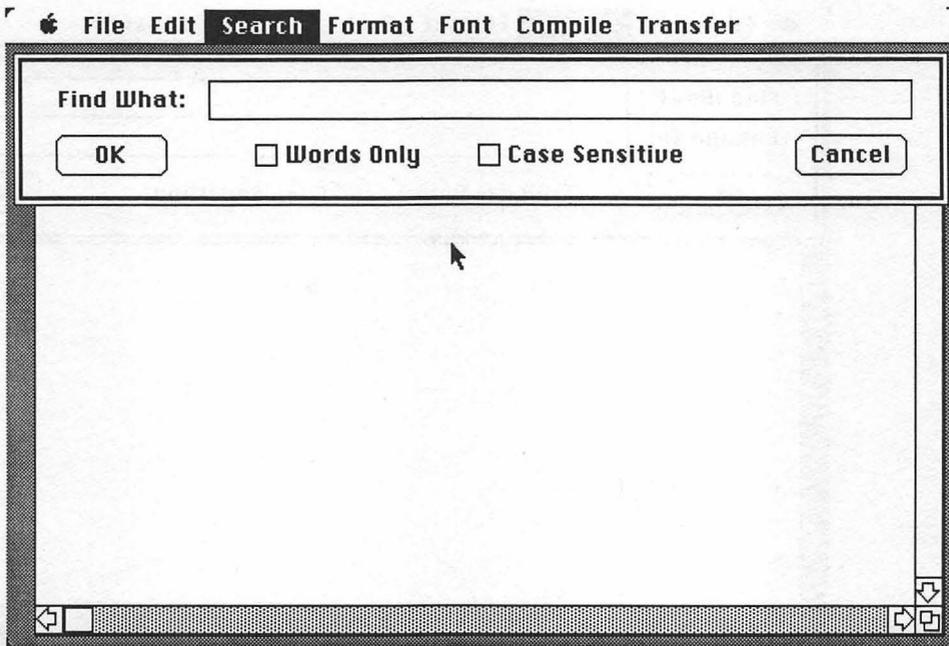


Fig. 2.13.

find that string, the Mac will beep and you'll be back where you were before the Find... option was selected.

Find Next may be used to search again for the next occurrence of the string last specified in either a Find... or Find Next. If you want to find all instances of the word **end** in your program, start by selecting the Find... option and specify the words-only string **end** as the search criterion. Once you find the first occurrence, select Find Next... without reentering the string, and it will take you to the next occurrence.

If you select the Change... option from the Search menu, the dialog box shown in Figure 2.14 will be displayed.

In this box you can specify the string to be searched, the change to be made, and Words Only or Case Sensitive. Once you have entered the necessary information and clicked OK (or pressed "RETURN") in this dialog box, the dialog box shown in Figure 2.15 is shown every time Turbo encounters your search string.

Within this dialog box you can specify whether you want the change to be made to each case of the search string. If you know you want to change every occurrence, click the all button, and Turbo will make the changes without any intervention on your part.

The last two commands in the Search menu, Home Cursor and Window, are little extras that don't actually do any searching but can assist you in

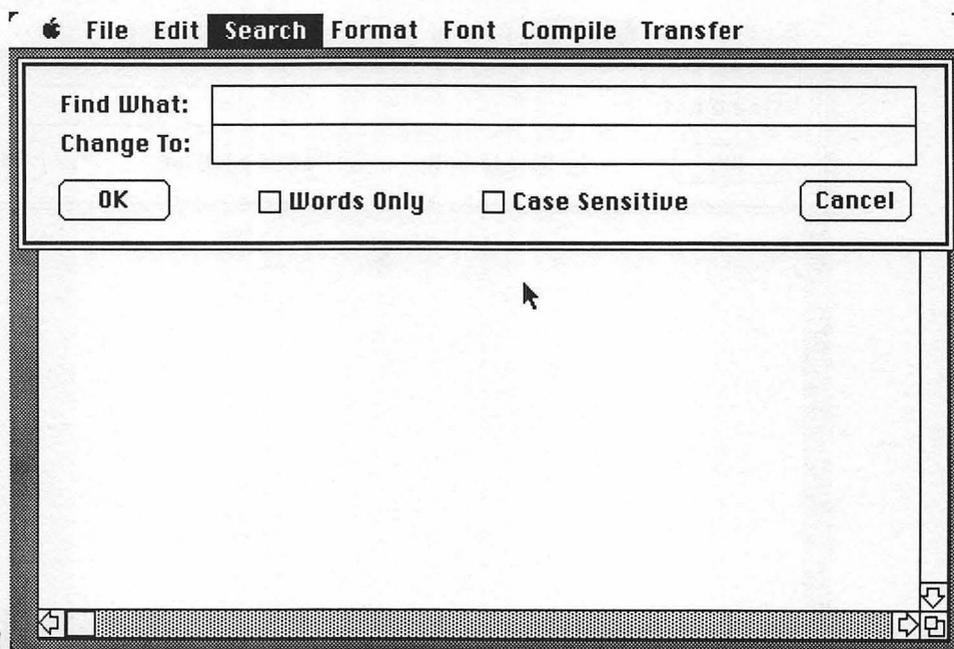


Fig. 2.14.

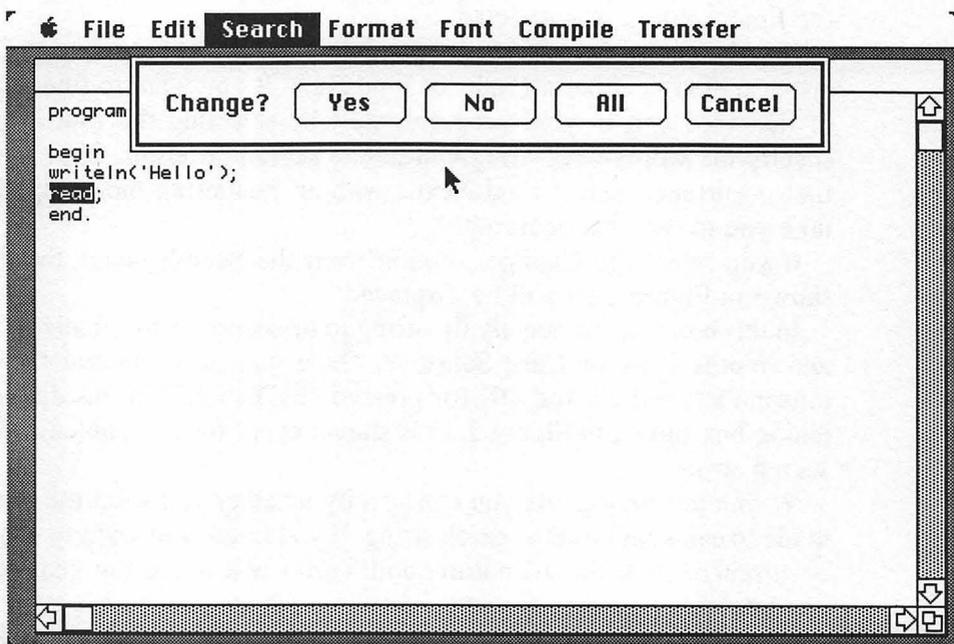


Fig. 2.15.

moving the cursor and switching windows. Home Cursor will move the cursor to the top left corner of the active window, the home position. This can be handy when you are working with a large file and want to jump and move the cursor to the top of the file with one command. The Window command allows you to hop from one window to the next within the Turbo editor. As described later in this chapter, you can have up to eight windows open in the editor, and this command allows you to move back and forth between all the open windows.

The Format Menu

The Format menu gives you the ability to set up your windows in the Turbo editor in a couple of different ways. It also allows you to expand and shrink the active edit window as well as select different type sizes for each window (see Figure 2.16).

The first option in the Format menu, Stack Windows, will take all the open edit windows and arrange them so that the title bars for each are visible and

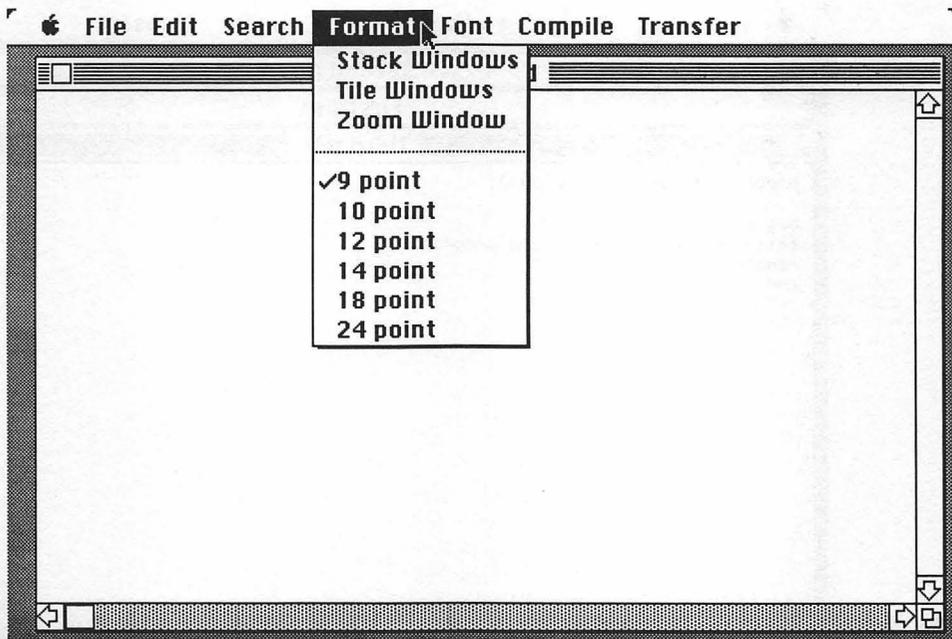


Fig. 2.16.

one window is stacked on top of the next (see Figure 2.17). This is Turbo's default method of laying out windows as you open them in the editor.

Conversely, if you select the Tile Windows option in the Format menu, your edit windows will be laid out so that all of the windows are completely visible, although they are shortened vertically (see Figure 2.18).

This format allows you to see at least a portion of the text in each edit window while working in any one. This window layout is particularly attractive with the next option in the Format menu, Zoom Window. The Zoom Window will cause the active window either to expand to occupy the entire screen or shrink back to the size it was originally. This is essentially the same as double-clicking on the active window's title bar. In the Tile Windows environment Zoom Window allows you to take one of the edit windows, blow it up to full size to do some work within it, then select Zoom Window again to reduce it to the size it was before you selected Zoom Window. Used together, Tile Windows and Zoom Window are quite handy when working with multiple files in the Turbo editor.

The remainder of the Format menu consists of several type sizes you may select for use in any of your windows; each window can use a different type size.

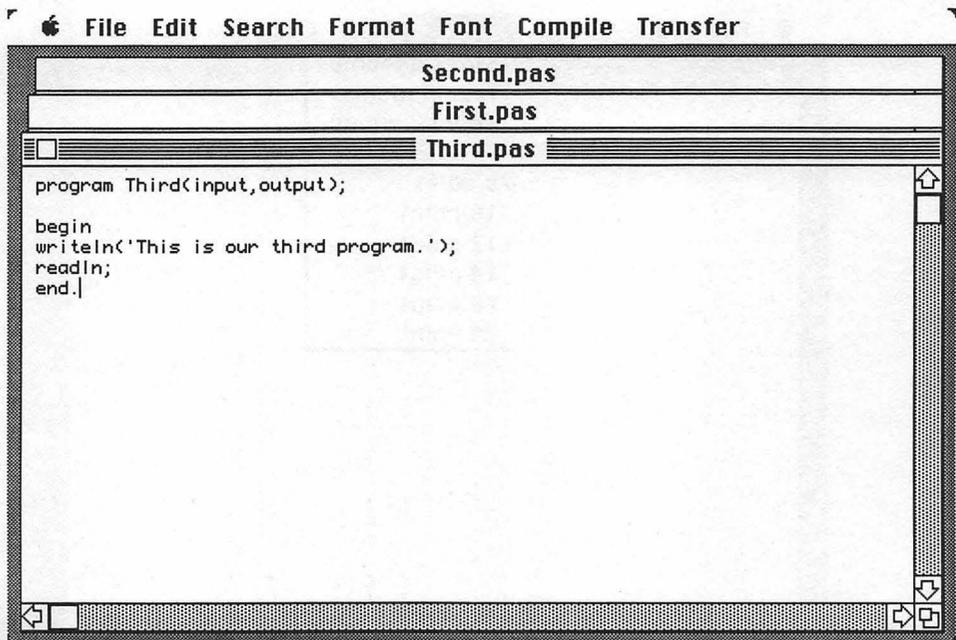


Fig. 2.17.

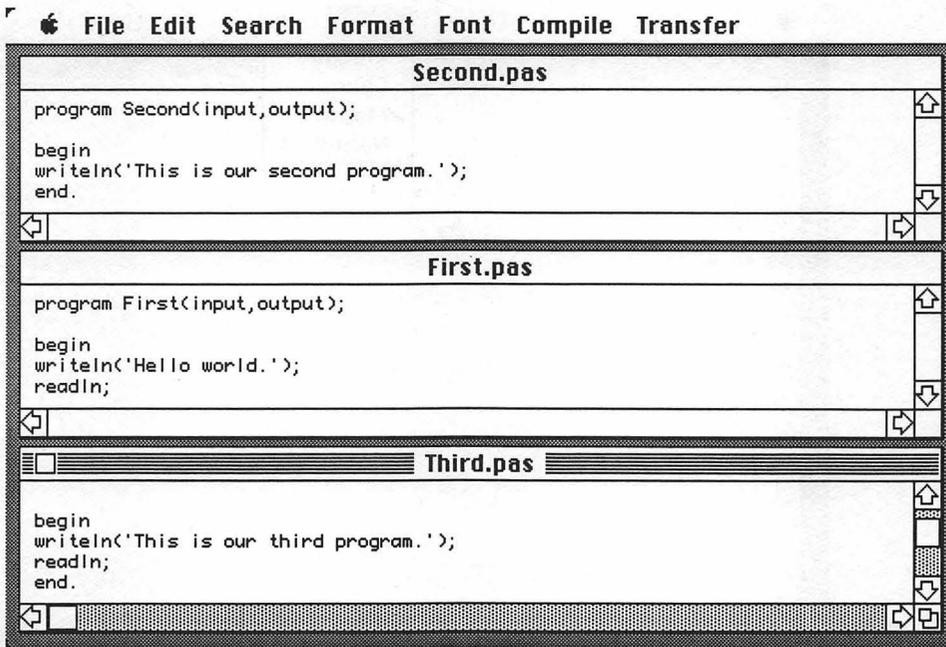


Fig. 2.18.

The Font Menu

The Font menu shows all the available fonts you may use in your edit windows. Just as you may wish to have different type sizes in different windows, you may wish to have different type styles (see Figure 2.19).

The fonts listed in this menu are those held in the system file. You may install or remove fonts from this file using the Font/DA Mover application.

The Compile Menu

The Compile menu is probably the menu you will use most often in Turbo Pascal. It provides you with the ability to compile, execute, check syntax, find errors, and a few other goodies (see Figure 2.20).

The first option, Run, will compile the program in the active window to memory (discussed below) and execute it immediately provided no errors are encountered. If your program has already been compiled and no changes have been made to it, Run will execute your program without recompiling it. Run is probably the most often used command on the Compile menu, since it automatically recompiles any time you change your program.

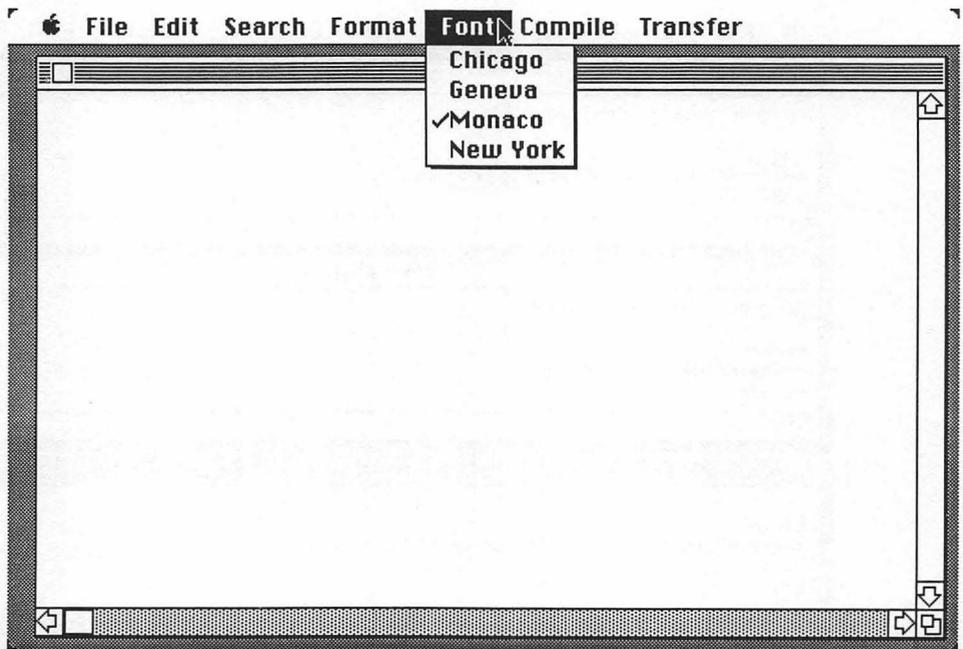


Fig. 2.19.

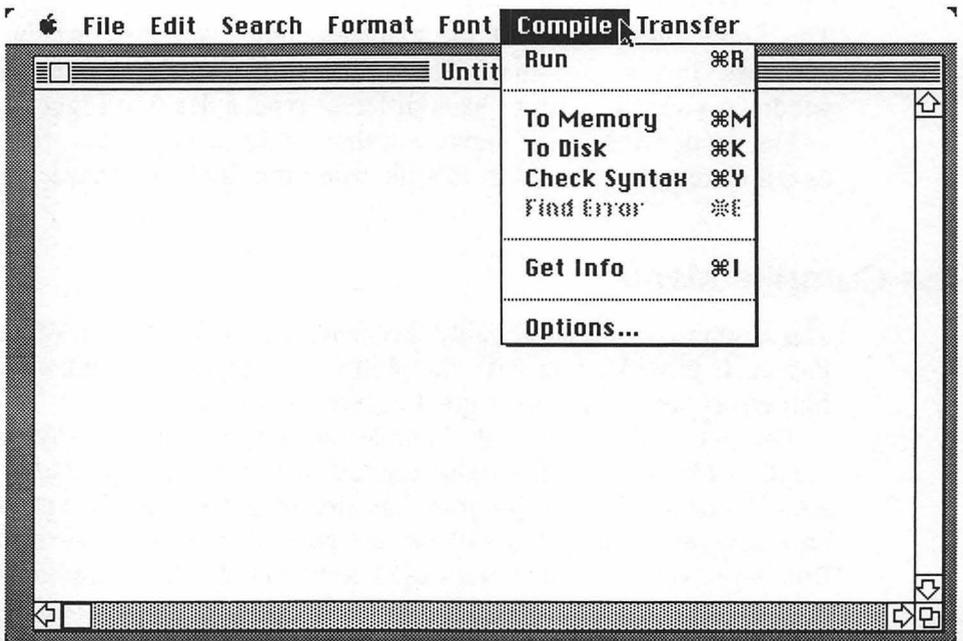


Fig. 2.20.

The next two options, To Memory and To Disk, compile your program to memory and disk respectively. Compiling to memory is processing the Pascal code in the active window and placing the resulting executable module (68000 machine code, which the Macintosh understands) in RAM. As you know, RAM is volatile, so if you select this option and your Macintosh loses electrical power or you quit the Turbo program, the executable version of your program will not be saved anywhere. Of course, all you have to do to run your program again is get back into Turbo and recompile it, but if you want to have a double-clickable application on disk, you must choose To Disk in the Compile menu. When To Disk is selected, Turbo compiles the program in the active window and writes the resulting executable module to the active disk. This results in an additional icon that appears on that disk's contents on the desktop. This icon, if you double-click on it, will execute just as does MacWrite or any other Macintosh application. We'll discuss these concepts in further detail later in this chapter, but you should now understand the difference between compiling to memory and compiling to disk.

The next option in the Compile menu is Check Syntax. It does just what it says. When you select this option, Turbo processes the program in the active window for any syntax errors. If an error is encountered, a message is displayed. Generally the offending code is selected in reverse video. If no error is found, you are returned to the editor within that program's window. When Turbo checks your programs for errors via Check Syntax, Run, or either of the compile options, as soon as it encounters an error, it stops and displays an appropriate error message. However, if you have more than one error in the program, the next error will not be detected until the next time you try to run the code through the compiler. This practice of signaling only one error at a time is fairly common among PC-based compilers such as Turbo. No doubt many who are used to running compilers on larger machines are spoiled by the ability of the more sophisticated compilers to find many errors on a single pass. You may find yourself having to recompile your programs many times before all your errors are uncovered, but the Turbo compiler is so fast and interacts so well with the Turbo editor that this is not a problem.

The next option on the Compile menu, Find Error, will remain disabled until you encounter a run-time error. A run-time error crops up after compilation, while the program is executing. In other words, the program passed all the syntax tests Turbo uses before creating an executable module but contains a problem so severe that execution stops. All the possible run-time error types are listed under System Error Messages in your Turbo Pascal manual. If you are unfortunate enough to encounter one of these errors, control will be passed back to the Turbo editor and the error message will be displayed at the top of the screen. Once you click on the error message, it will disap-

pear and the line of code where Turbo detected the error will be selected in reverse video, just as with a compile error. If you happen to move about in that window and perform some editing and wish to find where the error was detected, select the Find Error option and the window will scroll back to location of the run-time error and the code will again be selected in reverse video.

Get Info is next on the Compile menu and when it is selected, a dialog box like the one shown in Figure 2.21 is displayed.

This box shows information on the active window's program text, code, and heap sizes. The last option in the Compile menu is Options.... When this option is selected, the dialog box shown in Figure 2.22 is displayed.

At the very top of the box you will see options for the symbol table size and Auto Save. A symbol table is used by the compiler to hold names of various identifiers in your code. Unless you are working with a large program, the 32K symbol table default size should be sufficient. You may wish to turn on Auto Save by clicking in the box so that every time you select Run from the Compile menu, your Pascal source code is saved to disk. This can be quite handy if you forget to save your file and somehow crash the system, which is not all that difficult to do sometimes. If you selected Auto Save, at least you know all your code was saved to disk before the system went down. The remainder of the Options dialog box shows five lines; each starts with a dollar

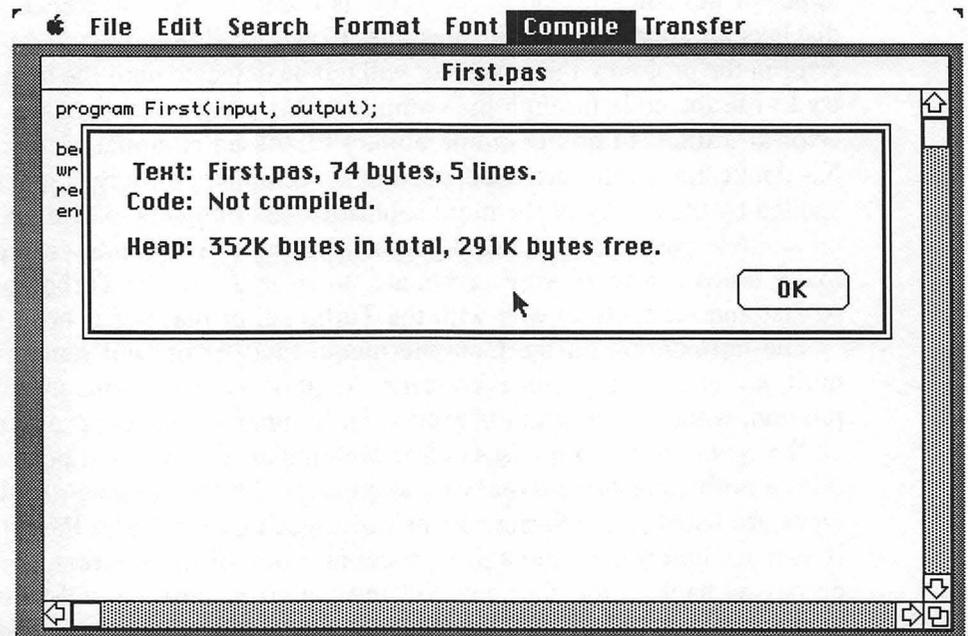


Fig. 2.21.

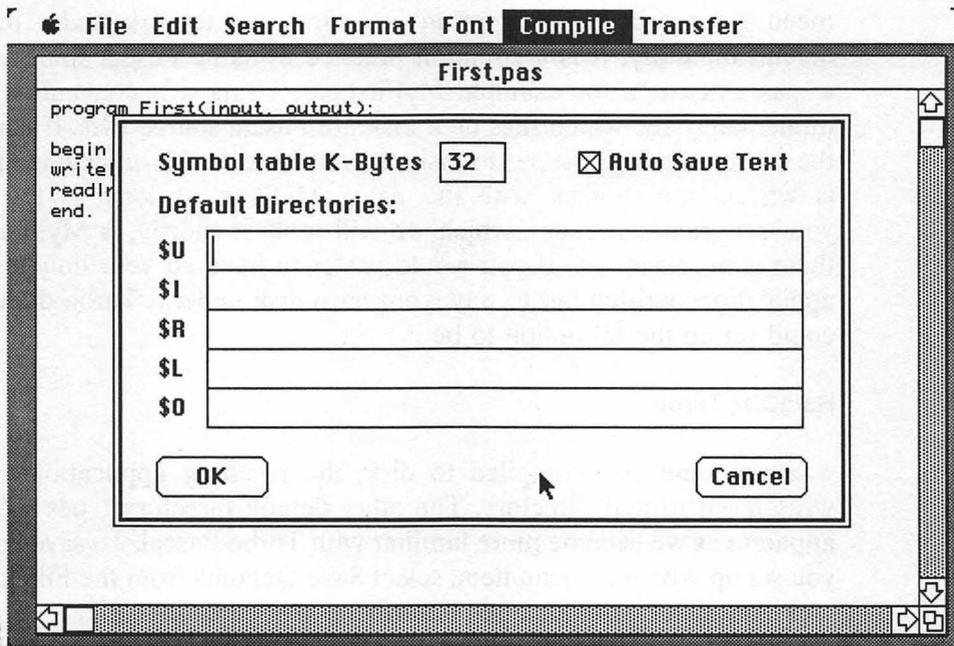


Fig. 2.22.

sign (\$) and shows room for text entries. These are the default directories. These lines may be used to inform Turbo that you wish to refer to special files in different directories. Directories are used to separate different files on your disks. For instance, the line

```
MyDisk:TempDir:
```

tells Turbo to look on the disk named MyDisk and in the TempDir directory. You may have several files on MyDisk within that TempDir directory but you have set them up so that they may be referenced in this manner rather than just having them jumbled about on the disk.

The various default directory types are

\$U	Unit file directory
\$I	Include file directory
\$R	Resource file directory
\$L	Link Object file directory
\$O	Output file directory

Each is described in detail in the Turbo Pascal manual, but we are particularly interested in the \$O directory. When you select To Disk in the compile

menu, your executable program is written out to disk under the current default directory. It is a common practice to name Pascal source files with a .pas extension, for example MyFirst.pas. Using this convention, you can immediately see which files on a disk are Pascal source files. If you compile the source file MyFirst.pas to disk, the resulting double-clickable application is written out to disk with the name MyFirst (as long as the name in your program statement, which we will look at shortly, is MyFirst). Notice there is no extension. If you would prefer to have all your double-clickable applications written out to, say, your hard disk under a Turbo directory, you could set up the \$O option to be

HardDisk:Turbo

Every time you compiled to disk, the resulting applications would be written out to that directory. The other default directory types will become apparent as we become more familiar with Turbo Pascal. To save any options you set up with this menu item, select Save Defaults from the File menu after you have made all your option entries.

The Transfer Menu

The concept behind Transfer was discussed earlier for the Edit Transfer option in the File menu. Any applications installed via Edit Transfer will show up in the Transfer menu (see Figure 2.23).

To switch to any of these applications, select it from the Transfer menu.

Now that I have analyzed all the Turbo Pascal menus, it's time to get down to business and work with your first Pascal program.

Your First Turbo Pascal Program

Up to this point I have discussed some programming concepts without writing any Pascal source code. I cannot emphasize enough how important it is first to analyze a problem and then to determine a general solution before writing the actual algorithm.

When you are ready to begin programming, information or data is entered via the keyboard and mouse in the form of letters, numbers, and special characters. The data is processed by the Macintosh and then put out via video display or a printer. The output is a result of the Turbo Pascal instructions that you provide.

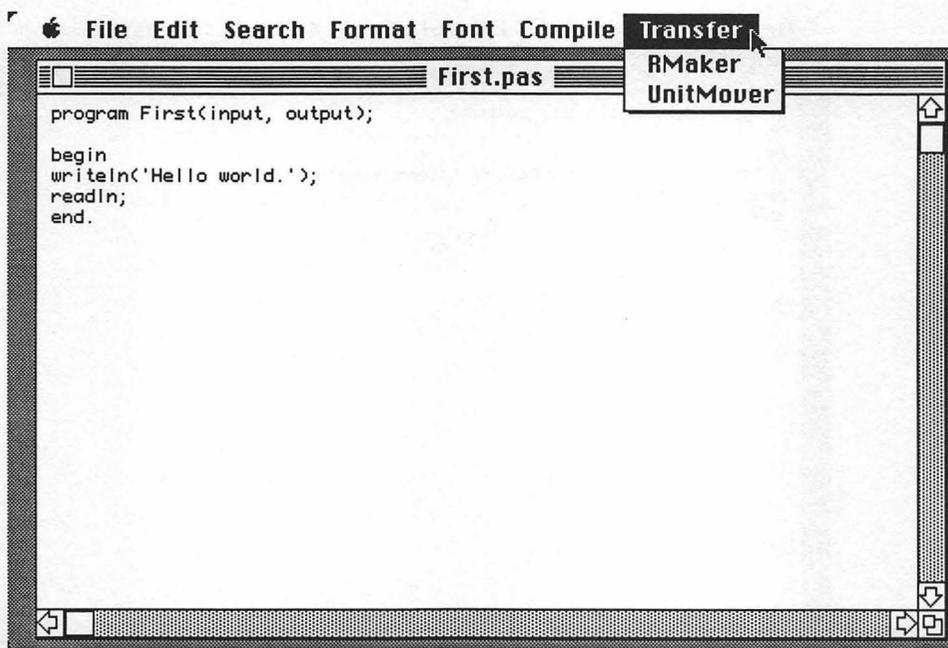


Fig. 2.23.

I will start with a very simple program that displays the following message on your screen: Isn't Turbo Pascal interesting? This message is the output of the program. The message will remain on your screen until you press the return key. Although no data is coming in with the return key, the act of pressing it may be referred to as a form of input to the program.

Start up the Turbo program and in the Untitled window, enter the following short Pascal program:

```
Program MyFirst;  
  
begin  
  writeln('Isn't Turbo Pascal interessting?');  
  readln;  
end.
```

Your screen should look like the one shown in Figure 2.24. If you have never worked with an editor before, just type the statements in as if you were using a typewriter; if you make any mistakes, just backspace over them and reenter the line.

If your program looks like the one in Figure 2.24, select To Memory in the

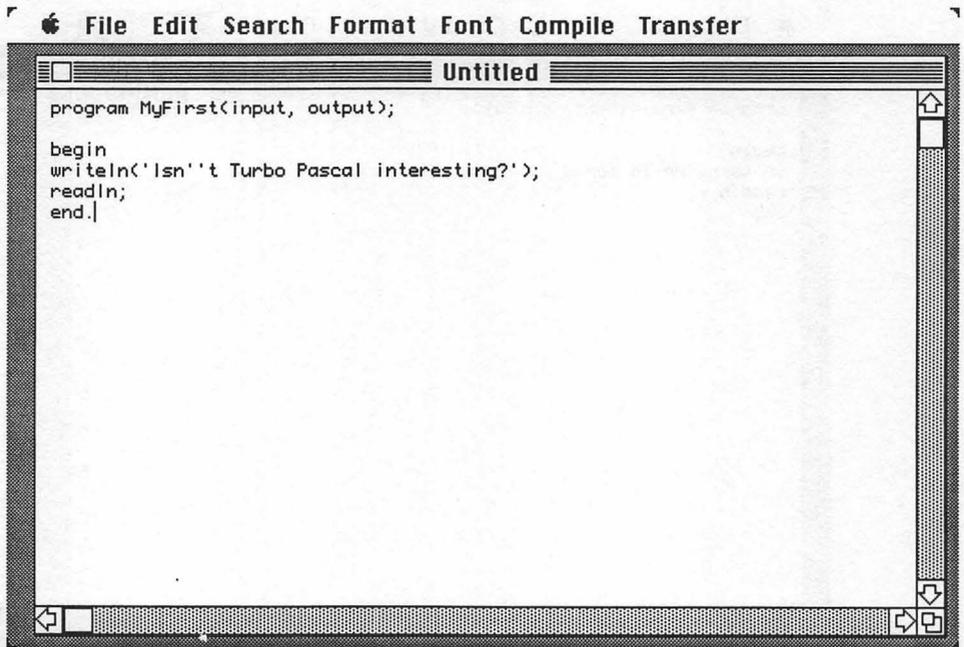


Fig. 2.24.

Compile menu to make sure there are no typos. If all went well, you should be placed back in the edit window. Two items are very important when entering programs in Turbo Pascal: blank spaces and semicolons. If we insert a space between two words, you should too. Otherwise a syntax error may occur. Semicolons could be explained in a chapter all their own. We detail their use in Chapter 3. For now, it is important that you realize that Turbo Pascal is very rigid about them. In general, almost every program line must end with a semicolon. The semicolon is used to separate statements and to indicate to the computer that you are ready to begin a new statement. Many beginners' programming errors involve a missing or misplaced semicolon. When to use and when not to use a semicolon is sometimes confusing. For now, follow the general rule and watch your punctuation closely by carefully copying our examples. Also, you should follow the statement indentation and spacing as presented throughout this book when working with Turbo Pascal. For example, the statements between **begin** and **end** are indented to show the block. Now let's take a look at this simple program.

The first statement—

```
Program MyFirst;
```

—announces the beginning of a program named MyFirst. This is the *program statement* or the *program heading*. All Pascal programs must have a similar

heading. As we mentioned earlier, when you compile this program to disk, the resulting double-clickable application gets its name from this program statement. So a Turbo Pascal source file named `TheFirst.pas` could contain the program statement

```
Program MyFirst;
```

Compiling this program to disk produces an executable file called `MyFirst`.

The next statement in the program is **begin**. You may also have noticed that the program ends with **end**. All Pascal programs have a block of executable lines that start with **begin** and conclude with **end** followed by a period. The executable lines are called the program statements. A program block always follows the program heading. In fact, several program blocks may follow, making up a large structured program. It is important to note that many program blocks within a single program will include **end**, but only the last one should be followed by a period. The period signals Pascal that this is the last line in the program.

Immediately after **begin** comes the statement responsible for displaying the message:

```
writeln('Isn't Turbo Pascal interesting?');
```

This is a *write-line statement*. Its format will be detailed later, but for now you should realize that it is used to display messages.

Finally comes the *read-line statement*:

```
readln;
```

As with `writeln`, `readln` will be discussed in detail later. For now all you need to know is that this statement causes all activity on the Macintosh to cease until you press the return key. Now that we have a rather limited understanding of our first program, let's see what happens when we execute it.

Running Your First Program

Select Run from the Compile menu and watch what happens. If all went well, your screen should look like Figure 2.25, displaying the message `Isn't Turbo Pascal interesting?` This message remains on the screen until you press the return key.

Again, if you ran into a compiler error, check your typing and make sure your program matches the listing shown above. Some programmers spend

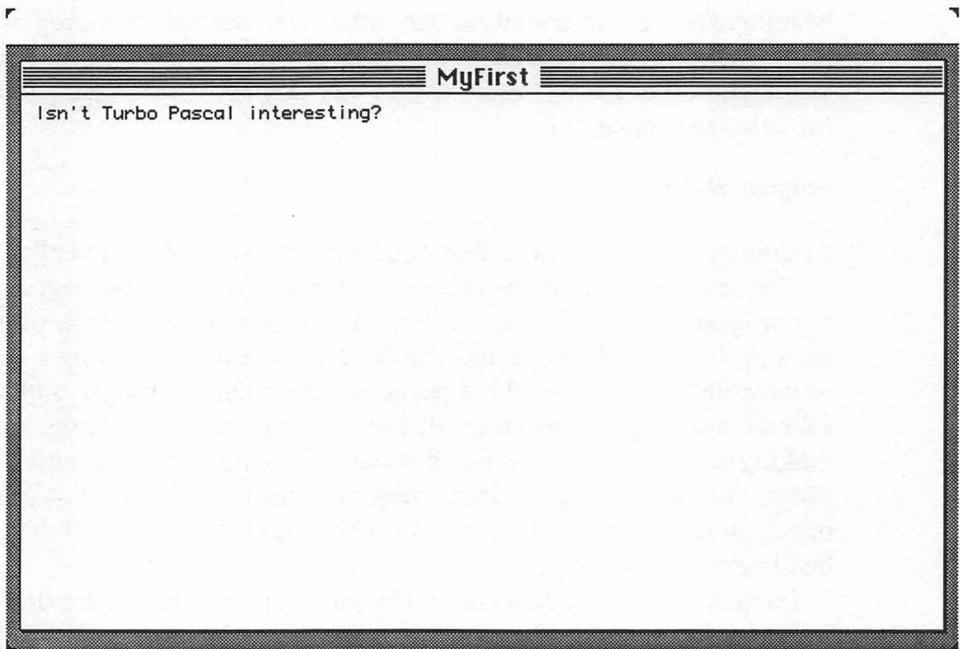


Fig. 2.25.

much time correcting errors, also called debugging a program. Keep in mind that debugging is not just correcting typos. Debugging is more precisely defined as finding and fixing incorrect statements and improper program construction. Correcting typing mistakes is a matter of editing. It is important to review editing tools next.

Editing Your Program

The computer won't notice typing mistakes in a program until you submit the code for compilation. For example, if you enter the following program—

Program Sample;

```
begin
  writeln('Programming is fun!');
  writeln('Have a nice day!');
  readln;
endd.
```

—the computer responds with an error message when you try to compile it because you misspelled the word “end” in the last line of the program. In addition, the line responsible for the error is selected in reverse video.

To correct the error, first respond to the error message by pressing either the mouse key or the return key. The next step is to correct the mistake. You do this using the general editing techniques of the Macintosh. You move the mouse to place the pointer at any point on a program line, backspace to erase, hit the space bar to insert, or drag to select text. Selected text is erased by pressing the backspace key or replaced by typing the desired text.

To correct the spelling error, move the pointer to the extra “d,” select the letter by dragging the pointer horizontally, and then press the backspace key to delete the letter. Now run the program by selecting Run from the Compile menu. The resulting output should look like Figure 2.26.

You may want to modify a program by inserting a statement between previously entered items. Insert a line between your two `writeln` statements like this:

1. Move the pointer to the beginning of the second `writeln` statement.
2. Enter the new line as

```
writeln('This is a Turbo Pascal Program.');
```

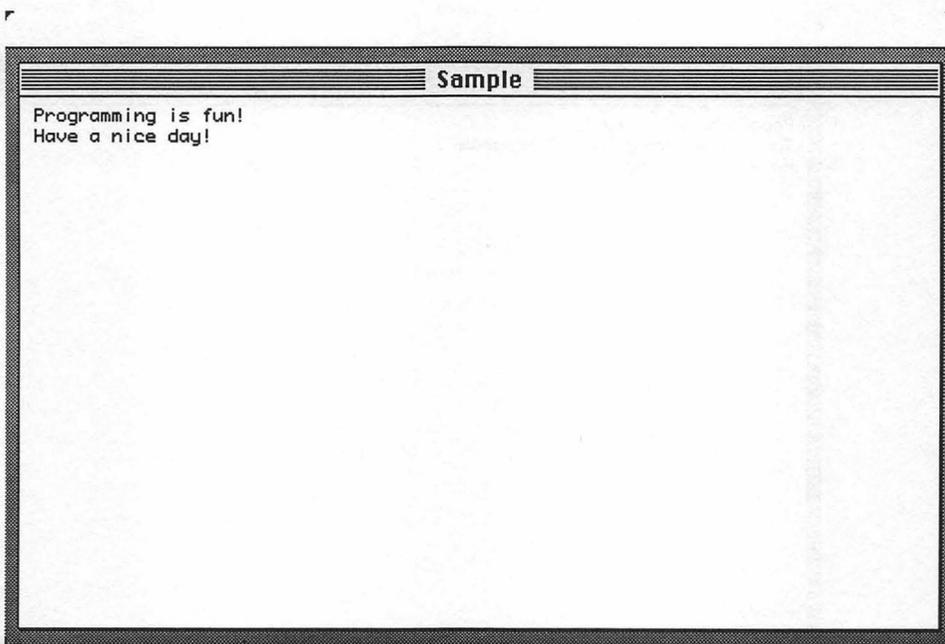


Fig. 2.26.

Now if you run your program, the result should look like Figure 2.27.

You can also use the Cut, Copy, and Paste functions in the Edit menu as well as the functions in the Search menu. You should take some time to familiarize yourself with these edit functions before we move on to the next subject, saving your programs.

Saving Your Program

Recall that the Run command in the Compile menu will compile your program, create an executable version of it in 68000 machine code and place it in RAM. This command then executes the program but does not save it to disk. The option To Disk in the Compile menu must be used to write an executable version of your program to disk. If you compile to disk and leave Turbo without saving the source code, you will be able to run your program from the desktop, but you will not have the code in the editor. For that reason, you should periodically select Save from the File menu to save your source code. Let's save a copy of the following sample program to disk by selecting the Save option in the File menu.

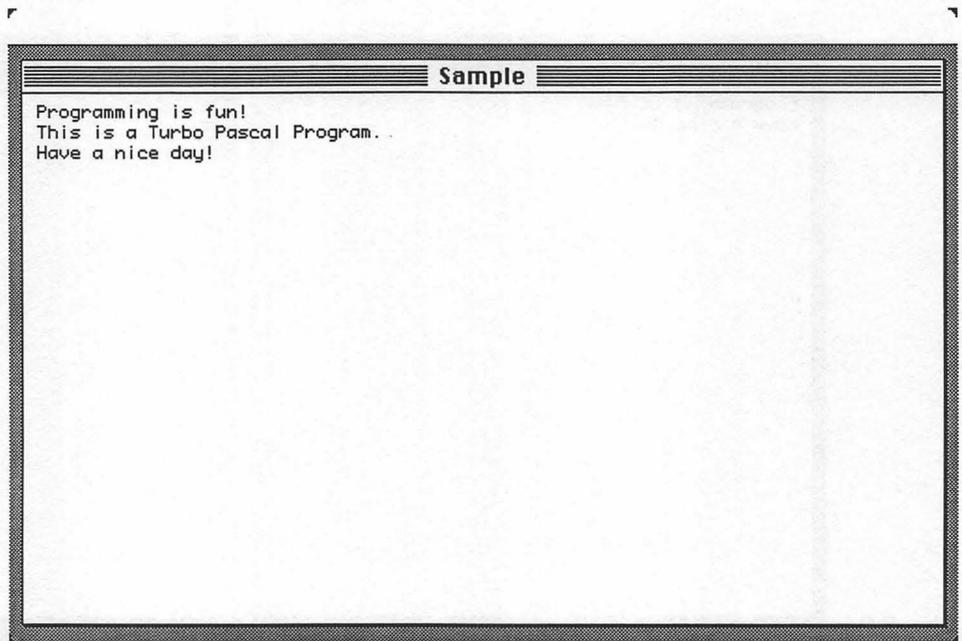


Fig. 2.27.

Program Sample;

```
begin
  writeln('Programming is fun!');
  writeln('Have a nice day!');
  readln;
end.
```

Save displays the dialog box shown in Figure 2.28. In this box specify the name of your source file; I will call mine Sample.pas. We mentioned earlier that it is a common practice to flag Pascal source filenames with the .pas extension.

Alternatively, you may select Save As... from the File menu. Recall that Save As... always displays the dialog box prompting for the filename, whereas Save displays the box only when the active window is untitled. You can use Save As... later on if you decide to modify the sample program but not overwrite the original version. Once you have saved the sample program to disk, your edit window should look like the one shown in Figure 2.29. Notice that the window header now contains the name of the source code file.

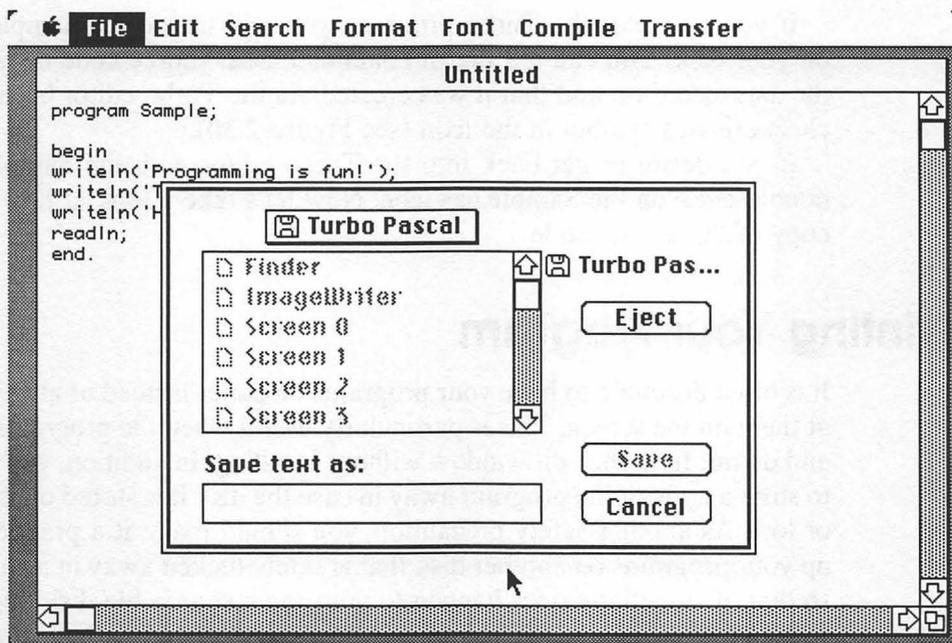


Fig. 2.28.

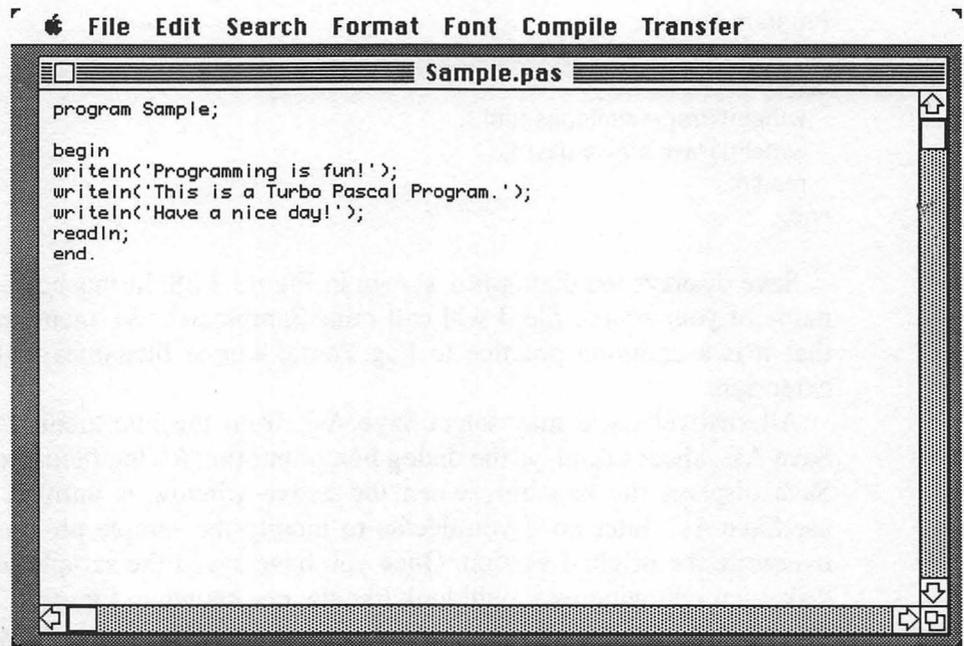


Fig. 2.29.

If you now exit the Turbo program, you will notice the Sample.pas icon on your disk. You can tell that it contains Pascal source code because it has the .pas extension and that it was created via the Turbo editor because of the checkerboard symbol in the icon (see Figure 2.30).

If you desire to get back into the Turbo editor with the Sample.pas file, double-click on the Sample.pas icon. Now let's take a look at how to print a copy of the source code.

Printing Your Program

It is often desirable to have your programs on paper instead of always looking at them on the screen. This is particularly useful when the programs are large and do not fit in the edit window without scrolling. In addition, you may want to store a copy of the program away in case the disk it is stored on is damaged or lost. As another safety precaution, you should make it a practice to back up your programs on another disk that is safely tucked away in a storage case so that, if something does happen to your main source file disk, the contents won't be lost. Finally, it's sometimes necessary to study a long program for

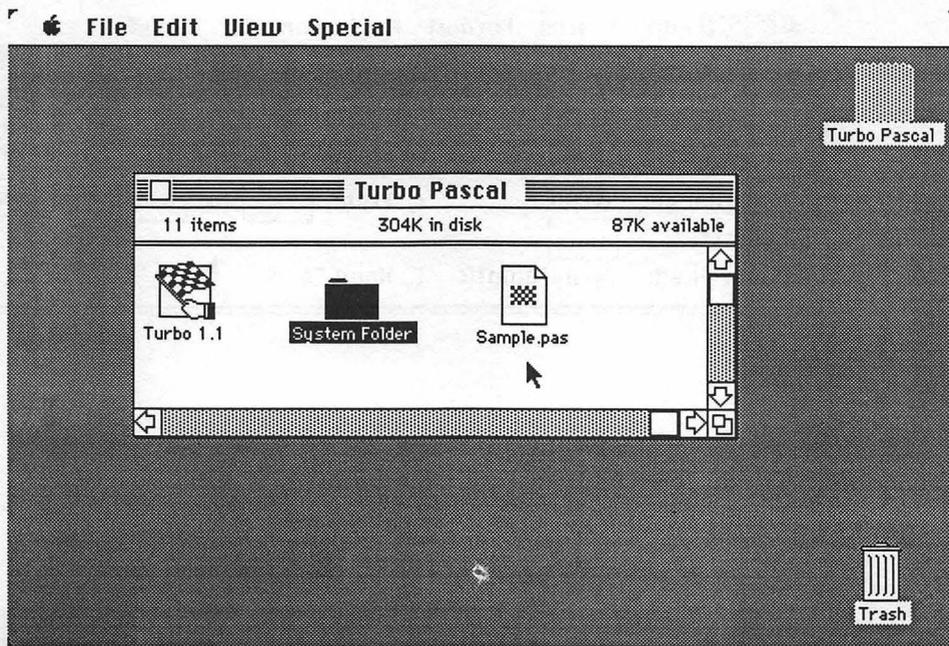


Fig. 2.30.

errors, making changes, and so on. A hard copy of a program makes all these tasks easier.

Printing a hard copy is a simple procedure. Make sure that your printer is properly connected to your Macintosh, that it is plugged into an electrical outlet, and that the paper is fed correctly. Next select Print... from the File menu, enter the desired options in the dialog box shown in Figure 2.31, and watch as the listing is printed.

The Macintosh provides two printing features you may find useful. To print a copy of the screen (sometimes called a screen dump), engage the caps lock and press `<COMMAND><SHIFT><4>`. The command key is the one with the cloverleaf on it. Alternatively, you can dump the screen to a disk file by pressing `<COMMAND><SHIFT><3>` while the caps lock key is locked down. You can edit Screen 0, the file created by this option, via MacPaint or any comparable drawing program. Any subsequent screen dumps sent to the disk are named Screen 1, Screen 2, and so forth.

This chapter covers a lot of ground and explores several important operations you will be using throughout the remainder of this book. In Chapter 3 I will start looking at the structure and syntax of Turbo Pascal programs.

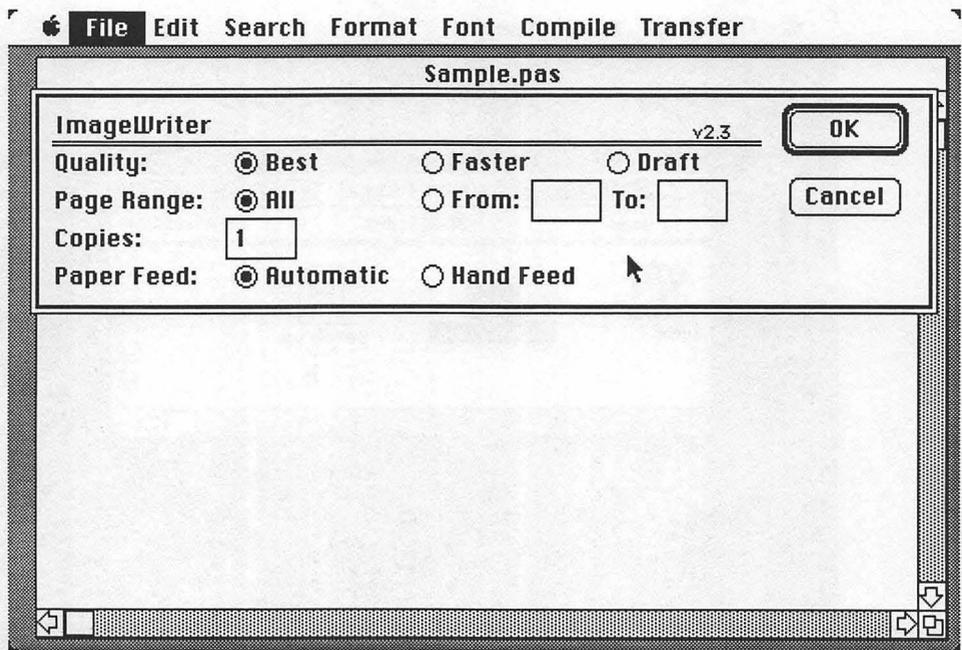


Fig. 2.31.

Review Summary

1. The menu bar in Turbo Pascal has eight entries: the Apple menu, File, Edit, Search, Format, Font, Compile, and Transfer.
2. Turbo Pascal supports up to eight windows open simultaneously; the operations for using windows in Turbo are simple extensions of the same operations performed on other Mac software such as MacWrite.
3. Information, or data, is put into the Mac via a keyboard or mouse in the form of letters, numbers, and special characters. The data is processed by the Mac, and then put out via an external device such as the video display or printer.
4. All programs must begin with the program heading followed by at least one program block.
5. A program block starts with the reserved word **begin** and concludes with the reserved word **end** followed by a period (.). All the statements within a program block are the executable lines of the program. In most instances, a Pascal statement is terminated by a semicolon (;).
6. Turbo Pascal is not case-sensitive. That is, you can enter your programs in either upper or lower case or a mixture of both and Turbo will not differentiate between the two.

7. Editing your programs is made simple with the Edit menu. Just like MacWrite and other word processors, you can cut, copy, and paste as well as use the Clipboard for temporarily storing text.
8. Storing your programs to disk is accomplished with the Save or Save As... command in the File menu.
9. To print a hard copy use the Print option in the File menu.

Quiz

1. Does the Turbo editor provide a file wrap feature on the Search options?
2. What does the Clear option in the Edit menu do?
3. What is the difference between Stack Windows and Tile Windows in the Format menu?
4. What is the difference between compiling to disk and compiling to memory?



THE TURBO PASCAL LANGUAGE

Turbo Pascal Revealed: Structure and Syntax

The Right Stuff: Syntax
The Elements of a Turbo Pascal Program
Data Types: What to Declare
What to Declare: Variables
Assigning Variables
Constants
A Word on Punctuation: The Semicolon
Expressing Yourself
Mathematical Order of Operation
Simple Turbo Pascal Arithmetic
More on Data Types
Program Formatting
Review Summary
Quiz

In this chapter you will learn:

- The proper structure and syntax of a Turbo Pascal program.
- What the different types of data are.
- What a variable is.
- How to use comments and constants.
- How to use mathematical operations for arithmetic.

This chapter focuses on many basics necessary to use Turbo Pascal on the Macintosh. The first step in programming is to analyze the problem and develop a general solution, or algorithm. Next the solution is translated into Pascal as instructions and implemented to test the results. From now on I will emphasize writing program instructions developed from algorithms.

Before you can begin to solve complex problems, you must master some fundamental concepts of the language itself. The first concept is the structure of a Pascal program and the elements that make up a simple program. Later in this chapter I will introduce the variable and show how this feature is

invaluable for complex programming problems. Assuming Turbo Pascal is loaded, up, and running, let's get started by reviewing the fundamentals of Turbo Pascal programming.

The Right Stuff: Syntax

In any language certain rules must be followed. In a natural language such as English these rules are called grammar. In programming these rules are called syntax. A program in Pascal must be properly constructed according to strict rules. Programs must be syntactically correct, using valid statements. If the rules of syntax are not followed—for example if a semicolon is misplaced or a key word is missing—the program will fail. Throughout this book you will learn many syntax rules. In the next section I introduce a few of the most common ones.

The Elements of a Turbo Pascal Program

A valid program must include a few elements. These elements, introduced in Chapter 2, make up the structure of a Pascal program. Let's review the elements of a simple program step by step. The basic organization includes a program name or heading, declarations and/or definitions, and the program body, as shown in Figure 3.1.

A complete program might look like Figure 3.2.

The First Line

The construction of this example begins with the first element, the program heading, which must be the first line of the program. A program name, called an identifier in Turbo Pascal, can be made up of capital or lowercase letters, numbers, and underscores (_). Although an identifier may be as long as you like, only the first 63 characters are actually used by Turbo Pascal. All identifiers *must* begin with a letter. The rules governing the program identifier are a bit more strict: a program name cannot be longer than 31 characters. When you compile to disk, the resulting executable file's name is the same as the name in the program statement, so keep program names within this limit.

Two important rules of syntax are illustrated with the program heading. First, a space must separate the word "program" from the program name. Second, a semicolon must immediately follow the program name. Both punctuation marks are necessary and should be mastered early on.

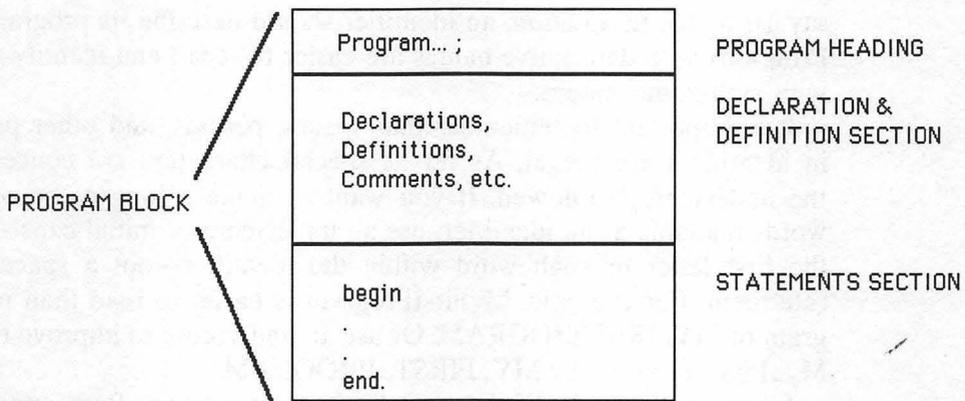


Fig. 3.1.

A Word on Identifiers

Turbo Pascal provides a great amount of flexibility when naming identifiers. Early versions of Pascal and other high-level languages were restricted in the number of characters allowed for identifiers or similar naming procedures. Some versions allowed only one or two characters; others allowed more, but only the first two characters were significant and recognized by the computer. In Turbo Pascal you can use as many characters as you desire. However, you may want to limit the number of characters to a manageable number,

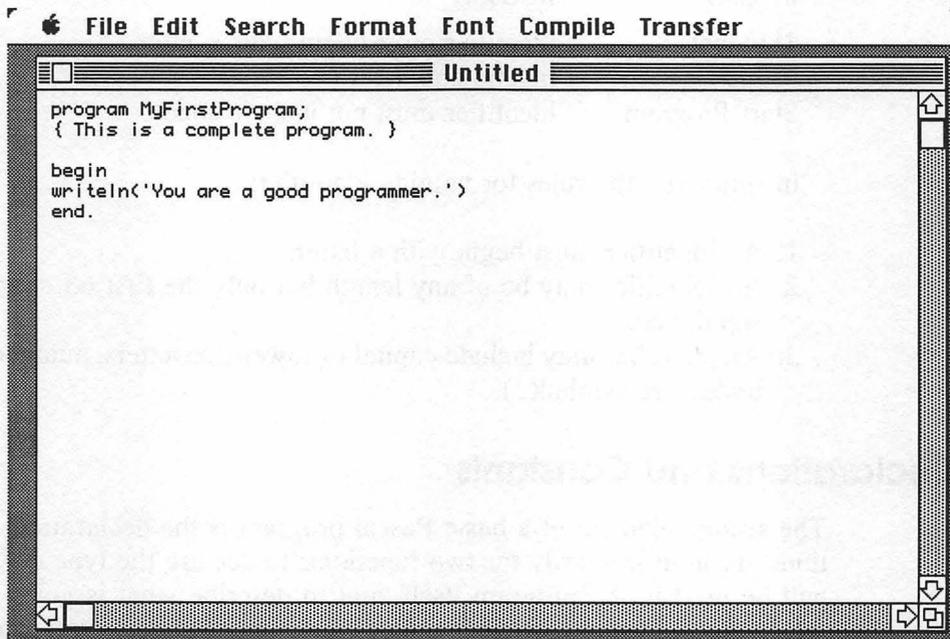


Fig. 3.2.

say 10 to 15. In addition, an identifier should describe its program or item. Programs with descriptive names are easier to recall and identify than those with ambiguous names.

It is important to remember that spaces, periods, and other punctuation in identifiers are illegal. As far as special characters are concerned, only the underscore is allowed. If you want to make a long name or separate words readable as an identifier, use an underscore or initial caps—capitalize the first letter or each word within the identifier—not a space, to separate them. For example, MyFirstProgram is easier to read than myfirstprogram or MYFIRSTPROGRAM. Or use an underscore to improve readability: My_First_Program or MY_FIRST_PROGRAM.

Remember, all identifiers must begin with a letter. Both uppercase and lowercase characters are recognized as the same by Turbo Pascal and are interchangeable. Any other change in the first 63 characters makes the identifier unique. For example, A1 is different from A2, and SimpleGreeting is different from Simple_Greeting. The following is a list of some valid and invalid identifiers:

Valid

First_Sample	Game1	Payroll
Employee25	F8	Start_Add_Here

Invalid

1Student
Value#
Start Program

Reason

Identifier must begin with a letter
Invalid character (#)
Identifier must not include space

In summary, the rules for naming identifiers:

1. An identifier must begin with a letter.
2. An identifier may be of any length but only the first 63 characters are significant.
3. An identifier may include capital or lowercase letters, numbers, and the underscore symbol(_).

Declarations and Constants

The second element of a basic Pascal program is the declarations. Declarations are used primarily for two functions: to declare the type of values that will be used in the program itself, and to describe what is going on within the program at any particular point. Declaring values is explained in detail in the following section, which describes variables. In Figure 3.2 the second

line of the program represents the descriptive use of a declaration. This type of declaration is called a comment.

Comments are useful tools employed by good programmers. These lines are ignored by the computer except under special conditions explained below but make a program easily understood. A comment should explain what is going on within a program. A properly documented program can be read by someone besides the programmer without too much clarification.

A comment is enclosed in either a set of { curly brackets } or (*parentheses and asterisks*). The matching delimiters are not interchangeable. For example, the comment line in Figure 3.2 can also be written as

```
(* This is a complete program *)
```

but not as:

```
{ This is a complete program }
```

The comment line may contain anything, but avoid using another curly bracket or parenthesis/asterisk pair like this:

```
(*{ This is an embedded comment }*)
```

This is commonly known as a comment within a comment, or an embedded comment. Although Turbo Pascal permits you to do this, it is not a good practice because so many other compilers treat embedded comments as syntax errors.

Because almost all comments are ignored by the computer, they may appear anywhere in a program. Special comments, which begin either (*\$ or a {\$, indicate a compiler directive. Compiler directives are discussed in detail in Chapter 4. For now all you need to know is that all comments are ignored by Turbo Pascal unless the first character of the comment is a dollar sign. If you use a comment to describe a particular line, place it after the semicolon delimiter of the same line. For instance,

```
writeln ('Hello world. '); {This is a writeln statement}
```

If you are explaining a complete block of code, it might be more understandable to place a comment block before it like this:

```
(*****  
(* Now we want to add the taxable amounts and subtract *)  
(* the nontaxable amounts to get the net income. *)  
*****)
```

followed by your actual Pascal statements. The use of comment blocks similar to this greatly improves the readability of programs and makes the comments easy to locate. Commenting on your code is a very important step in the development of programs. Many a programmer has gone back to a piece of code a few years or even days after writing it, and because it was not properly documented, cannot figure out its purpose.

The Program Body

The *program body* is the workhorse element of a Turbo Pascal program. The structure of this element is preceded by the reserved word **begin** and is completed with the reserved word **end**. The lines listed between the reserved words **begin** and **end** are the executable portion of the program and are called statements. Collectively, these lines represent a program block.

A number of statements may appear with the **begin-end** pair. Our example in Figure 3.2 contains only one statement, which you were briefly introduced to in Chapter 2. `Writeln` (pronounced “write line”) tells the Macintosh to write the message and move the cursor to the next line. This statement is used to send information to an output device such as your screen, a printer, or a file. In this instance we are requesting the output to go to the screen. Examine the syntax of this statement. The information that you wish to display on your screen must be enclosed within parentheses and apostrophes exactly as it appears in Figure 3.2. The statement may conclude with a semicolon. However, because it is the last statement that precedes an **end**, the semicolon is not required. You may recall that earlier we mentioned that the use of the semicolon is a tricky subject. Up to now we understood that all statements concluded with a semicolon. The only exception to this was the very last **end**, which is followed by a period to signify the end of the program. Nor is any statement immediately preceding an **end** required to have a semicolon. For example, this program contains three `writeln`s, and all but the last one are concluded with a semicolon.

Program Test;

```
begin
  Writeln ('This is a test. ');
  Writeln ('This is only a test. ');
  Writeln ('For the next 60 seconds...')
end.
```

The last `writeln` does not require a semicolon because the **end** signifies the end of the block. Pascal recognizes that the statement **end** terminates

the block and therefore also the preceding statement. I will discuss new exceptions to the semicolon rules as they arise. Later in this chapter I summarize these rules for semicolons.

Writeln displays the characters desired and then performs a carriage return and drop the cursor down to the beginning of the next line. We will explore this statement in more detail shortly.

At least one program block must follow a program heading. It is important to note, however, that several blocks may appear throughout a program. Where several program blocks are used as modules, structured programming occurs. Whenever **begin** appears, it marks the starting point for one or more executable statements; **end** marks the stopping point for the same. Each module is often used to perform a specific programming job. This structure is the basis for subdividing large programming problems into smaller, workable program modules.

Each program block must include a **begin-end** pair. These pairs, or compound statements, follow the syntax rules applied to a program that contains only one **begin** and **end** pair. When **begin** appears, an **end** must follow one or more executable statements. The one notable syntax rule is the punctuation following **end** within the compound statement. A period must always follow the last **end** to indicate that the program is complete. An **end** that is not at the end of a program must be followed by a semicolon. Since only one **end** followed by a period should appear, at the end of the program, this rule is easily mastered.

In summary, the rules for using the compound **begin-end** pair are as follows:

1. At least one **begin-end** pair must follow a program heading.
2. Where a **begin** exists, a matching **end** must follow.
3. Only the last **end** should be followed by a period. Many programs contain several program blocks, hence a number of **begin** and **end** pairs. An **end** that is not the last one is followed by a semicolon.

Program Review

The program in Figure 3.2 includes two elements of a Turbo Pascal program: the program heading and the executable body, bounded by **begin** and **end**. The comment, you recall, is used to clarify programming intentions and is ignored by the compiler. The result is Simple_Example displays this message on the screen: You are a good programmer. Selecting Run from the Compile menu produces this result (see Figure 3.3).

For more complex programs we need to explore the third element, the

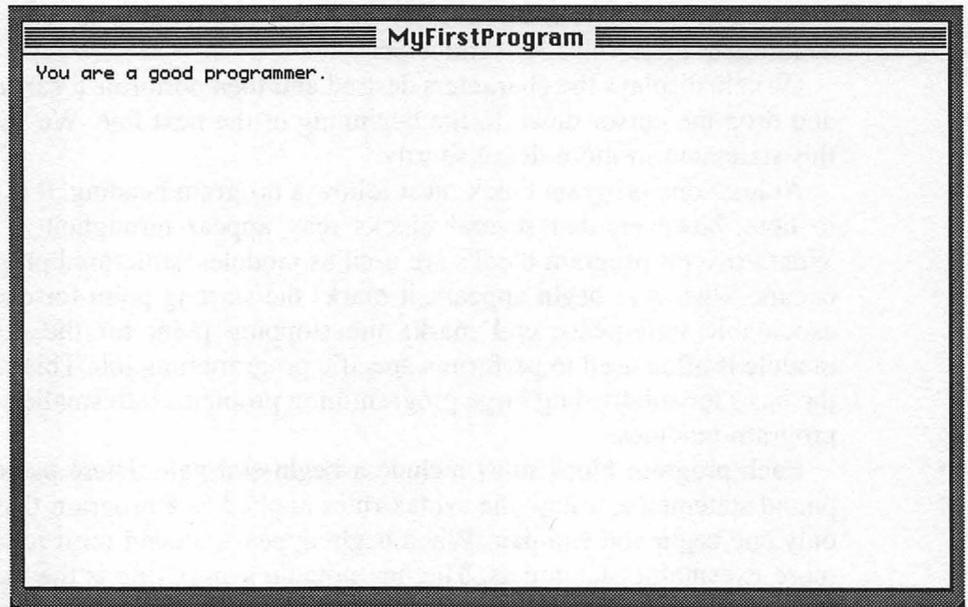


Fig. 3.3.

declaration section, in more detail. This section is used to define the type of data processed within the program itself.

Data Types: What to Declare

In programming, data is represented by letters, numbers, and special characters, including standard punctuation marks. Before information can be processed, the characters must be defined. In this section I review the common data types required for short programs. These data types include integers, real numbers, chars, Boolean values, and strings.

Integer

An integer is a whole number; it may be either positive or negative but cannot include a comma. Signs, positive (+) and negative (−), are allowed. If a sign is omitted, the positive is assumed. Although the number of integers is infinite, there is a limit to the number available on the Macintosh in Turbo Pascal. The largest number represented by an integer is 32767; the smallest is −32768. Examples of valid integers include

5 3 −14 0 1836 46 −135

The maximum and minimum values for integers are determined by the fact that Turbo Pascal allots 2 bytes of memory for each integer. In Chapter 1 I discussed how the Macintosh uses ASCII representation to interpret one-byte values as characters. Each byte consists of 8 bits, and the value of the byte is determined by exactly which of its bits are on and which ones off. The standard integer type contains 2 bytes, or 16 bits. You are already quite familiar with decimal, or base 10, numbers. Base 10 is the number system we all use. We represent numbers in base 10 with the digits 0 through 9. Computers work with base 2, or binary, numbers, using the digits 0 and 1. That's it. There are only two values that may occupy any position in a number in base 2; the position is either 0 or 1, just as computers understand only two states: on and off, zero and one, yes and no. Take a look at the following base 10 numbers and their binary equivalents:

<u>Base 10</u>	<u>Binary</u>
0	0000000000000000
1	0000000000000001
2	0000000000000010
3	0000000000000011
4	0000000000000100
5	0000000000000101

Notice that I show all the bit positions of the binary numbers, so there are always 16 digits. This is not necessary, but want to reinforce the fact that the computer views an integer as a 2-byte or 16-bit value. The binary equivalent of 0 is 0000000000000000. That is no surprise. The binary equivalent of 1 is 0000000000000001. That should not be any surprise either. However, the binary equivalent of 2 is 0000000000000010, and that relationship may not be so obvious. Since 1 is 0000000000000001 in the binary system and there are only 2 digits for each position (0 and 1), we must slide over to the next position to represent a 2. It's just like the next whole number above 9 in base 10; there are no more unique digits, so another position must represent higher values. Another way to look at binary numbers is to consider that each digit except the one on the right in the number is some multiple of 2, just as each digit except the one on the right in a decimal value is some multiple of 10. For example, the decimal value 135 may be interpreted as follows:

$$\begin{aligned}1 \times 100 &= 100 \\3 \times 10 &= 30 \\5 \times 1 &= 5\end{aligned}$$

The digits in boldface are the digits in the number 135. The 1 is multiplied by 100 because it is in the 100s position. The 3 is multiplied by 10 because it is in the 10s position. Finally, the 5 is multiplied by 1 because it is in the 1s position. Actually, each digit in the number 135 is multiplied by a value of 10 raised to a power (10^0 , 10^1 , 10^2 , and so on). The powers increase as you go from right to left in the number. If you add up the results of the multiplications, you wind up with the original number: 135. The same principle holds true with binary numbers; the only difference is the naming of the positions. The 1s position is the same as in base 10, but the other positions are represented by powers of 2 (2^0 , 2^1 , 2^2 , and so on). So the binary number 000000000000010 is equal to 2^1 , or 2, in base 10. As another example the base 10 equivalent of 000000000001111 is $2^0 + 2^1 + 2^2 + 2^3$, or $1+2+4+8 = 15$.

After this crash course in binary numbers, you know how the computer looks at 2-byte integers. At least you know how positive integers are represented; what about negative integers? When working with signed integers, the Macintosh (along with quite a few other computers) uses the high-order bit to determine the sign. The high-order bit is the leftmost bit. You might think that -1 is 100000000000001, but it is not. The Macintosh uses what is known as 2's complement notations for representing negative numbers. In short, this means translating a negative number to its absolute value requires changing all 1s to 0s and 0s to 1s and then adding 1. That sounds rather odd at first, but that's how it works. For example, -1 in binary is 111111111111111. You can see that this value is negative because the leftmost bit is 1. Switching all the 1s to 0s and the 0s to 1s yields 000000000000000. Adding 1 gives 000000000000001, which is equal to 1.

Now that you know about integer values in Turbo Pascal, let's look at how to represent numbers with fractional parts.

Real Numbers

Real numbers may contain fractional parts. Generally, at least one digit is placed to the right of the decimal point to indicate the fractional part of the real number. These numbers can be expressed in two ways. One method is to use the decimal point and place a number of digits to the right of the decimal point; for example, 4.12, 123.4, 0.234. If the number is negative, the negation sign is placed before the number, for example, -4.23 , -36.2 . The second method of notation, which requires a little explanation, is called scientific or exponential notation.

Scientific notation is used to display very large or very small numbers. For example, the number 5 million, represented as 5.0×10^6 , is written thus:

5.0E+6

The number 5.0 is called the *coefficient*, or *mantissa*, and the number +6 is called the *exponent*. The 5 is the whole part of the number to the left of the decimal point, while the exponent represents the power of 10, or the number of places to the right of the decimal point. The letter E means exponent. Therefore, to read a number written in scientific notation, move the decimal point of the mantissa to the right the number of places specified by the exponent. Another example: 4.34E+2 represents 434.

Similarly, very small numbers are represented with scientific notation. For example, the real number represented by 0.00001234 is written thus:

1.234E-5

The negative exponent means that the decimal point of the mantissa should be moved 5 spaces to the left instead of the right. For a negative real number place the negation sign in front of the mantissa, that is, -4.52E-3, which represents -0.00452.

A unit of real data occupies 4 bytes of memory. The real type is identical to the single type used for declaring real numbers. The format used to store real values in memory is based on the Standard Apple Numeric Environment (SANE) Library. This library is discussed in Chapter 26 of your Turbo Pascal manual.

As discussed earlier, the computer may display using scientific notation. Such numbers are often called *floating-point* numbers and may contain fractional parts. We have just introduced the exponential or E notation used for very large and small numbers. Turbo Pascal provides for other variations of the data types involving both real numbers and integers. Such data types include double-precision, extended real, and comp real numbers, which provide a greater degree of accuracy than the single-precision real numbers just discussed. Actually, real numbers are converted to the extended type before any mathematical operations are performed on them. Therefore it is more code-efficient to use extended data instead of any of the other real types so that this conversion does not always have to take place. However, the extended type occupies 10 bytes of memory compared with the 4 bytes required by a single or real variable. So you must determine whether processing speed is more important than memory space or vice versa. For the remainder of this book we will use ordinary single-precision real numbers.

Characters and Strings

Numbers are not the only data type that can be manipulated with Turbo Pascal. Text information represented by upper- and lowercase letters and special characters can also be used. Such information is represented by the data types `char` (character) and `string`.

Both characters and strings are represented by enclosure in single quotes or apostrophes. Examples of chars include

```
'A' 'b' '$' '?' '8'
```

How does the value `'8'` above differ from an integer that contains the value 8? To represent an integer that contains the value 8, look like this:

```
0000000000001000
```

The bit in the 8 position is on and is the only one on, so the value is 8. However, a `char` type occupies only one byte, and we represent the value `'8'` within it by using the ASCII value for 8 (see ASCII table in Chapter 1) like this:

```
00111000
```

As you can see, there is a definite difference between `'8'` and 8, so be careful not to get the values of integers and chars or strings confused.

Because a single quote is used to denote the beginning and end of chars and strings, to represent a single quote itself, the quote must be written twice:

```
''
```

We can also represent alphanumeric information as a sequence of characters instead of just one character at a time. This data type, the `string`, is particularly useful for displaying messages like the ones we have seen in our earlier simple programs. For example:

```
'How old are you?'
```

or

```
'Isn't Turbo Pascal interesting?'
```

Notice the double apostrophe in the word “Isn’t”; again, because the single apostrophe is used as a delimiter to show the beginning or end of a string, a double apostrophe in the string represents an actual apostrophe. Strings may contain up to 255 characters on a single line, enclosed within single quotes, of course. It is important to note that a string *must* end on the same line it began. The following is not syntactically correct:

```
writeln ('Hi there!);  
writeln (How are you today?');
```

Some programming languages allow you to write code like this, but if you try it in Turbo Pascal, you’ll get a syntax error. Sometimes this error arises because you forgot to put the apostrophe at the end of the first string and the beginning of the second one. Either way, because the compiler does not find the end of the string (denoted by another apostrophe) before it gets to the end of the statement’s line, it is treated as an error.

A string is declared by specifying the maximum size of the string. For example, a `String[5]` can hold up to 5 characters. A `String[20]` can hold up to 20 characters. By specifying the maximum string size at declaration, you tell Turbo Pascal how many bytes of memory to reserve for your strings. How many bytes do you think a `String[5]` requires? If you said 5, you’re on the right track but not quite right. A character may be represented in 1 byte, but when working with strings, you need to know exactly how long they are, not just the maximum size. For instance, the `String[5]` declared earlier may hold the string.

```
'Joe'
```

If this is the case, only 3 of the bytes within that string are in use or significant. For this reason all strings in Turbo Pascal are preceded by a 1-byte-length indicator that tells exactly how many characters are in the string. So if ‘Joe’ is in our `String[5]` type, the 1-byte-length indicator would hold the value 3. However, if the string

```
'Sarah'
```

is in our `String[5]` type, the length indicator holds the value 5. You should be able to see that this length indicator allows you to place strings in longer string types; that is, to place “Joe,” which is 3 characters long, in a string type 5 characters long. Now how many bytes do you think a `String[5]` requires?

The answer is 6; the string holds up to 5 characters and requires 1 byte for the length indicator.

A string type may be declared without specifying size. This is done by using the type string as opposed to String[5]. This is the same as declaring the string to be of type String[255] since the string type without a size reserves a block of memory large enough to handle the largest definable string (255 characters). Also, a string that contains no characters is called a *null* string. A null string's length indicator is set at 0.

Boolean Data

Another data type available with Turbo Pascal is called the Boolean type, in honor of logician George Boole. Boolean data occupies 1 byte of memory and has only 2 logical values, true and false. With the Macintosh the value true is 1 and the value false is 0. Boolean data represents answers to questions and is generally used for testing conditions. We will explore programs that have the ability to choose between true and false conditions in Chapter 4. The concept of Boolean values is a very powerful feature of programming in Pascal.

What to Declare: Variables

In this section we discuss a fundamental concept of Turbo Pascal and programming in general. What is a variable? You may recall from basic algebra that a variable is something that changes. Computers use variables in much the same way. A variable can change. Specifically, data values are stored in memory at different locations. Each location may contain a value that can change. The computer must have the ability to manipulate data and change its values.

In algebra a common expression is $y = x + 2$. The value of y changes as the value of x changes; x and y are called variables because their values can change. Now visualize an empty box marked "x". This box represents a location in memory, and it can accept only one item at a time. You can put any value you wish in this box, as if it were a constant value, until you change or remove it.

On the Macintosh (or on any computer) each box is represented by a numeric address or location. The various versions of the Macintosh have anywhere from 128,000 to over 8 million of these locations in RAM. Turbo Pascal allows assigning a descriptive name called an *identifier*, to these locations. Using identifiers to represent specific locations in memory is much

easier than using numeric addresses. Each identifier must have a data type associated with it in order for Turbo Pascal to understand how you intend to use it. Let's look at how to assign some of the data types we already know about in Turbo Pascal.

Because Turbo Pascal must know what sort of data to associate with a particular variable, you must define or declare a data type before it can be used. The data type must be consistent with the type of information assigned to the variable. For example, integers are declared as integer data types; real numbers are declared as real data types, text is declared as char or string data types, and so forth.

To declare a variable, or block of variables, use the reserved word **var** (a complete list of reserved words in Turbo Pascal is presented in Appendix B). **Var**, or the variable section, is placed after the program statement and prior to the first program block, or **begin-end** pair. The **var** section denotes the declaration of variables that will be used in the program. Variables themselves are identified by a variable declaration statement. Such a statement consists of the variable's name or identifier, a colon, and the type of the variable identified. A semicolon follows a variable declaration to separate multiple statements. If you want to declare more than one variable of the same data type, you can type all the variables on the same line, separating them with commas. The following are examples of valid variable declaration statements:

```
Var  
count           : Integer;  
battingAverage : Real;  
answer, response, selection : Char;  
last_Name      : String[20];
```

Before moving on I want to emphasize that when providing variable identifiers, we *strongly* recommend using identifiers that are descriptive of the variable defined. Identifiers may consist of as many characters as you wish, although only the first 63 are significant to the compiler. While there is nothing technically wrong with defining an identifier with a single character, readability should be a major consideration for proper program documentation. For example, a variable that represents someone's grade point average is better suited by an identifier that is descriptive of its use rather than the letter x:

```
Var  
gradePtAve : real;
```

Assigning Variables

In the box example I state that a variable inside the box, or memory location, can contain any value. To do this, you need a method for putting values into a memory location. You can assign values to a variable by using the assignment statement, which has the following general syntax:

```
variable := expression;
```

The variable on the left is assigned the value of the expression on the right. The symbol: = is called the *assignment operator*. If the variable represents a legal numeric variable identifier, an assignment statement might be

```
age := 29;
```

where the value 29 is assigned to the memory location defined by the variable age. It is important to note that the assignment operator here is not the same as the equal sign used in basic algebra. In algebra two values may be equal. Turbo Pascal's assignment operator simply means a variable is assigned to or is given or becomes the value of the expression.

Remember, when assigning variables it is important that the two sides of the assignment statement be compatible with each other. If the left side of the statement is a numeric integer, the right side must be an integer. Likewise, if the left side is a string, the right side must be a string. The reason is that numeric variables are stored in memory differently than are text and string variables. In addition, integers are stored differently from reals.

Only one variable can be on the left side of an assignment statement. Given the following declarations:

```
Var  
  pay           : real;  
  letter        : char;  
  score         : integer;  
  test, result  : integer;
```

the following are valid assignment statements:

```
pay := 6.25;  
letter := 'X';  
score := 8;
```

the following are invalid assignments:

```
letter := 5; {5 is not a character or string data value; '5' is}  
score := 99.9; {score is an integer, 99.9 is a real value}  
test, result := 100; {only one variable allowed on left side of :=}
```

Constants

Having discussed identifiers whose values can change, or variables, I will now examine a value that remains unchanged, or constant. As the name applies, the *constant* has a value that is assigned to it. Constants, like variables, must be identified via a declaration statement. A constant declaration appears in the constant section. The constant section usually appears before the **var** (variable) section and after the program statement, although the constant and variable blocks do not have to appear in any specific order in Turbo Pascal. To declare a constant, begin with the reserved word **const**. For example, the following represents the constant value 30 assigned to an identifier called **MaxStudents**:

```
Const  
  MaxStudents = 30;
```

In our example the constant identifier (**MaxStudents**) is an integer value because the value *implicitly* assigned (30) is an integer. Likewise, we can define a real constant by assigning a real data value:

```
Const  
  Rate = 2.5;  
  Multiple = 6.24;
```

or define a character string or *literal*:

```
Const  
  name = 'Sarah';
```

Notice that the constant declaration uses the equal sign instead of the **:=** operator used with the assignment statement. It is important to note that unlike the variable, a constant identifier that represents a location in memory

cannot be changed or redefined later in the program; it remains constant and equal to the value assigned to it. For instance, if we had the string constant `Name = 'Sarah'`; in a program, we could say:

```
Var
  aName : String[25];

begin
  aName := Name;
end.
```

but not

```
Var
  aName : String[25];

begin
  Name := 'Kelly'; { this is a no-no! }
  aName := Name;
end.
```

It may help to visualize how Turbo Pascal treats constants to understand why the above example will result in a syntax error. When Turbo compiles your program and it finds a declaration for a constant, it replaces all references to that constant with the value assigned to it. So in the valid example, when this assignment is made:

```
aName := Name;
```

the literal `'Sarah'` is substituted for the identifier `Name`. Remember, no modifiable memory location is set up on a constant declaration.

A Word on Punctuation: The Semicolon

As promised, it is time to restate some important points about the use of semicolons. You must be acutely aware that the syntax rules for program construction are very important for proper execution of a Turbo Pascal program. In particular, the omitted semicolon is often the first mistake a beginning programmer makes. In fact, experienced programmers have been known to omit the ominous semicolon.

What is the semicolon used for? To separate program statements in Turbo Pascal. However, not all statements must end with a semicolon. For example, the last **end** in a program is followed by a period. The semicolon separates statements on separate lines as well as statements that appear on the same line. Yes, you can have two separate Pascal statements on the same line. For example:

```
lowScore := 110; highScore := 287;
```

The Turbo compiler knows they are separate statements because of the semicolon at the end of the first one. Sometimes the semicolon is optional. For example, the last statement before an **end** doesn't require a semicolon and is generally not used there. Also, there are special circumstances when it seems a semicolon should be used, but actually it's wrong to do so. When such a situation arises, I will point it out. For now, follow the general rule that semicolons are used to separate statements. Proper use of the semicolon is a fundamental concept that you will master with practice.

Expressing Yourself

I have introduced the concept of the expression during the discussion of the assignment statement. An expression is a sequence made up of variables, mathematical operators, and constants. The following are valid expressions:

```
6 + 2
a + b - 3
pay - 3.25
sum + 1
```

Use the assignment operator to assign the value of an expression to a memory location defined by an identifier. The operators used in an expression must be valid for the type of data they operate on. The most commonly used expressions are arithmetic. Turbo Pascal is capable of performing most of the mathematical operations that you are familiar with

+	addition
-	subtraction
*	multiplication
/	division (non-integer)
DIV	integer division
MOD	modulus or modulo division (remainder from integer division)

You should recognize $+$, $-$, $*$, and $/$; however, the operators for DIV (integer division) and MOD (modulo division) may be new to you. When dividing two integers, the result is also an integer. Use the integer division operator thus:

```
6 DIV 3 {the result is the integer 2}
8 DIV 2 {the result is the integer 4}
9 DIV 4 {the result is the integer 2}
```

The first two examples should be easily understood; in the third the result is 2 because in integer division the remainder is discarded. However, you can determine the discarded remainder via the modulo operator.

The modulo operator is used to calculate the remainder of integer division. For example:

```
6 MOD 5 returns a value of 1
8 MOD 4 returns a value of 0
9 MOD 6 returns a value of 3
9 MOD 2 returns a value of 1
```

DIV and MOD have no meaning on real values so may not be used with them. When dividing real values, use the $/$. Integers and reals may be used in the same expression, but the integer value is converted to a real, and the result is a real value. In other words, integers may be assigned to real variables, but real variables may not be assigned to an integer variable.

Mathematical Order of Operation

Mathematics is performed in a particular order of operation. For example, multiplication and division are performed before addition and subtraction. To illustrate, what do you think the result of this operation will be:

$$3 + 5 * 6$$

If you read this example from left to right, you may interpret the result as 48 instead of 33, but the multiplication ($5 * 6$) should be done before the addition ($3 + 5$). This is a result of the order of precedence of the operators. Operators with high precedence are performed before operators with low precedence. The high-precedence operators are $*$, $/$, MOD, and DIV. Operators with low precedence are $+$ and $-$.

If operators are of the same precedence, the calculation may be performed

from left to right. In the example below the left-hand operation (addition) may be calculated first.

$$6 + 9 - 2$$

As you probably determined, the result of this expression is 13.

Parentheses are used to dictate the order of operations in the same way they are used in basic mathematics. The operations contained within parentheses have a higher order of precedence than multiplication and division. For example, consider this statement:

```
Total := (4 + 8) DIV 2;
```

The result is that the variable Total is assigned a value of 6. The operation inside the parentheses is performed before the division (DIV) operation.

Operations within parentheses must follow the normal order of precedence. Can you determine the value of this statement?

```
rate := (6 * 2 + 18) - (8 - 16 DIV 4);
```

The value is 26.

In summary, arithmetic expressions consist of a sequence of variables, operators and/or constants. To eliminate any ambiguity in an arithmetic expression, a mathematical order of precedence of operations must be followed.

Simple Turbo Pascal Arithmetic

I have covered several fundamental concepts in this chapter, and so it is appropriate to review them in a couple of simple examples. Consider calculating your gas mileage, a simple problem that Turbo Pascal on your Macintosh can easily solve. How many miles do you travel to work or school? How many gallons of gas do you use? Mileage is the number of miles traveled divided by the number of gallons of gas used. If you answered these questions—number of miles and number of gallons—you have defined the programming problem. If you determine that you traveled 200 miles last week on 8 gallons of gas, you can write a program that looks like this:

```
(*****)  
(* This program calculates your gas mileage. *)  
(*****)  
Program GasMileage;
```

```
Var  
  miles, gas, mileage : real;  
  
begin  
  miles := 200.0;  
  gas := 8.0;  
  mileage := miles / gas;  
  writeln (mileage);  
  readln  
end.
```

When you run this program, the result is displayed in the upper left-hand corner of the display as shown in Figure 3.4.

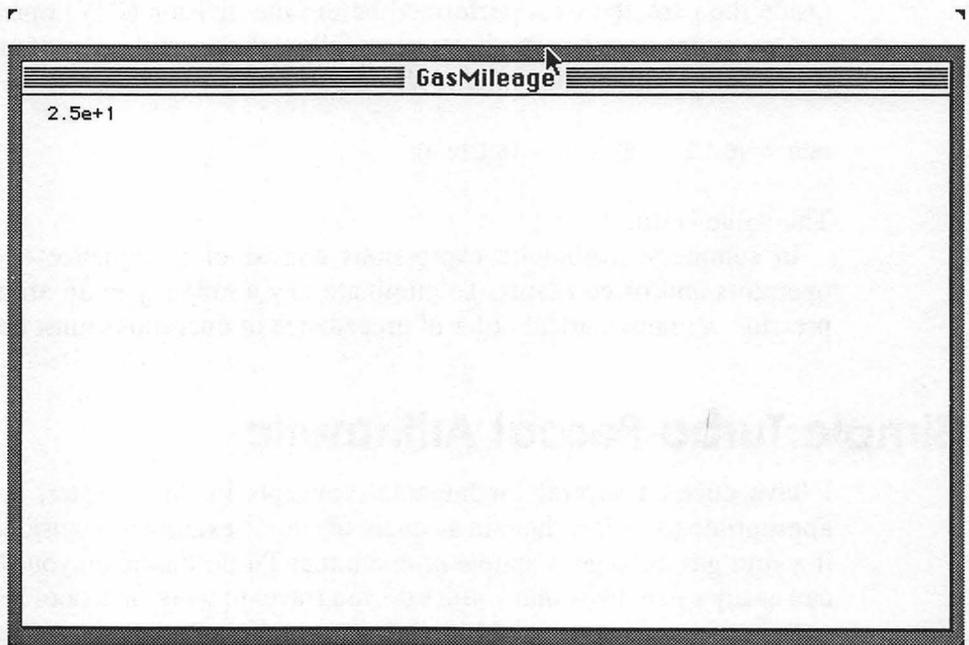


Fig. 3.4.

Review this program step by step. I won't explain every program in such excruciating detail, but it is important that you master these beginning concepts early.

The first few lines are a comment block that briefly describes the program's purpose. The next line names the program `GasMileage`. Don't forget to place the semicolon at the end of the line.

The next line begins the declaration section for variables. The reserved word `var` specifies this section. The variables `miles`, `gas`, and `mileage` are

declared as real variables. Because they are all of the same data type, we can declare them with a single statement separated by commas.

The next section is the program block, identified by the **begin-end** pair. The program statements within the program block are the executable portion of the program. The first three lines following the reserved word **begin** are all assignment statements. Each line is concluded with a semicolon to separate the individual program statements. The last executable program is the `readln` statement. This statement causes nothing to happen until you press the return key. This allows you to read the result displayed on the screen. Note the punctuation (mileage), not 'mileage', in the `writeln` statement. If you use quotes, the string or literal value "mileage" itself is printed on the screen instead of the real calculation assigned to it. Remember, the last statement in the program block (before **end**) does not require a semicolon.

The last line, of course, signals the end of the program. All program blocks must have a matching **begin-end** pair. Since this is the last program block, the **end** must be followed by a period.

Are you beginning to understand the importance of punctuation? Omitting it will cause you many headaches. Did you notice that the placement of the semicolon followed a few general rules but that you didn't have to put one at the end of each line? For example, the semicolon is used to separate program statements, but you don't place them at the end of a few reserved words, such as **var** and **begin**. Don't worry if you're making a few initial mistakes. You will master the placement of the semicolon as we progress.

Our next example further illustrates the use of mathematical variables and assignment statements. Study the following program and determine if you can understand the components. If you can't, you should review the appropriate section. I have liberally used comment statements to facilitate understanding.

(* This program calculates a student's grade point average *)

Program StudentTestScore;

```
Const { starts the declaration section for constants }  
  AName = 'Joe'; { declare AName as literal "Joe" }
```

```
Var { starts the declaration section for variables }  
  test1, test2, test3, test4: real; { test score variables }  
  testAverage : real;           { average test score }  
  name : String[25];           { name of a student }
```

```
begin { starts the program block }  
  test1 := 85.0; { the following are assignments }
```

```
test2 := 90.0;
test3 := 78.0;
test4 := 79;
name := AName;
testAverage := (test1 + test2 + test3 + test4) / 4.0;
writeln ('What is the student's name and test average?');
writeln (name); { output the information }
writeln (TestAverage);
readln; { wait for user to press <RETURN> }
end.
```

When you execute this program, your screen should look like the one in Figure 3.5.

More on Data Types

Now that you have examined a couple of simple programs, I want to discuss a few more options for data types. The most common data types are real, integer, char, string, and Boolean. In this section we discuss how you can define your own data types.

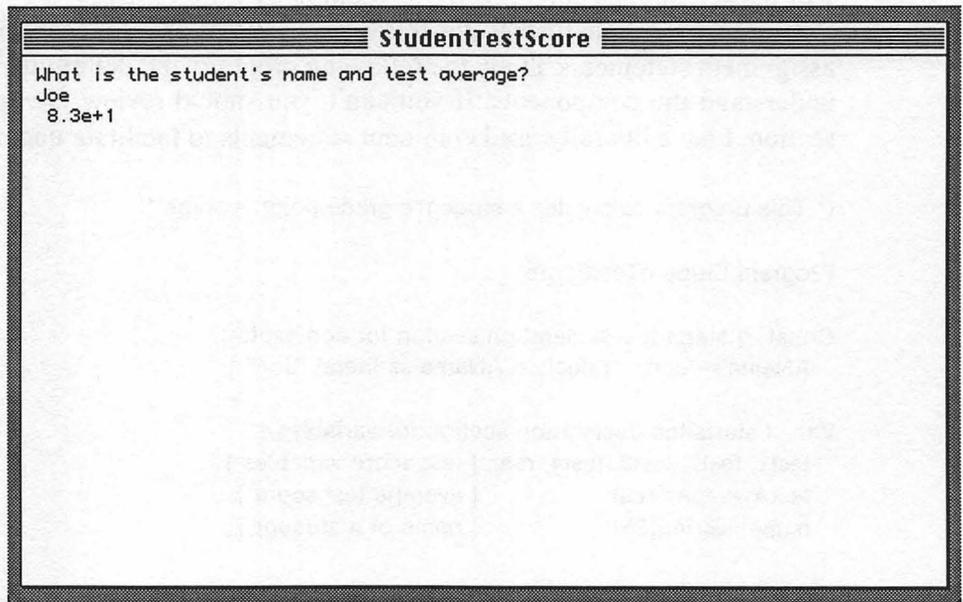


Fig. 3.5.

User-Defined Data Types

Turbo Pascal lets the programmer create new names and data types with the reserved word **type**. The **type** declaration tells the computer that the identifier is a user-defined data type. The most simple data type is called *ordinal* or *enumerated* because it is defined by listing the possible values that represent the data. Other data types, such as records and arrays, define *structures* and will be discussed later in this book. If the data type is represented by more than one value, the values are separated by a comma and enclosed in parentheses. For example:

Type

```
Sport = (Football, Basketball, Hockey, Baseball);
```

where the reserved word **type** appears after the program heading and before **var**. The three declaration sections we know of so far are **const**, **type**, and **var**, and they are generally presented in that order, although they need not appear in this sequence in Turbo Pascal. Once a data type is created, you can declare a variable of that type just as you would declare a variable that is an integer, char, or any other predefined type. Thus:

Type

```
Sport = (Football, Basketball, Hockey, Baseball);
```

Var

```
favoritePastime : Sport;
```

where the variable `favoritePastime` is declared as data type `Sport`.

Why are user-defined data types popular? The answer is simple: readability. Programs should be documented and as readable to the human eye as possible. Our example could have simply assigned a meaningless identifier, such as `x`, to represent the list of possible defined values (`Football`, `Basketball`) or perhaps have declared the same values as constants using strings. However, defining the data types provides a little more flexibility when writing a program for other people to read.

Once I have declared the variable `favoritePastime` above, I can make assignments to it like this:

begin

```
    favoritePastime := Baseball;
```

end.

This is nice, but what if I want to display this information via `writeln`? Try this little program:

```
Program SportTypeExample;

Type
  Sport = (Football, Basketball, Hockey, Baseball);

Var
  favoritePastime : Sport;

begin
  favoritePastime := Baseball;
  writeln ('My favorite pastime is', favoritePastime);
end.
```

If you try to run or compile this program, you will get a compile error because Turbo Pascal does not permit this use of a variable. Some Pascal compilers will allow you to display the value of `FavoritePastime` as it is defined in the enumerated type. For instance, on those compilers, the output from the above program would be:

```
My favorite pastime is Baseball
```

Let's take a closer look at what goes on within the Macintosh when you define an enumerated type like `Sport`. First of all, Turbo doesn't know what football, basketball, and so on are, but it represents their values internally with the values 0 through 3. So when you declared the **type** `Sport`, Turbo set up a table so that when any references are made to the **type** `Sport` or any of its defined values, it assigns integer values to the variable instead of the names `Football`, `Basketball`, and so forth. The table is set up like this:

```
Football = 0
Basketball = 1
Hockey = 2
Baseball = 3
```

So when you made the assignment

```
favoritePastime := Baseball;
```

Turbo assigned the value 3 to favoritePastime. In addition, the variable favoritePastime occupies only 1 byte in memory. This is because 1 byte of memory can hold 256 different values (0 through 255). So I could have an enumerated type with up to 256 unique possible values and it still would occupy only 1 byte of memory.

Subranges

Ordinal data types have an important characteristic: they include a distinct set of values or constants that are ordered:

Type

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

In this type Monday follows Sunday, Tuesday follows Monday, and so on. Such a data type, where the set of values is ordered, is called *scalar*.

You may want to use only a portion of your new data types. Turbo Pascal allows you to define a specific portion of any ordinal or scalar type (except real) as a specified subrange. Subranges are specified by placing two periods between the first value and the second value of an enumerated list. For example:

Type

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

```
WeekDay = Monday..Friday;
```

Var

```
aDay : Day;  
a WeekDay : WeekDay;
```

The subrange includes the values Monday, Tuesday, Wednesday, Thursday, and Friday. It is important to note that the first value of a subrange must be less than the second value. Note also that you do not use parentheses when defining subranges; simply specify the beginning and end of the range separated by two periods. Also, don't try to make this assignment

```
aWeekday := Sunday;
```

with the above declarations because you will get a compiler error, since it is outside of the defined subrange.

Program Formatting

Program formatting is a very important concept in Pascal. In general terms program formatting is where on the screen or line you begin a statement, insert a space, indent, and so on. The reason for program formatting is more than a compulsive desire for neatness; it makes your programs more readable, thus reduces the amount of time needed for debugging and for comprehension by someone other than the author.

Indentation is a major factor in program formatting in complex source code. It is recommended that every time an indentation is necessary, you indent a fixed number of spaces, say, 3 or 4. You can also set up Auto Indent from the Options selection in the Edit menu. Take note of the formatting practices I present and keep your eye on them. This way your programs will take on a much more professional appearance and will lend themselves to easy tracing by other programmers.

We have covered a lot of ground, so take a 15-minute break, absorb what you have learned thus far, review the points at the end of this chapter, answer the Quiz questions, and then go on to Chapter 4. If you don't understand all the concepts presented in this chapter, go back and reread the appropriate sections.

Review Summary

1. The primary elements of a Turbo Pascal program include the program heading, the declarations and definition section, and the program body.
2. The rules for writing a program are called syntax.
3. An identifier may include any combination of letters, numbers, and the underscore symbol (`_`). All identifiers must begin with a letter; they may be of any desired length, although only the first 63 characters are significant to the compiler.
4. Constants may be declared using **Const**. This section generally appears after the program declaration.
5. User-defined types may be declared with **type**. This section generally appears after the **Const** declaration section.
6. Variables may be declared using **var**. This section generally appears after the **type** declaration section.
7. To assign a value to a variable, use the assignment operator (`:=`).
8. The two sides of an assignment statement must be compatible. In other words, if the left side is a numeric integer, the right side must be a numeric integer.

9. An expression is a sequence made up of variables, mathematical operators, and constants.
10. Program formatting is where on your screen or line you begin a statement, insert a space, indent, and so on. Turbo Pascal offers several automatic formatting features such as Auto Indent.

Quiz

1. What is an identifier?
2. What are comments and where are they placed?
3. When does a period follow the reserved word **end**?
4. What is the general ordering of the **var**, **const**, and **type** declaration sections?
5. What is the difference between integer and real data types?
6. What is the semicolon used for?
7. What is the order of precedence for mathematical operations?
8. How is the order of precedence altered?

4

Turbo Pascal Statements

The Write and Writeln Statements
Assigning Data Values with Read and Readln
Statements of Choice: The Conditionals
Decisions, Decisions . . . The Thinking Mac
A Couple of Common Errors with If Statements
Nesting Your If Statements
Boolean Operators
Another Case to Consider
Review Summary
Quiz

In this chapter you will learn:

- How to communicate with the user; the Input/Output statements
- What the Write and Writeln routines are and how to use them for both screen output and printer output.
- What the Read and Readln routines are and how to use them.
- How to write programs that make decisions via if and case.

This chapter takes a deeper look into Turbo Pascal. You have learned many of the fundamentals of program construction. Most of the material you are about to read discusses various program statements that enable you to write more complex programs. As you recall, program statements are the executable statements that appear within a program block, or begin-end pair. You will explore fundamental control structures that give the computer the ability to make decisions. In addition, you will study interactive statements between the computer and the user. First, however, you need to examine how information is displayed or output to your screen or printer. Start by taking a closer look at write and writeln.

The Write and Writeln Statements

You have been briefly introduced to writeln (pronounced write line). In this section we cover writeln and its cousin write in more detail. These

statements are fundamental to programming in Turbo Pascal. In fact, these are the statements the Macintosh uses to communicate the output of your programs so that you can readily understand it. The general syntax for the write statement is

```
write ( parameter list );
```

where the parameter list is a list of values or strings, separated by commas, that you want displayed. If the parameter list consists of valid variables, the Mac displays the values assigned to those variables. For example, a complete program might appear as shown in program Sample below. The result of program Sample is output as shown in Figure 4.1.

Program Sample;

Var

a : integer;

b : char;

begin

a := 25;

b := 'C';

write (A, B);

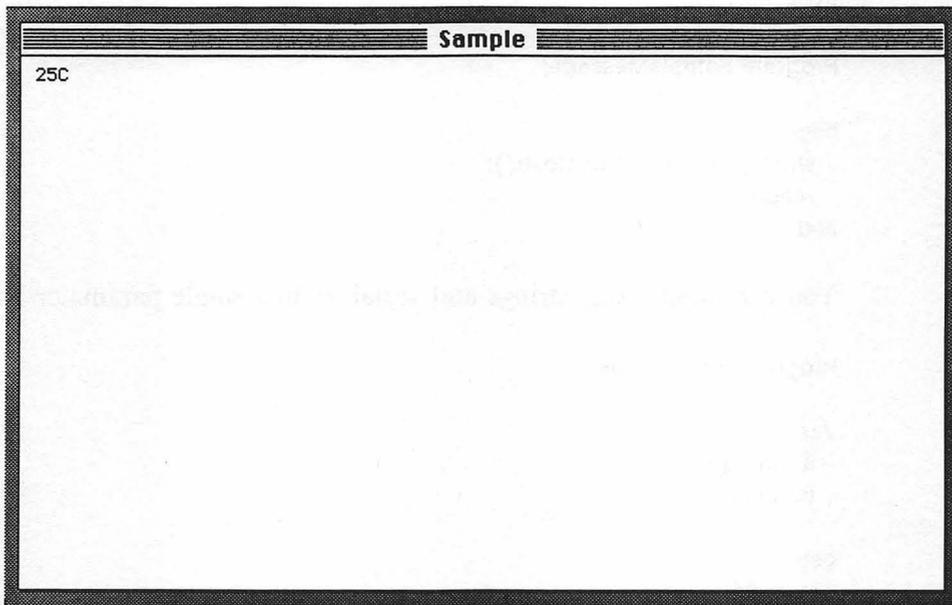


Fig. 4.1.

```
    readln;  
end.
```

Notice that the semicolon follows write if it is not the last statement in the program block (that is, not preceding end); otherwise it is not necessary.

Instead of a list of parameters separated by commas, you may wish to use a list of successive write statements to print each individual value or string. However, the resulting output is the same. For example, the write statement from program sample can be rewritten as two statements:

```
write (A);  
write (B);
```

I find the first method more efficient than the latter. However, the choice is yours. One important point is that whenever a write statement is executed, the values or parameter list will be printed on the same line. This is true even if you use a list of consecutive write statements to print individual values: all the values will be printed on the same line.

Suppose you want to print a character string or message instead of a value assigned to a variable. To do this, simply enclose the character string within a set of single quotes. The single quotes signal the Mac that anything printed within the set of quotes (including spaces) is to be displayed exactly as written. For example, to display the character string I am the Macintosh., write this program:

```
Program SampleMessage;
```

```
begin  
    write ('I am the Macintosh.');
```

```
    readln;
```

```
end.
```

You can interchange strings and variables in a single parameter list:

```
Program Interchange;
```

```
Var
```

```
    a : integer;
```

```
    b : char;
```

```
begin
```

```
    a := 25;
```

```
b := 'C';  
write ('Tom''s age is', A, 'and he rates a', B);  
readln;  
end.
```

When this program is executed, the output is displayed as shown in Figure 4.2.

Using Writeln

Writeln is similar to write except for one important feature: Writeln causes output to be displayed line by line by generating a carriage return after the output statement. In other words, when a writeln executes, it writes out all the parameters associated with it and advances to the next line on the screen for any subsequent output. Take a look at an example:

Program EasyMath;

Var

```
firstValue, secondValue, thirdValue : integer;  
addition, subtraction, multiplication : integer;
```

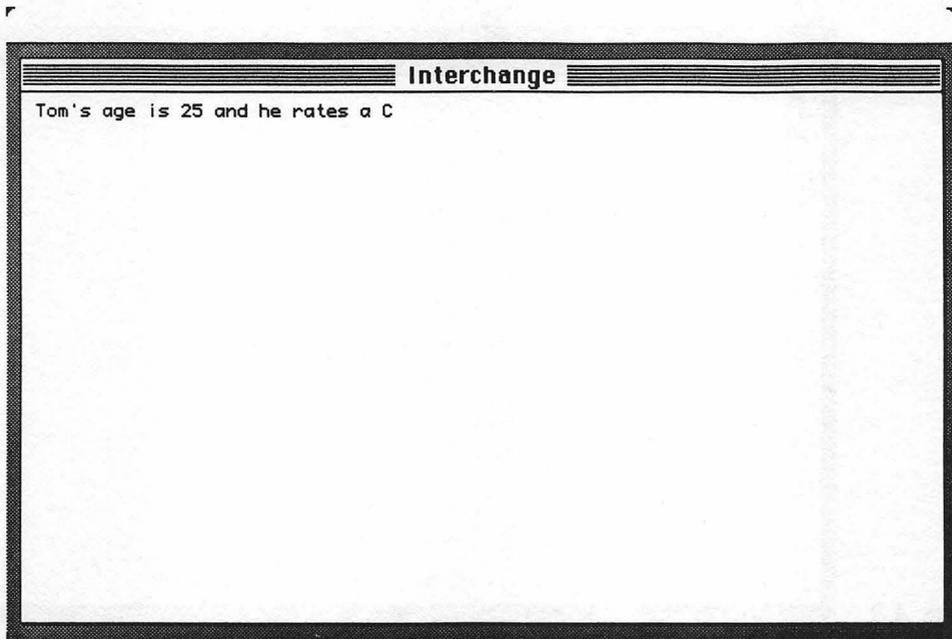


Fig. 4.2.

```
begin
  firstValue := 10;
  secondValue := 5;
  thirdValue := 2;
  addition := firstValue + secondValue;
  subtraction := secondValue - thirdValue;
  multiplication := firstValue * thirdValue;
  writeln (addition);
  writeln (subtraction);
  writeln (multiplication);
  readln;
end.
```

When this program is executed, the result is displayed as shown in Figure 4.3.

If I substitute the `writeln` statements with a single `write` or a series of successive `write` statements, the program displays the results on a single line because no carriage return was invoked. The output displayed in Figure 4.4 shows the results with the `writeln` statements changed to `write` statements.

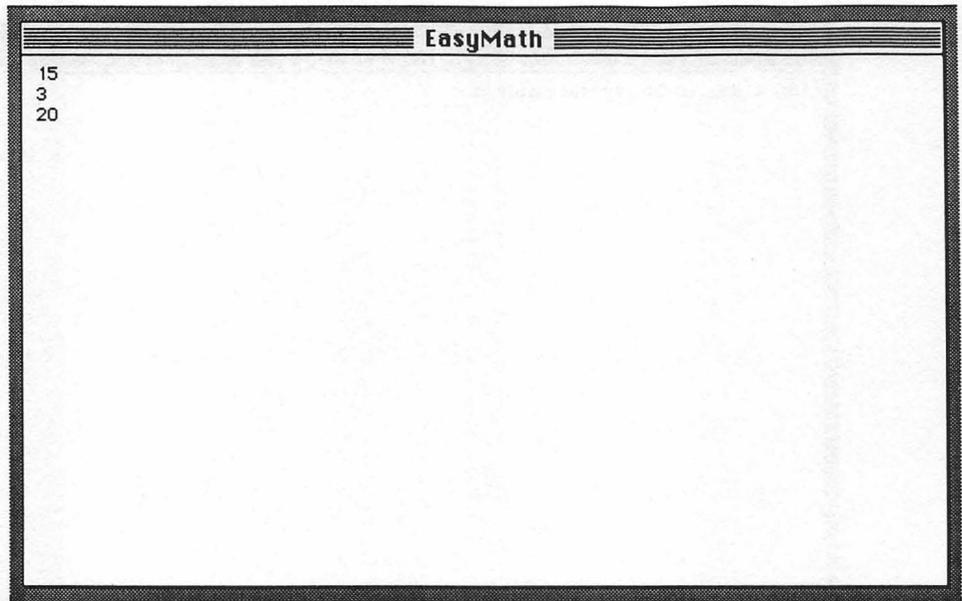


Fig. 4.3.

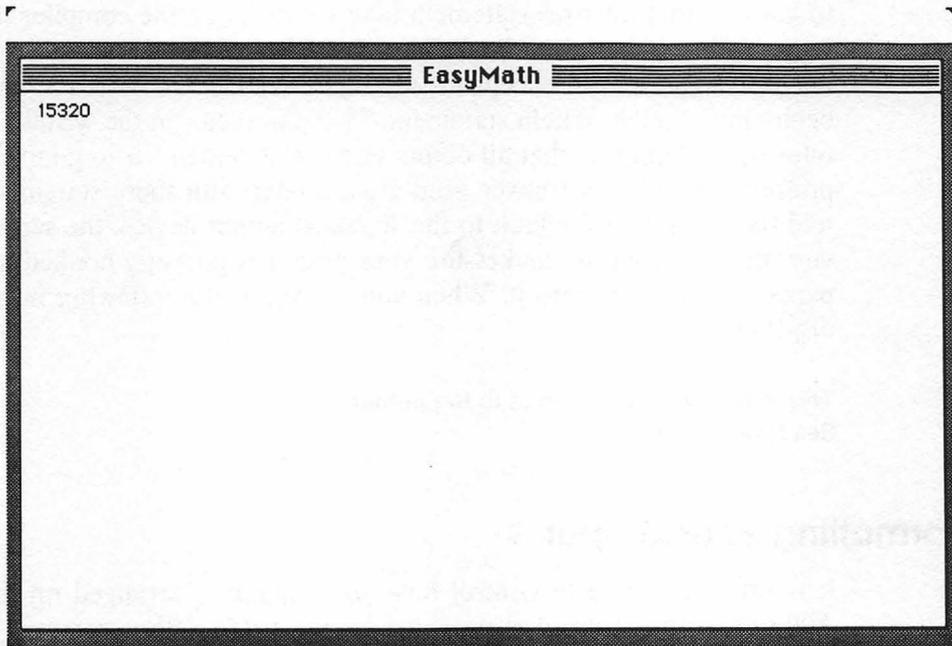


Fig. 4.4.

Sending Output to the Printer

Up to now you have seen how `write` and `writeln` may be used to send messages to your Macintosh's screen; they are also capable of sending output to other devices, like the system printer. Try out this short program:

```
Program SendToPrinter;
```

```
Uses
```

```
  PasPrinter;
```

```
begin
```

```
  writeln(Printer, 'This is how you send output to the printer.');
```

```
  writeln(Printer, 'See how easy it is?');
```

```
end.
```

After the program statement is the first example of the `uses` statement, which tells the compiler to use the information contained in the specified *unit*. The *unit* is a feature of Turbo Pascal used to promote structured programming. The *unit* will be discussed in great detail in Chapter 6. For now, all you need

to know is that the uses statement here is informing the compiler that I wish to use information in the PasPrinter unit. The rest of the program should be fairly straightforward except for the identifier Printer, which appears at the beginning of each writeln statement. This parameter in the writeln statement informs the compiler that all output in those statements is to go to the system printer; the fact that I never used a parameter with these statements before told the compiler to default to the standard output device, the screen. Before you run this program, make sure your printer is properly hooked up and the paper has been fed into it. When you do run it, the following messages are displayed:

This is how you send output to the printer.
See how easy it is?

Formatting Your Output

It is often desirable to control how your output is arranged on the screen. Additional print control is provided by the *field-width parameter*. In simple terms a vertical position on the Mac screen is sometimes called a field. The first printable position in the upper left-hand corner of the output window is the first field, or column. Then exactly what is the field-width parameter? Look at the following short program:

Program ShowFieldWidths;

Var

 x, y : integer;

begin

 x := 1;

 y := 2;

 writeln(x,y);

 writeln(x:10,y:10);

 x := 100;

 y := 200;

 writeln(x,y);

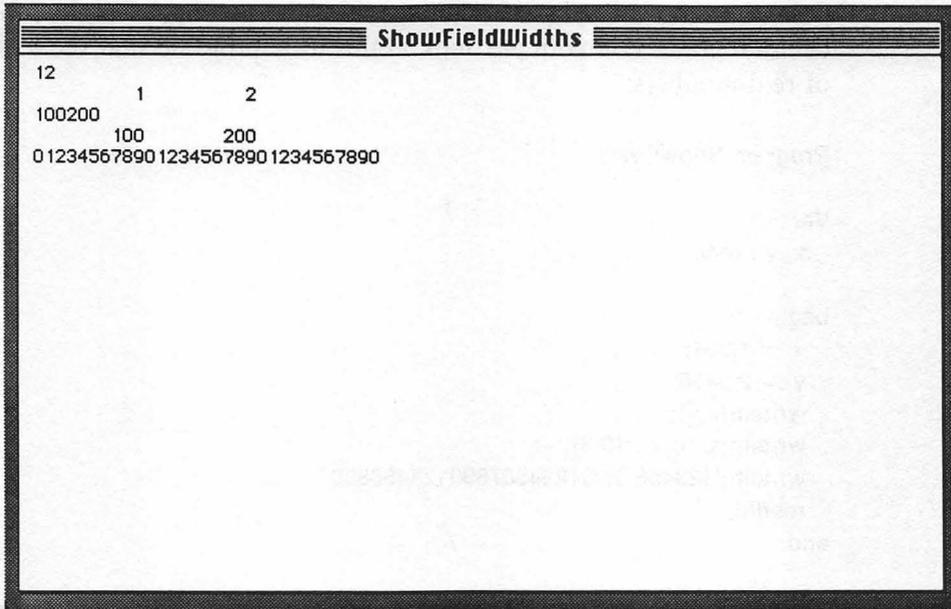
 writeln(x:10,y:10);

 writeln('0123456789012345678901234567890');

 readln;

end.

If you run this program, your screen will look like Figure 4.5.



```
ShowFieldWidths
12
100200
100      200
0123456789012345678901234567890
```

Fig. 4.5.

The first time I wrote `x` and `y`, they were pushed up against each other with no space in between. This is because the `write` and `writeln` statements don't implicitly put any spaces between integers when they are displayed. However, when I specify a field width like this:

```
writeln(x:10, y:10);
```

I am instructing the compiler to display the values held in `x` and `y` in fields a minimum of 10 characters wide. If you look at the last `Writeln` statement, I display a measuring device to show exactly which column the numbers start in; displaying repeated sequences of 1, 2, 3, and so on right-justifies the numbers displayed through the `Writeln (x:10, y:10);` statement in the 10th and 20th columns respectively. This is because `x:10` says to display the value in `x` in a field 10 characters wide and right-justified. Since `y:10` says the same thing, the first 20 characters in that line are occupied with the values of `x` and `y` plus the spaces necessary to pad them out to 10 characters each. The same thing happens after I assign the values 100 and 200 to `x` and `y` respectively. The field width is still 10 characters for each variable displayed, and each value is right-justified within its field.

To display real numbers Turbo Pascal provides a slightly modified notation. Two colons and accompanying field-width parameters are used. The

first parameter specifies the total minimum field width excluding the decimal point, and the second parameter specifies the number of digits following the decimal point. For example, look at this little program that shows formatting of real numbers:

```
Program ShowReals;  
  
Var  
  x, y : real;  
  
begin  
  x := 1.234;  
  y := 2.3456;  
  writeln(x,y);  
  writeln(x:10:3,y:10:3);  
  writeln('12345678901234567890123456890');  
  readln;  
end.
```

When you run this program, your screen looks like the one shown in Figure 4.6.

The first line of output is no surprise; it's the display of the real numbers in scientific notation we have seen before. However, the second Writeln

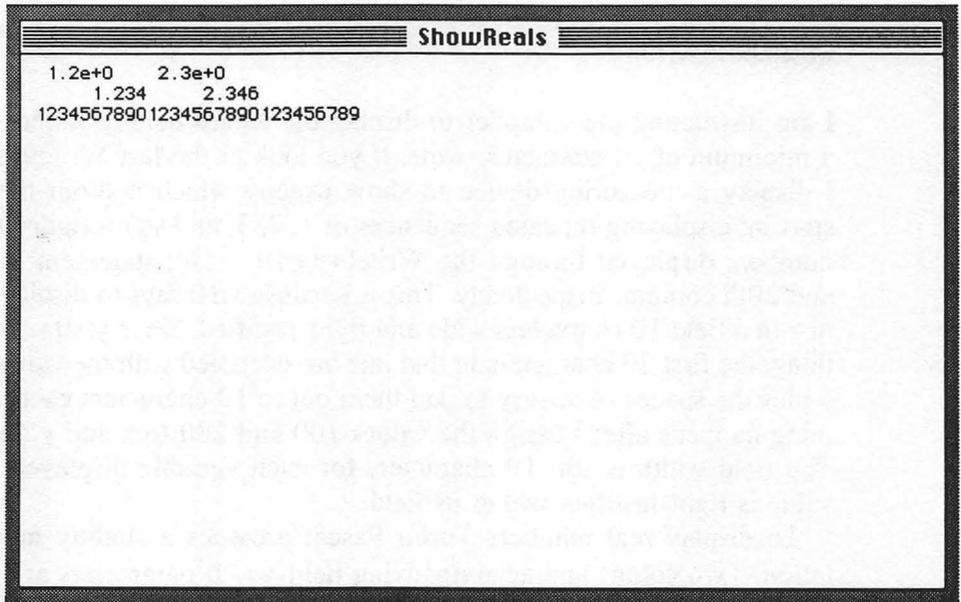


Fig. 4.6.

displays the same numbers in a format whose minimum total field width is 10 characters and that displays 3 digits to the right of the decimal point. Notice how in the first `Writeln` the exact values are not displayed, since in scientific notation only one digit after the decimal point is displayed. Also, because the field width to the right of the decimal point is only three, the value of `y` is truncated and rounded up. That is, the third digit to the right of the decimal point for `y` is rounded up to 6.

Using the same philosophy, you can format string output on your screen. Why bother with field-width parameters? This type of control is particularly useful for tabular material in rows and columns. It's much easier to align headings, labels, and prefixes when you control the display of your character output.

Printing Blank Lines

When you execute a `writeln` statement, a carriage return causes any subsequent displaying of data to start on the next line. If you just type a `writeln` statement without a parameter list, the statement is executed and a blank line appears. This program inserts blank lines:

Program ShowBlankLines;

```
begin
  writeln ('You can insert');
  writeln ('blank lines');
  writeln;      {this inserts a blank line}
  writeln('anywhere in your programs. ');
  readln;
end.
```

When you run the program, your screen looks like Figure 4.7.

Assigning Data Values with Read and Readln

The traditional assignment statement is not the only method at your disposal for assigning data values to variables. Turbo Pascal provides two additional interactive statements that assign input from within the program itself as it executes. The statements `read` and `readln` offer additional flexibility to the programmer because data values can also be assigned to a variable from outside the program. You have already seen examples of the use of the

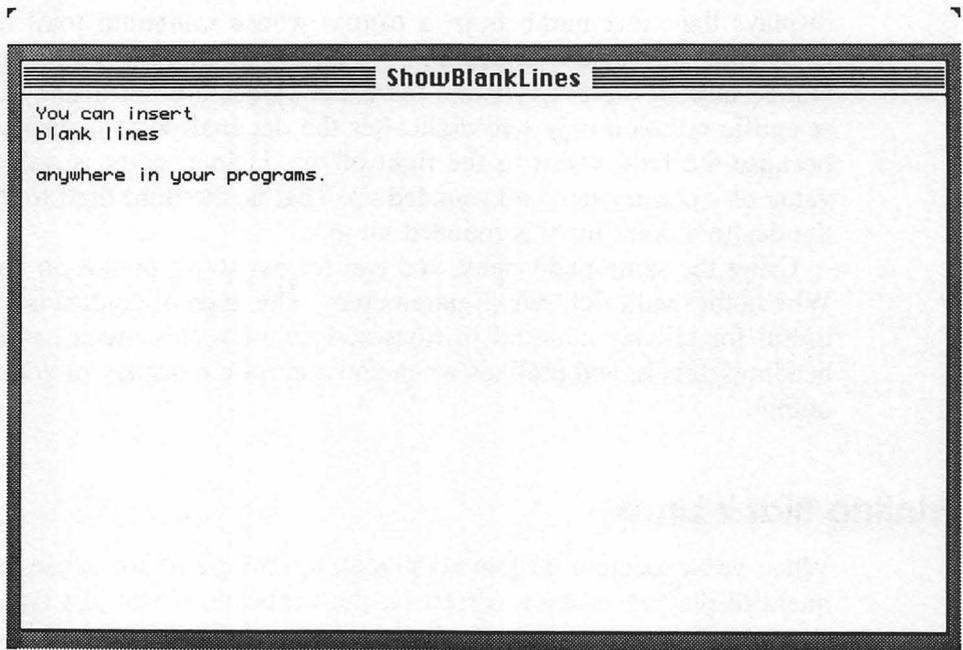


Fig. 4.7.

readln statement; in almost every program up to now I have used it to keep the program's output display on the screen until you hit Return so you have some time to look at each screen. The general syntax of read and readln is similar to that of write and writeln; the statement read or readln is followed by a parameter list of variables separated by commas and contained within parentheses. For example, if a is defined as an integer, I could say:

```
read(a);
```

or

```
readln(a);
```

The read and readln statements above are similar in that they both request the user to provide values for the variables, but readln is the more commonly used statement. For that reason I shall further discuss how to assign data values with readln.

Mac Asks Questions

Read and readln are often called interactive statements; that is, data values are assigned to variables directly from the user or programmer via the keyboard. When a read or readln statement is executed, the program stops and asks the user for information. The information given by the user is the value

assigned to the variables within the parameter block of the statement. Because read and readln act like an assignment statement, the same syntax rules for data-type compatibility apply. In other words variables must be first declared and subsequently assigned values of the data type used in the declaration. To illustrate, the computer can simulate a human conversation. A program that asks you for your name and then displays a customized message with your name on the screen can be written as

Program SayHello;

```
Var
  name : String;

begin
  write ('What is your name?');
  readln (name);
  writeln('You have a nice name', name);
  readln;
end.
```

Examine this program step by step. Line 1, of course, is the program statement; it defines the program as SayHello. Line 2 begins the declaration section with the reserved word var. Line 3 declares the variable name as a string. Simple enough so far.

In the next section the familiar begin-end pair signals the start of the meat of this little program. In this program block the first write statement asks the user for information, in particular, the user's name. Next the readln statement is executed. This line stops the program and waits for you to give an appropriate response. Type your name and press the return key. The return key must be pressed here to signal to the Mac that your entry is complete. The next writeln statement displays the message enclosed within quotes, followed by the value assigned to the variable name via the readln statement. Finally, we see our old friend the readln statement with no parameters used simply to let you look at the output screen before returning to the Turbo environment. A sample run of the SayHello program might look like this:

The Computer Displays:
What is your name?
You have a nice name Janet Craig

You Type
Janet Craig <RETURN>

In SayHello the variable name is assigned a value just as if with the following assignment statement:

```
name := 'Janet Craig';
```

The advantage to using a `read` or `readln` statement is that if you wish to change the value of the variable, you simply give a different response each time the program is executed. Otherwise the traditional assignment statement (using the `:=` operator) needs to be changed before the program is executed.

Please note that a valid response to a `read` or a `readln` is required. A valid response is one that matches the data type previously declared. If the data entered into the keyboard is incompatible with the variable type declared, the Macintosh will blow up with the familiar system error with the lit bomb. If you are running the program from the Turbo environment, not from the desktop, the `RESUME` option will be available and you can get back into the editor by selecting it. Once back into Turbo you will be greeted with an error message which says that an I/O check failed. Once I get into more sophisticated programs, I will show you how to avoid this sort of problem, but for now try not to mismatch your `readln` entries.

Here's an enhancement to my earlier program in which I not only request the user's name but also his or her age:

```
Program LetsTalk;
```

```
Var
```

```
  name : String;  
  age   : integer;
```

```
begin
```

```
  writeln('What is your name?');  
  readln(name);  
  writeln('How old are you', name, '?');  
  readln(age);  
  writeln('Gee, you're', age, 'years old', name, '!');  
  readln;
```

```
end.
```

A sample run of this program is shown in Figure 4.8.

As you can see, I ask first for the user's name and then, using the previously entered name, for the user's age. Finally I display a message of surprise at his or her age.

Reading Several Variables

You can assign a list of variables using a single `readln` statement or by listing the variables in successive `readln` statements. The result is the same. For example, if `a`, `b`, and `c` are declared as integers, the following `readlns` could be performed to input their values:

```
readln(a,b,c);
```

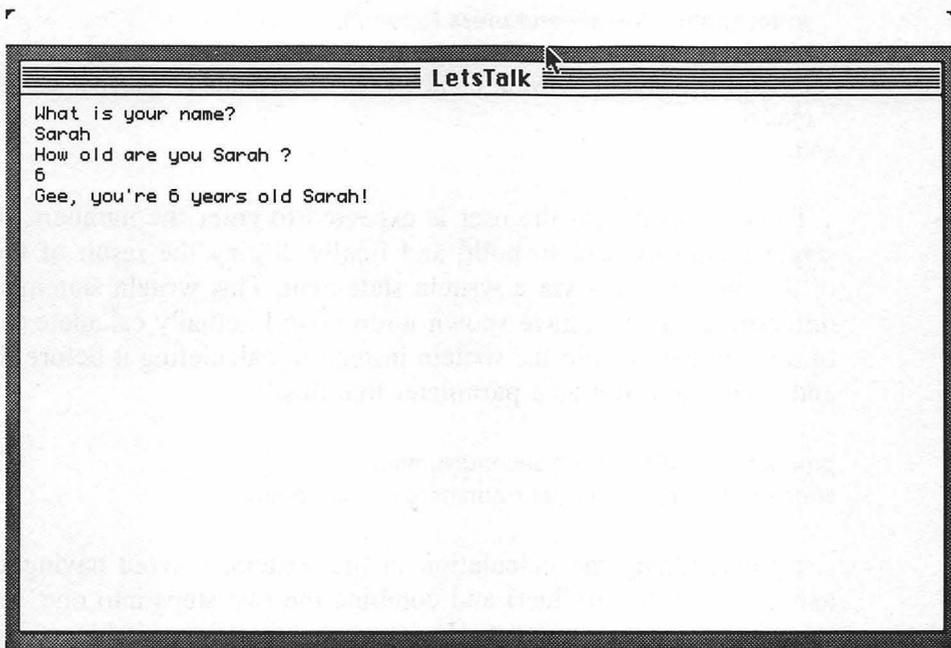


Fig. 4.8.

or

```
readln(a);  
readln(b);  
readln(c);
```

In the first example, `readln (a, b, c);`, you need to press the return key after each entry just as for the second example, whose entries are split among three separate `readln`s. In this particular situation, because `a`, `b`, and `c` are integers, you can enter them on one line and separate them with spaces and just press Return at the end of the line for the first example. However, we suggest that you press Return after each entry instead due to possible portability problems with other compilers. Here's an example of how you might implement a multiple-item `readln` statement:

Program Multiply;

Var

```
    firstNumber, secondNumber : integer;
```

begin

```
    writeln('Please enter one integer, press Return, and enter');
```

```
write('another integer and press Return:');  
readln(firstNumber, secondNumber);  
writeln('The product of your numbers is: ', firstNumber * secondNumber);  
readln;  
end.
```

I first explain how the user is expected to enter the numbers, then use a single `readln` to read in both, and finally display the result of the product of the two numbers via a `writeln` statement. This `writeln` statement is a bit different from any I have shown up to now; I actually calculate the product of the numbers within the `writeln` instead of calculating it before the `writeln` and using the result as a parameter like this:

```
product := firstNumber * secondNumber;  
writeln('The product of your numbers is: ', product);
```

By embedding the calculation in the `writeln`, I avoid having to declare another variable (`product`) and combine the two steps into one, making the Pascal code more compact. However, the resulting machine code may be equally large with either coding option because the compiler is still instructed to create machine code that does the same thing: multiply two numbers and show the result. A sample execution of this program is shown in Figure 4.9.

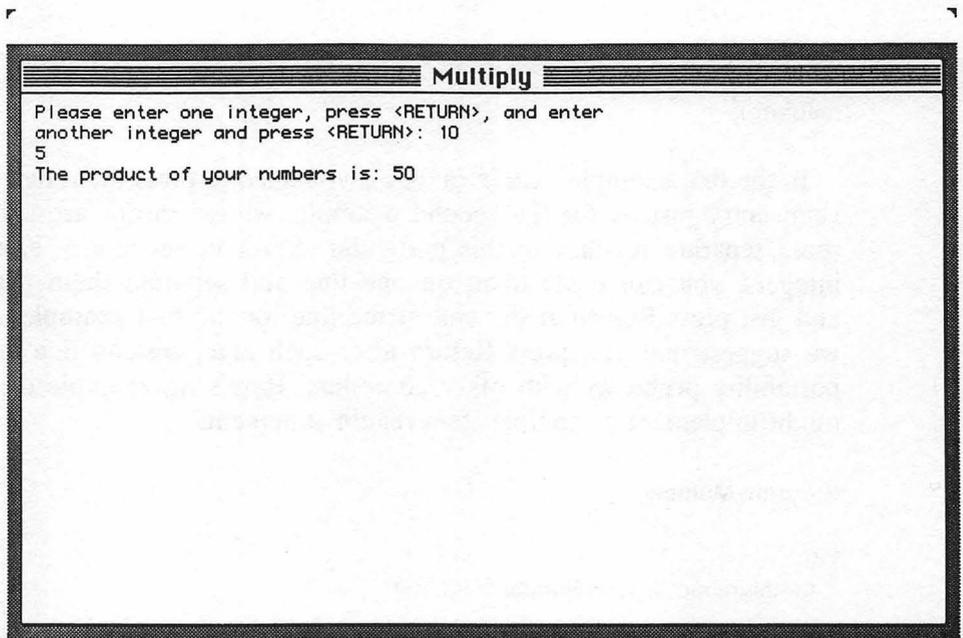


Fig. 4.9.

Explicitly Ask for Information

When you use `readln` interactively, you are prompting or asking for information from the user. As a good programming practice, it is helpful if you explicitly ask for the type of information desired. In my previous examples I explicitly ask the user to enter specific information (name and age). If you are not specific, you may wind up with a system error. This is a potential problem for now, but I will show how to solve it in Chapter 7.

Now that you understand the fundamentals for interactive use of `readln`, look at how the Macintosh can make decisions with the conditional statements `if` and `case`.

Statements of Choice: The Conditionals

A fundamental feature of any computer is its ability to make decisions based on certain conditions or tests. If a particular condition is met, the flow of control within a program is affected and a set of instructions is performed. Controlling the flow of a program is essential to writing powerful programs. The programs depicted in this book thus far simply execute each statement sequentially. Such step-by-step execution limits the ability to develop complex and useful programs. In the next section I will show how to change the flow of sequential instructions by looking at some fundamental control statements: the `if...then...else` compound statement and the `case` statement.

Decisions, Decisions...The Thinking Mac

For practical applications the computer must have the ability to make decisions based on a conditional, or `if` statement. For example, you ask a question; the response can be only true or false, and it will affect the output of the program. Such a response is an assertion to the circumstance of the question. The statement in Pascal that creates the conditional is the `if` statement.

The `if` statement makes a decision by performing a decision test. The test is a true or false (Boolean), yes or no, or 1 or 0 result based on the condition expression. We say 1 or 0 because the only way any computer can think is in terms of 1s and 0s as bits. The syntax of the `if` statement is written as a compound statement with the associated statement and the optional `else` statement. The structure of the simple `if` statement provides two related choices. The first option:

```
if(boolean expression is true)then
begin
  statement1;
  statement2;
  statement3;
end;
statement4;
```

This first option states that if the expression is true, statement 1 is executed, followed by statement 2, and so forth until the end of the block, when statement 4 is executed. The conditional block is enclosed within the familiar begin-end pair. If the expression is false, the conditional block is bypassed and the next statement to be executed is statement 4.

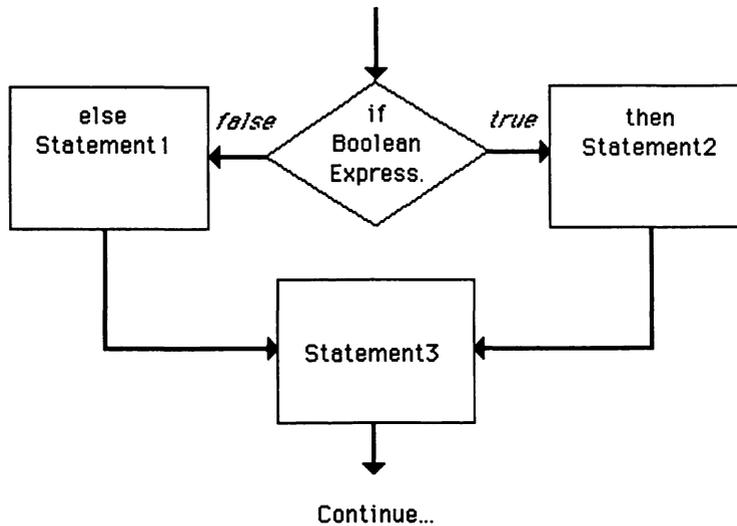
A second structure of the if statement can be written thus:

```
if(boolean expression is true)then
  statement1
else
  statement2;
statement3;
```

The choice above adds a second option, the else portion, to our compound statement. The addition of else provides for the possibility of one more statement being executed if the expression evaluates to false. Thus if the expression is true, statement 1 is executed, statement 2 is skipped over, and statement 3 is executed. If the expression is false, statement 1 is skipped over, statement 2 is executed, and then statement 3 is executed.

Remember, when a conditional expression is executed, a relationship is compared to determine if a condition is true or false. If the condition is true, then the next statement or block of statements is executed (see flowchart in Figure 4.10). The condition itself is specified by relational clauses or operators. These are the relational operators used in Pascal:

=	(equal)
<>	(not equal)
<	(less than)
>	(greater than)
<=	(less than or equal to)
>=	(greater than or equal to)

**Fig. 4.10.**

This simple example illustrates the conditional if statement:

Program DecisionTest;

Var

num1,num2:integer;

begin

write('Enter a number:');

readln(num1);

write('Enter another number:');

readln(num2);

if(num1 = num2)then

 writeln('Both numbers are equal!')

else

 writeln('Both numbers are not equal!');

 writeln('Isn't this fun?!');

 readln;

end.

Study this program. If you obey the computer and enter the numbers 4 and 2, then the second writeln statement is executed and displays:

Both numbers are not equal!

Isn't this fun?!

Your curiosity gets the better of you, however. You decide to test the computer to see if it's awake. You run the program a second time, but now you enter the same number twice. This time your Mac responds with

```
Both numbers are equal!  
Isn't this fun?!
```

Why? Our second test resulted in a condition that was true. Under a true condition the first `writeln` statement is executed. Our first test resulted in a false condition; therefore, the first `writeln` statement is skipped over, and control of the program passes to the next available statement.

A Couple of Common Errors with If Statements

Beginning programmers in Pascal often make a mistake when using the `if` statement. The mistake is a logical one where the output received is an unexpected value. Specifically, I refer to the result of the conditional when the condition is false and more than one statement follows the check.

When a condition is false, program flow skips the first statement and continues with the first statement after the affirmative block. Execution, however, continues from that point and executes the next statement in successive order. This may not provide the results desired. To illustrate, examine a portion of the previous example:

```
if(num1 = num2)then  
  writeln('Both numbers are equal!')  
else  
  writeln('Both numbers are not equal!');  
writeln('Isn't this fun?!');
```

When `num1` equals `num2`, the condition is true and the program flows to the first `writeln` statement and then to the third `writeln` statement. However, suppose you don't want the third `writeln` statement to be executed when the condition is true. At present this situation is not possible because the program flow will continue from the first to the third in a true condition. To change this, include all the alternatives in a single block of statements. The statements within the `begin-end` pair are executed in sequence. Consider this modification to the program:

```
Program NewDecision;
Var
  num1,num2:integer;

begin
  writeln('Enter a number:');
  readln(num1);
  writeln('Enter another number:');
  readln(num2);
  if(num1 = num2)then
    writeln('Both numbers are equal!')
  else
    begin
      writeln('Both numbers are not equal!');
      writeln('Isn't this fun?!');
    end;
  readln;
end.
```

In this situation the message “Isn’t this fun?!” is displayed only when two different numbers are entered. Because it is within the block of the else portion, as designated by the begin-end pair, it will never be executed when two identical numbers are entered.

Another error with if statements is usually encountered by programmers who get carried away with semicolons. You may have noticed that no statement preceding the else clause ends with a semicolon. For example:

```
if(num1 = num2)then
  writeln('They are equal.')
else
  writeln('They are not equal.);
```

The first writeln does not end with a semicolon. If you place a semicolon at the end of the first writeln, you will get an error when you try to compile because that semicolon is interpreted as the end of the if statement, and when the compiler looks at the else clause, it has no if with which to associate it. Even if you have a block of statements before the else clause like this—

```
if(num1 = num2)then
begin
  writeln('They are equal. ');
  writeln('The values are', num1, num2);
end
else
  writeln('They are not equal.);
```

the end before the else is not followed by a semicolon. So another rule to add to the guidelines for semicolon usage is never to place a semicolon after a statement immediately followed by an else clause.

Also, always indent statements within a compound block as shown above so that it is easy to trace which statements go with which block. It may not appear to be very important now, but when you start writing more sophisticated code, you'll be glad you did it.

Nesting Your If Statements

Any statements may follow the true or false clauses. They can be write statements, assignment statements, compound statements containing begin-end blocks, or even another if statement. An if statement within another is called a "compound if statement." Using multiple if statements within the same program structure is called "nesting". There is no steadfast restriction to the number of if statements that may nest within one another; this generally changes from compiler to compiler. However, such a complicated structure can become very difficult to follow logically. A sample of nested if statements:

```
begin
  if(condition1)then
    if(condition2)then
      statement1
    else
      statement2
  else
    statement3;
end;
```

Notice the successive indentation of each if statement. Again, this formatting makes programs much easier to understand.

Why use a nested if statement? Suppose you want a program to make a decision and then to perform one of two mutually exclusive actions. In this case, an if statement will satisfy your needs. Suppose, however, that you want the computer to provide three alternative actions instead of just two. To remedy this problem, we embed or nest an additional decision test. Study the following example of calculating an employee's pay:

Program Payroll;

```
Const
  Rate = 4.00;
```

```
Var
  hrs:integer
  pay:real;

begin
  write('How many hours worked?');
  readln(hrs);
  if(hrs > 40)then
    if(hrs > 45)then
      begin
        pay := (Rate*40.0) + (2.0*(Rate*(hrs - 40)));
        writeln ('You earned double OT! Your pay is $', pay:6:2);
      end
    else
      begin
        pay := (Rate*40.0) + (1.5*(Rate*(hrs - 40)));
        writeln('You earned OT! Your pay is $', pay:6:2);
      end
    else
      begin
        pay := Rate*hrs;
        writeln ('Your pay this period is $', pay:6:2);
      end;
    readln;
  end.
```

When the program is executed, the user is requested to enter the number of hours worked. If a value of 40 hours or less is entered, a straight-time pay rate (\$4.00 per hour) is provided. If a value greater than 40 hours but not exceeding 45 is entered, the employee earns overtime pay. A value of greater than 45 hours results in double overtime pay. For example, if you enter 40, the screen displays

Your pay this period is \$ 160.00

If you respond by entering 45 instead, the output is

You earned OT! Your pay is \$ 190.00

Lastly, if you enter 50 instead, then the output is

You earned double OT! Your pay is \$ 240.00

In our sample program the nested if statement provides three courses of action. Only one statement or statement block may follow the affirmative, or true, clause and appear before each else.

Boolean Operators

We have not discussed Boolean operators. A conditional, logical, or Boolean expression is offered a greater amount of flexibility with the Boolean operators OR, AND, and NOT. These operators let you write compound expressions. For example, suppose you input a value that you want tested for whether it is between 1 and 10. Write

```
if(num > 1) AND (num < 10)then
```

where the expression is true only if both assertions are true. Notice that both assertions are enclosed in parentheses and separated by AND. The individual assertions should be enclosed within parentheses to assure your intent is understood.

The remaining two operators function in a similar fashion. The logic operator OR states that if one or both of the individual assertions are true, the entire expression is true. The logical NOT provides the opposite of a truth value and is called the logical negation; that is, not-true is false. The result of the three logical operators can be summarized in a truth table as follows:

<u>A AND B</u>		<u>Result</u>	<u>A OR B</u>		<u>Result</u>	<u>NOT A</u>	<u>Result</u>
T	T	T	T	T	T	T	F
T	F	F	T	F	T	F	T
F	T	F	F	T	T		
F	F	F	F	F	F		

Another Case to Consider

You can write a segment of code that checks a variable for a specific value and based upon that value executes certain statements. We have seen this in if statements like this:

```
if(num = 5)then
    writeln('The number is five')
else
    writeln('The number is not five');
```

With two alternatives, this statement is acceptable. However, what if I had to display a unique message if the number was any integer from one to five? The if statement would look like this:

```
if(num = 1)then
  writeln('The number is one')
else if(num = 2)then
  writeln('The number is two')
else if(num = 3)then
  writeln('The number is three')
else if(num = 4)then
  writeln('The number is four')
else if(num = 5)then
  writeln('The number is five')
else
  writeln('The number is not within one to five');
```

As you can see, it is allowable to have progressive checks in an if statement via else if. This form of if statement checks each condition, and once it finds a true one, it executes the statement block associated with that condition and skips over the rest of the if statement.

This form of if statement is indeed quite useful, but Pascal provides another, the case statement, that is much more readable than multiple else...ifs. The same lengthy if statement looks like this as a case statement:

```
case(num)of
  1:writeln('The number is one');
  2:writeln('The number is two');
  3:writeln('The number is three');
  4:writeln('The number is four');
  5:writeln('The number is five');
  otherwise
    writeln('The number is not within one to five');
end;
```

As you might have guessed, the expression (num) is evaluated and its value is compared against the figures before each colon. When a match is found, that statement block is executed. After execution of the appropriate block case jumps over the remaining blocks. Finally, if the expression does not equal any of the case values, the otherwise block is executed.

You have just completed another large portion of Pascal programming basics. In Chapter 5 we cover the looping statements, which provide additional control over the flow of your programs. It's time now to review what

you have learned so far. Again, if you have trouble with a particular section, review it before continuing on to Chapter 5.

Review Summary

1. The statements used to communicate the output processed by your computer are `write` and `writeln`.
2. To display a literal character or literal string of characters via `write` or `writeln`, enclose them within single quotes. To display a single quote itself, enter it twice.
3. You can control how your output is arranged by using a field-width parameter. A field-width parameter indicates the total number of spaces allowed when displaying numeric values.
4. The interactive statements that are used to assign data values to variables from outside the program are `read` and `readln`.
5. When prompting for information with `readln`, first explicitly state what type of information is desired by using a `write` or `writeln` statement.
6. The compound `if...then...else` statement is a conditional statement used to make a decision based on a Boolean expression. A Boolean expression represents either a true or false condition.
7. A nested `if` statement lies within another `if` statement. Nested `if` statements let the programmer set up more sophisticated logical checks.
8. A Boolean expression can take on only a value of true or false. A Boolean expression may consist of a single Boolean variable or expressions using relational and/or Boolean operators. The Boolean operators are `AND`, `OR`, and `NOT`.
9. The case statement is another decision control structure that provides multiple alternatives to a decision test. The case statement is a good choice when multiple possible values produce different logic paths for each result.

Quiz

1. What is the difference between `write` and `writeln`?
2. Assuming the real value 65535.342 is assigned to the variable `rate`, what will the following display: `write(rate:7:2)`?
3. How can you insert blank lines in your program output?
4. What ability does `readln` provide the programmer?
5. Write a Boolean expression stating that the value of `A` is both less than 100 and greater than 0. Write the same expression where the value of `A` can be either 100 or 0.

More Statements: The Looping Structures

Programs That Repeat
The While Statement
Counting Your Loops
Loops That Sum
The For Statement
Backward Looping with Downto
Summing up For Loops
The Repeat Statement
While versus Repeat...Until
Nested Loops
One More Loop: The Goto Statement
Review Summary
Quiz

In this chapter you will learn:

- How to use the while statement to construct a programming loop.
- What a counting and summing loop is and how to use one.
- How to use the for statement in a looping structure.
- How to use the repeat...until statement to construct a loop.
- What a nested loop is and how to use one.

In this chapter I continue the discussion of Pascal statements. In particular I examine a few additional control structures that allow you to write programs that repeat or loop. Looping programs allow you to alter the flow of a program or its physical order of execution sequentially from the first program statement to the last. Pascal offers several looping constructs: the while statement, the for statement, and the repeat...until statement. Begin by looking at one of my favorites, the while statement.

Programs That Repeat

Thus far I have discussed programs that are executed one line at a time. To execute a program you double-click on its icon on the desktop or start it up via the Run command in the Compile menu of Turbo Pascal. If you wish to execute the program again, you must double-click on it a second time or select Run again. One statement in Pascal allows you to execute repeatedly a program or a block within a program without double-clicking or running it over and over again. The technique for this is to write a program that repeats itself over and over until a certain condition is false. This conditional control is the while statement.

The While Statement

Similar to the if statement, a while statement tests a condition to determine which two courses of action to take. However, unlike the if statement, after execution the while statement loops back to its start to test the condition again. As long as the tested condition is true, the first action is repeated endlessly. A sample syntax of the while loop can be written thus:

```
while(boolean expression)do
    statement1;
statement2;
```

The reserved word `while` is followed by a Boolean expression and “statement1” may represent one statement or a block of statements enclosed within the begin-end pair. The expression is, of course, a true or false condition. If the expression tests true, the program will execute the statement (or statement block) once. After this execution the program loops back to test the expression again. This continues until the expression results in a false value. A false value tells the computer to skip statement 1 (or the statement block) and execute statement 2. At this point, the looping while statement is completed, and the program will execute the next available statement. The program flow of the while statement is shown in Figure 5.1.

A special example of the while loop appears in the following program. We provide this special program to illustrate a point. In this example you are requested to enter a number. Any response other than 13 will make the condition true and endlessly continue to display the statement following the while check. This is an infinite loop. To stop an infinite loop on your Mac, press the interrupt switch on the programmer’s key on the left side of your Mac. This switch will come in handy later on, when you learn how to fix

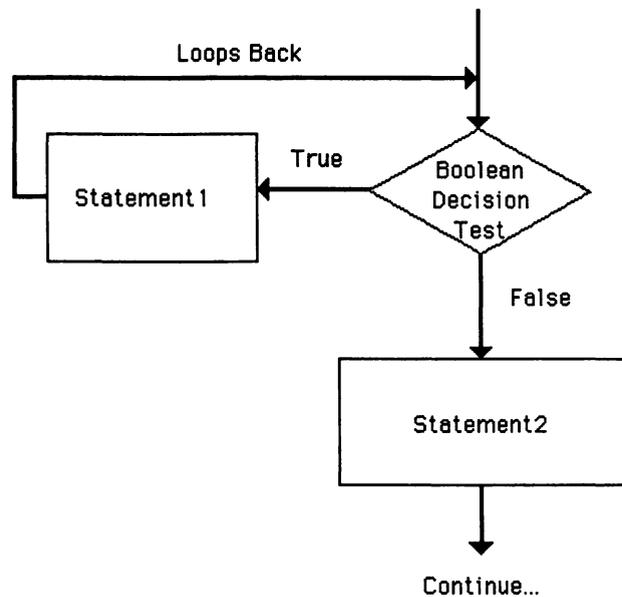


Fig. 5.1.

software problems. If you press this key while running a program with an endless loop in the Turbo Pascal environment (e.g., not from the desktop), you get a dialog box that allows you to resume. Note: If you have a debugger, such as MacsBug, installed, depressing this key will put you into the debugger. Select the resume button (the restart button will reboot your Mac) and you will find yourself back in the Turbo editor. In the following program if the number entered is 13, the loop is avoided.

Program FirstWhile;

Const

MyNum = 13;

Var

num:integer;

begin

write('Please enter a number:');

readln(num);

while(num <> MyNum)do

writeln('You entered the number', num); { infinite loop }

writeln('You entered the number 13!!!');

readln;

end.

Try running the program above. A response of 13 executes the second `writeln` and the loop is avoided. To see the effect of the infinite loop, enter a number other than 13. The screen fills up so fast with the same line it is almost hard to see each new one being displayed.

The example above illustrates poor programming technique. Infinite loops should be avoided. A more practical use of the while loop is a structure whose loop is executed a specific number of times. This type of technique is discussed next.

Counting Your Loops

To control the number of times a loop executes, insert a *counter* to monitor the number of iterations. A counter is a variable assigned a value that rises each time the loop is executed. When the counter reaches some predefined value, the condition becomes false and the flow of control moves out of the program loop. To illustrate, try this program:

Program AnotherWhile;

```
Var
  count,loops_left:integer;
  num1,num2      :integer;

begin
  count:= 0;
  while(count < 5)do
  begin
    write('Enter a number:');
    readln(num1);
    write('Enter another number:');
    readln(num2);
    writeln('The product of this set is',(num 1*num2));
    count:= count + 1;
    loops_left:= 5 - count;
    writeln('This loop has executed', count, 'time(s).');
    writeln('This program will loop ', loops_left, 'more time(s).');
  end;
  writeln; writeln;
  writeln('THIS PROGRAM HAS ENDED!!!');
  readln;
end.
```

This program illustrates two important points. First, notice the construction of the while statement. The first course of action following the while line is a compound statement (begin-end pair). As we have said before, the compound statement may contain any number of statements and still be considered a single block by the construct of the while loop. In other words, referring to the syntax of the while loop, the compound statement represents statement1.

The second point of this example is the construction of the loop counter. I have appropriately called mine "count." The counter is set to 0 outside the loop. Within the loop the counter is increased by 1 each time the while loop executes. When the condition of the while statement is false, program flow exits from the loop and informs the user that the program has ended. The loop itself executes exactly five times. How do we know this? The number of times the loop executes is determined by the loop control variable, in this case the variable count. Compare the value assigned to the variable when it was initialized (count := 0;) with the value of the condition test when the test is false (count >= 5). I can set the count equal to 1 and the condition test as (count <= 5). Either way the loop executes five times. Be careful when setting the number of controlled loops. If the counter is initially set to 0 and the decision test is set as (count <= 5), the loop will execute six times, not five. A sample execution of the program above appears in Figure 5.2.

```
AnotherWhile
This program will loop 4 more time(s).
Enter a number: 3
Enter another number: 2
The product of this set is 6
This loop has executed 2 time(s).
This program will loop 3 more time(s).
Enter a number: 1
Enter another number: 5
The product of this set is 5
This loop has executed 3 time(s).
This program will loop 2 more time(s).
Enter a number: 5
Enter another number: 2
The product of this set is 10
This loop has executed 4 time(s).
This program will loop 1 more time(s).
Enter a number: 8
Enter another number: 5
The product of this set is 40
This loop has executed 5 time(s).
This program will loop 0 more time(s).

THIS PROGRAM HAS ENDED!!!
```

Fig. 5.2.

Loops That Sum

A program can accumulate a result within a loop. Such a loop is used to sum a list of values and is called an *accumulator*, or *summing loop*. This type of loop looks just like the counter just discussed; in fact, the difference between the two is almost negligible. The point is that a summing loop doesn't need to be dependent upon the value of the counting variable within a while loop. The two loops, a counter and an accumulator, may coexist within the same while statement. For example:

```
Program SumTheValues;
```

```
Var
```

```
    num, count, sum:integer;
```

```
begin
```

```
    writeln('This program will sum a list of numbers.');
```

```
    writeln('You will be prompted to enter 5 numbers and you');
```

```
    writeln('should press <RETURN> after each entry.');
```

```
    count := 0;
```

```
    sum := 0;
```

```
    while(count < 5)do
```

```
    begin
```

```
        write('Enter a number.');
```

```
        readln(num);
```

```
        sum := sum + num;
```

```
        writeln('The sum of your numbers so far is ', sum);
```

```
        count := count + 1;
```

```
        writeln('We have made ', count, ' passes through the loop.');
```

```
        writeln;
```

```
    end;
```

```
    writeln('The program is over and the sum of your numbers is ', sum);
```

```
    readln;
```

```
end.
```

As you can see, the variables `count` and `sum` are used to see how many times the program has been through the loop and the total of the numbers entered so far respectively. A sample run of the program might look like this:

```
This program will sum a list of numbers:
```

```
You will be prompted to enter 5 numbers and you  
should press <RETURN> after each entry.
```

Enter a number: 1
The sum of your numbers so far is 1
We have made 1 passes through the loop.

Enter a number: 2
The sum of your numbers so far is 3
We have made 2 passes through the loop.

Enter a number: 3
The sum of your numbers so far is 6
We have made 3 passes through the loop.

Enter a number: 4
The sum of your numbers so far is 10
We have made 4 passes through the loop.

Enter a number: 5
The sum of your numbers so far is 15
We have made 5 passes through the loop.

The program is over and the sum of your numbers is 15.

Every time you enter a number, the sum so far is reported, as is the number of times you have gone through the loop. After five passes a message reports that the loop is finished and your final total is shown. Now take a look at another method of looping, the for statement.

The For Statement

The for statement, like the while statement, is a looping structure used to execute statements repeatedly. The for statement uses a built-in counter to specify the number of times a statement or compound statement executes. In addition, for allows you to initialize and perform regular updates on other variables. A sample format of the for statement is

```
for x := 1 to 3 do
    statement1;
statement2;
```

where x represents the looping or counter-control variable, sometimes referred to as the *index variable*, followed by the assignment operator (:=). The

loop will repeat itself within a specified range set by the beginning point (1 above) to an ending point (3 above). The beginning and ending points are predefined values that can be represented by an expression. Similar to the while statement, the counter-control variable is checked against the ending value and repeats until the ending value is exceeded. In the above example the loop repeats three times (1 to 3).

What is the value of a loop? I have briefly discussed how a loop provides additional control over the flow of a program. A loop can be a real time-saver too. Consider this program segment:

```
for x:= 1 to 100 do
  writeln(num);
```

When the program executes this loop, the values assigned to x are printed from the selected range of 1 to 100. In other words, the value of x is printed 100 times. An alternative method is to use 100 readln and writeln statements. Which method is more practical? Study a simple example:

Program Accountant;

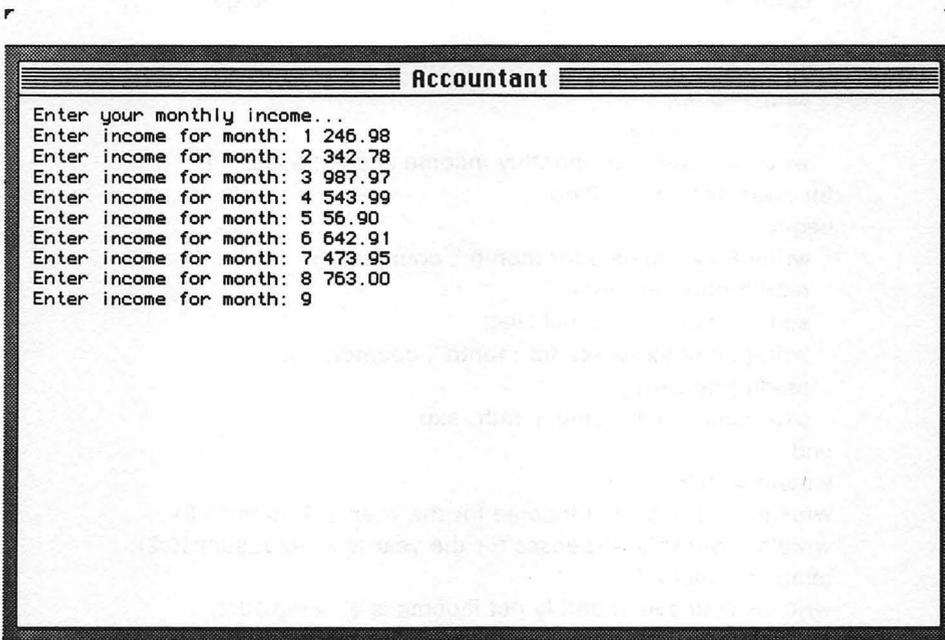
```
Var
  sum, exp_sum, mth_net_inc, mth_exp:real;
  inc_ave, exp_ave           :real;
  month                       :integer;

begin
  sum := 0.0;
  exp_sum = 0.0;
  writeln('Enter your monthly income...');
  for month := 1 to 12 do
  begin
    write('Enter income for month: ', month, ' ');
    readln(mth_net_inc);
    sum := sum + mth_net_inc;
  end;
  writeln;writeln;
  writeln('Enter your monthly expenses...');
  for month := 1 to 12 do
  begin
    write('Enter expenses for month: ', month, ' ');
    readln(mth_exp);
    exp_sum := exp_sum + mth_exp;
  end;
```

```
writeln;writeln;
writeln('Your total net income for the year is $', sum:10:2);
writeln('Your total expenses for the year is $', exp_sum:10:2);
inc_ave := sum/12.0;
exp_ave := exp_sum/12.0;
writeln('Your ave. monthly net income is $', inc_ave:10:2);
writeln('Your ave. monthly expenses is $', exp_ave:10:2);
if(inc_ave < exp_ave)then
  writeln('Your accounts are in the red!')
else
  writeln('Your accounts are in the black!');
readln;
end.
```

When you execute this program, you are requested to enter your net income for the month and then expenses for the same period. A sample run of this program is shown in Figure 5.3.

Examine the for statement itself. Both loops contain an ending value of 12, which specifies that each loop will execute exactly 12 times. The statement following the for statement is a compound set (begin-end pair). All statements contained within this block are executed 12 times. This is how we are able



```
Accountant
Enter your monthly income...
Enter income for month: 1 246.98
Enter income for month: 2 342.78
Enter income for month: 3 987.97
Enter income for month: 4 543.99
Enter income for month: 5 56.90
Enter income for month: 6 642.91
Enter income for month: 7 473.95
Enter income for month: 8 763.00
Enter income for month: 9
```

Fig. 5.3.

to request input via `readln` 12 times. I included an accumulator within each loop to sum the amounts of net income and monthly expenses. When each loop is finished, program flow exits the loop and executes the next available statement. Note that any range that counts 12 times can be used in this example; a range of 12 to 23 will loop 12 times as well. It is the number of increments between the beginning and ending values, not how they are labeled, that is important.

Using a compound statement following each `for` statement is important, because I want the I/O (`writeln` and `readln`) and the accumulator to execute 12 times. If the begin-end pair is omitted, only the first statement following the `for` statement is executed as part of the loop. When the loop is finished, program flow passes to the `readln`, which is executed only once, and everything is a mess.

You should be familiar with the components of the rest of this program. My goal was to simplify a programming task via the `for` statement, and although I have a fairly compact program, since I don't have 12 separate `writeln`s and `readln`s for the I/O, I can compact things even more like this:

Program NewAccountant;

```

Var
    sum, exp_sum, mth_net_inc, mth_exp, temp:real;
    counter                               :integer;

begin
    sum := 0.0;
    exp_sum := 0.0;
    writeln('Enter your monthly income and expenses...');
    for counter := 1 to 12 do
    begin
        write('Enter income for month ', counter, ' ');
        readln(mth_net_inc);
        sum := sum + mth_net_inc;
        write('Enter expenses for month ', counter, ' ');
        readln(mth_exp);
        exp_sum := exp_sum + mth_exp;
    end;
    writeln;writeln;
    writeln('Your total net income for the year is $', sum:10:2);
    writeln('Your total expenses for the year is $', exp_sum:10:2);
    temp := sum/12.0;
    writeln('Your ave. monthly net income is $', temp:10:2);
    temp := exp_sum/12.0;

```

```
writeln('Your ave. monthly expenses is $', temp:10:2);
if((sum/12.0) < (exp_sum/12.0))then
  writeln('Your accounts are in the red!')
else
  writeln('Your accounts are in the black!');
readln;
end.
```

In this version I have removed a couple of variables (`inc_ave` and `exp_ave`) and consolidated them into one (`temp`). More important, however, is the ability to consolidate the income and expense loops into one. Here enter each month's income followed immediately by each month's expenses. In order to cut out variables and code, I added a bit of processing time to the program, since it now has to calculate both average income (`sum / 12.0`) and average expenses (`exp_sum / 12.0`) in both the assignments to `temp` the if statement.

The for statement is a very flexible control structure. A single program block may contain several for statements. In addition, you can specify the number of iterations of a for statement before it executes by predefining the ending value of the increment, say, 1 to 5. We can also write a program that lets the user interactively specify the number of iterations before the loop executes. The following example is another slightly modified version of the accountant program:

Program EvenBetterAccountant;

```
Var
  sum,exp_sum,mth_net_inc,mth_exp,temp1,temp2:real;
  num_mnths,counter                               :integer;

begin
  sum := 0.0;
  exp_sum := 0.0;
  write('How many months do you wish to do:');
  readln(num_mnths);
  writeln('Enter your monthly income and expenses...');
  for counter := 1 to num_mnths do
  begin
    write('Enter income for month', counter, ':');
    readln(mth_net_inc);
    sum := sum + mth_net_inc;
    write('Enter expenses for month', counter, ':');
```

```
    readln(mth_exp);
    exp_sum := exp_sum + mth_exp;
end;
writeln;writeln;
writeln('Your total net income for the period is $', sum:10:2);
writeln('Your total expenses for the period is $', exp_sum:10:2);
temp1 := sum/num_mnths;
writeln('Your ave. monthly net income is $', temp1:10:2);
temp2 := exp_sum/num_mnths;
writeln('Your ave. monthly expenses is $', temp2:10:2);
if(temp1 < temp2) then
    writeln('Your accounts are in the red!')
else
    writeln('Your accounts are in the black!');
readln;
end.
```

The primary difference between this program and the last version of the accountant program is that the user interactively selects the number of times the loop executes. This task is accomplished with the `Readln` statement, which assigns a value to the variable representing the ending value of the range in the `for` statement. Specifically, I used the variable `num_mnths` to represent the ending value. Therefore, from the previous program, the statement

```
for counter := 1 to 12 do
```

becomes

```
for counter := 1 to num_mnths do
```

where the value assigned to the variable `num_mnths` is the ending value of the `for` loop that replaces the value 12. This value must be assigned before the `for` statement executes. In this example the user is prompted to enter the desired value just before the `for` statement. Such a method provides a larger degree of flexibility. You can select 3 months, 6 months, 12 months, or any whole number that you desire. A sample run of this program is shown in Figure 5.4.

Referring to the rest of the program, notice that the formats of the formulas before the `writeln` statements calculating `temp1` and `temp2` that display the average value have been changed to incorporate the `num_mnths` variable. If I simply left these denominators at 12, the resulting values would be correct only when the user entered in 12 for the number of months to process.

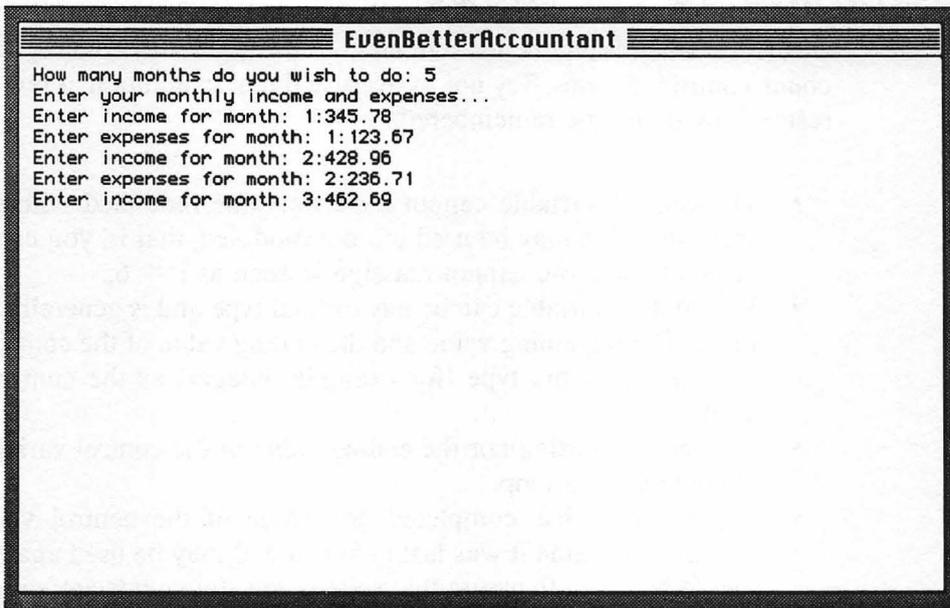


Fig. 5.4.

Backward Looping with Downto

The for statement automatically increases its loop one number at a time. You can alter this rule by substituting the **to** option with **downto**. **Downto** allows the loop to decrease its count by one.

Can you guess the number of times the following loop will execute?

```
for z := 10 downto 6 do
  writeln (z);
```

If you guessed five, you are correct. This loop outputs

```
10
9
8
7
6
```

Summing up For Loops

Be careful when using the for statement. A program may contain a number of count-controlled loops. Try not to overuse them. In addition, a few important restrictions should be remembered:

- The control variable cannot have its value redefined within the loop itself. Its value may be used but not modified; that is, you can write the value of *i* but you cannot reassign it, such as *i* := 6;
- The control variable can be any ordinal type and is generally an integer value. The beginning value and the ending value of the control variable must be the same type (for example, integer) as the control variable itself.
- Neither the starting nor the ending value of the control variable can be altered within a loop.
- When a loop has completed, the value of the control variable still contains the value it was last assigned and may be used again, but care should be taken to assure that subsequent statements are not dependent upon an earlier value. For example, you may define a general loop counter (count) and use it in several different for loops. If you try to use the final value later in the program, be sure you know which loop executed last to determine the exact value.
- The starting value must be less than or equal to the ending value. If the values are equal, the loop executes once only. In the case of downto the opposite is true; the starting value must be greater than or equal to the ending value.

In the next section I discuss one more looping control structure, the repeat...until statement. This structure is the last of Turbo Pascal's three looping options.

The Repeat Statement

The repeat statement consists of two parts: the body and the termination condition. It is essentially an upside-down while statement in which the condition is not checked until the loop has performed at least one iteration. The statement is used to repeat a set of statements (the body) until a condition is true (the termination condition). A sample of this statement is

```
repeat
  statement1;
  statement2;
  statement3;
  .
  .
  .
  statementn;
until (expression is true);
```

The reserved word `repeat` is followed by a group of statements terminated by the reserved word `until`. The control structure tests the loop at the end of the loop. If the expression is evaluated as a false condition, the statement sequence is executed again. If the condition is true, the loop ends. Unlike `while` and `for` statements, `repeat...until` does not require a begin-end pair to designate a block of executable statements. The `repeat` statement contains a body of multiple statements that are delimited by the reserved word `until`.

As with the `while` statement, you should include a condition in which the expression will be true and the flow exits the loop. Otherwise the loop will be infinite. This simple program example satisfies this requirement:

Program FamilyBudget;

```
Var
  rent, utilities, groceries, car, misc:real;
  net_income, budget          :real;

begin
  writeln('This program calculates your monthly family budget.');
```

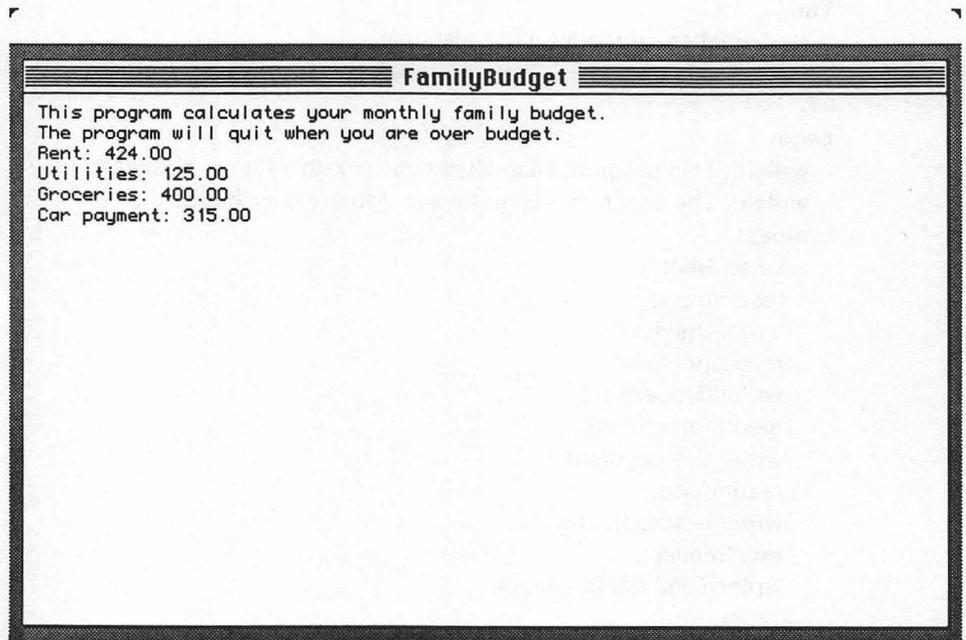
```
writeln('The program will quit when you are over budget.');
```

```
  repeat
    write('Rent:');
    readln(rent);
    write('Utilities:');
    readln(utilities);
    write('Groceries:');
    readln(groceries);
    write('Car payment:');
    readln(car);
    write('Miscellaneous:');
    readln(misc);
    write('Total net income:');
```

```
    readln(net_income);
    budget := net_income - (rent + utilities + groceries + car + misc);
    writeln;writeln;
    writeln('You are left with $', budget:10:2);
until(budget <= 0.0);
writeln;writeln;
writeln('Your expenses exceeded your net income!!!');
readln;
end.
```

This program calculates a family's monthly budget. A sample run of the program is shown in Figure 5.5.

All the statements in the loop will execute at least once. The program first instructs the user to enter a series of expenses (rent, utilities, groceries) followed by the user's net income. Expenses are totaled and then subtracted from net income to arrive at a budget for this particular month. If net income is greater than expenses, the program repeats another calculation (calculate another month). The mechanism used to exit the loop is provided by the condition at the end of the loop—until (budget <= 0.0). Specifically, when the net income for a particular month is less than the total amount of expenses, program flow exits the loop and the program ends.



```
FamilyBudget
This program calculates your monthly family budget.
The program will quit when you are over budget.
Rent: 424.00
Utilities: 125.00
Groceries: 400.00
Car payment: 315.00
```

Fig. 5.5.

While versus Repeat...Until

Because of the similarity in structure of the while loop and the repeat...until loop, they are often compared. These two loops, however, have one significant difference regarding their execution. Both must test a condition in order to execute the loop itself. The location of the decision test is the primary difference between the two. Specifically, the while statement tests the loop condition before executing the loop. On the other hand, the repeat...until statement tests the loop condition at the end of the loop. In other words repeat...until always executes its loop at least once. In addition, the result of the tested condition has just the opposite effect on each loop. The while statement executes and continues to loop as long as the tested condition is true; the repeat statement continues to loop as long as the tested condition is false. Which to use is entirely up to you, but for most instances the while version will be more appropriate.

Nested Loops

A nested loop is a loop within a loop. I have illustrated nesting program structures before, such as the compound begin-end pair and nested if statements. The logic for nesting a loop is similar. In simple terms the outer loop executes its initial task and then waits until the inner loop completes all of its loops or tasks. Here's a simple example that illustrates a nested for statement:

Program NestedLoop;

Var

loop1, loop2, cnt1, cnt2:integer;

begin

```
write('How many times do you want');
write('the outer loop to execute:');
readln(loop1);
write('How many times do you want');
write('the inner loop to execute:');
readln(loop2);
for cnt1 := 1 to loop1 do
begin
writeln('Outer loop iteration #', cnt1);
for cnt2 := 1 to loop2 do
writeln('Inner loop iteration #', cnt2);
end;
```

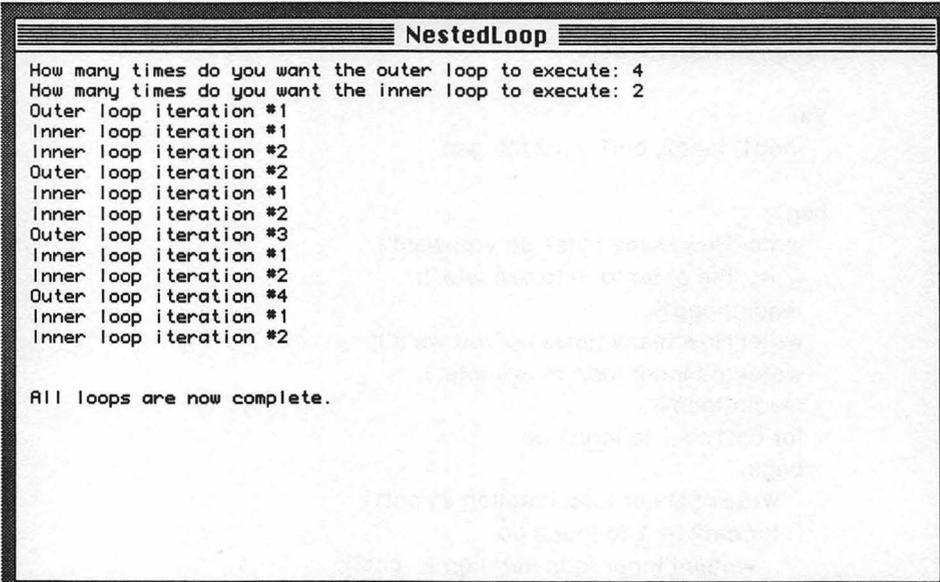
```
writeln;writeln;  
writeln('All loops are now complete.');
```

This program requests you to select the number of times you want the nested loop to execute. You enter a value for the outer loop and then you are requested to enter a value for the inner loop. In each case the outer loop and inner loop will execute the number of times assigned to the ending value (loop1 and loop2) of the control variable (cnt1 and cnt2). A sample run of this program is shown in Figure 5.6.

One More Loop: The Goto Statement

While and repeat...until are conditional looping statements, and if and case can perform a conditional branch. Conditional structures execute a selected number of statements depending on the value (condition) of an expression. Pascal offers one additional looping structure that alters program flow unconditionally. This is the goto statement.

Goto is common among programming languages, for example BASIC. However, it is not generally recommended because of its uncontrollable



```
NestedLoop  
How many times do you want the outer loop to execute: 4  
How many times do you want the inner loop to execute: 2  
Outer loop iteration #1  
Inner loop iteration #1  
Inner loop iteration #2  
Outer loop iteration #2  
Inner loop iteration #1  
Inner loop iteration #2  
Outer loop iteration #3  
Inner loop iteration #1  
Inner loop iteration #2  
Outer loop iteration #4  
Inner loop iteration #1  
Inner loop iteration #2  
  
All loops are now complete.
```

Fig. 5.6.

nature and its inconvenience when trying to trace program flow. Its purpose is to pass control of the program flow unconditionally from one point in a program to another, skipping any statements between the two points. A common use of goto is to exit from a loop.

The point to which the flow of program control is transferred is identified with a label. The general format of the goto statement is

```
goto label;
```

Label represents a valid name—with naming rules identical to those of variables—plus an unsigned integer (for example, 1 or 1000). A label is declared in the label section of the program declarations. Labels are declared immediately after the program name and before any other declaration. So the ordering of the declaration sections I have discussed so far is labels, constants, types, then variables. A sample label declaration:

```
Program ShowLabels;
```

```
Label  
  here;
```

```
begin  
.  
.  
.  
end.
```

An example of how to use this label is

```
here: writeln('We used a goto statement');
```

where “here” is a valid label (declared above) and the statement

```
goto here;
```

although it does not sound grammatically correct, causes program flow to jump to the Writeln statement denoted by “here.”

Pascal offers several control structures designed for solid structured programming technique. The goto statement is not one of them. Programs can be written efficiently without goto. I was tempted to omit this statement from my discussion, but you do have the right to know it is available. Compare it with the control structures that have only one exit and one entry. Which

structure would you like to read, follow, or debug? I think you will agree that you should avoid using goto.

This chapter presents several types of program loops. You learned about the three important looping structures, while loops, for statements, and repeat...until loops; about counting loops and loops that accumulate; about the differences between loops.

It's time to take another break. Review the summaries and exercises that follow. When you're ready, continue to Chapter 6 to study procedures and functions or subroutines, a very important feature of structured programming in Pascal.

Review Summary

1. The while statement is a conditional control structure that repeats itself until a certain condition is false.
2. An infinite loop repeats itself endlessly.
3. A counter is a variable that increases or diminishes by one each time a loop is executed.
4. An accumulator is a special form of counter. An accumulator is used to collect or sum a total.
5. A for statement is used to execute statements repeatedly. The for statement executes with its own built-in counter that either rises or falls by one each time the loop executes.
6. The repeat statement is used to repeat a set of statements until a condition is true. The repeat statement always executes at least once and is delimited by the reserved word until followed by the test condition.
7. These are all conditional loops. One loop that branches unconditionally is the goto statement. Its purpose is to pass control of program flow unconditionally from one point in a program to another. The point to which flow of program control is transferred is identified with a label.
8. Labels are generally declared immediately after the program name and before any other declaration such as const, type, or var.

Quiz

1. Give examples of a counter and an accumulator and state how they differ.
2. Does the position of the counter inside a loop have any effect on the program? Why or why not?

3. Write a while loop that reads input of five numbers and then displays the accumulated sum once.
4. When and why are begin-end pairs used within a for statement?
5. Write a program that permits interactively specifying the number of iterations of a for loop.
6. What is the primary difference between a while loop and a repeat...until loop?
7. What are nested loops used for?
8. Why should goto statements be avoided?

6

Procedures, Parameters, and Units

Structured Design

What Is a Procedure?

What Does a Procedure Consist Of?

Where Is a Procedure Placed?

How Are Procedures Used?

Using Variables with Procedures

Global versus Local Variables

Passing Information with Parameters

Variable Parameters

Value Parameters

Introducing Functions

Compiler Directives

The Turbo Pascal Unit

The Uses Clause

PasConsole Information

Using UnitMover

Review Summary

Quiz

In this chapter you will learn:

- How to write a program that features structured design.
- What a procedure is and how to define one.
- How to use a procedure with variables and parameters.
- What a function is and how to define one.
- What compiler directives are and how to use them.
- What a Turbo Pascal Unit is and how to work with them.
- What the PasConsole environment is and how it simulates standard Pascal I/O.
- How to use UnitMover.

Thus far I have covered many important programming tools from which you can build useful programs. All of our examples have used a single program module or block to accomplish a single task. In this chapter I discuss ways of putting together a group of modules, each performing a single task. When the individual modules are taken collectively, a larger programming task is performed; this is called *modular programming*. One mechanism used to accomplish this goal is the *procedure*.

Structured Design

Large programming objectives are often broken down into smaller ones. Subprograms are further divided until each subprogram consists of a few manageable statements. These miniprograms are generally easier to solve than the original macro program. Each subprogram performs a particular task, such as processing data mathematically or printing results. The individual subprograms are *modules*. The final program consists of a collection of modules. This is referred to as *structured programming* because of its hierarchical tree nature.

In this chapter we are going to discuss a particular type of structured design called *top-down design*. In *top-down design*, or called *stepwise refinement*, the original macro problem is represented by a main module that may consist of the major programming steps required to solve a problem. The main module calls upon individual subprograms to solve specific tasks. When all the simpler tasks are solved, the original problem is solved.

The main module is divided into smaller and smaller modules. Each module represents a specific level of programming that performs a task independently of the other modules. For example the main module can be depicted as level 0, the next set of modules as level 1, and so on (Figure 6.1). Any module within this tree structure can make demands on any module at a lower level.

The program as a collection of modules is a very important feature of Pascal. Turn your attention to the procedure, which identifies a subprogram or module.

What Is a Procedure?

A procedure is a declaration used to identify a collection of Pascal statements that perform a specific task. The procedure represents a subprogram and is executed each time the procedure is invoked. To use a procedure you assign it a name and then call it by name from some other point in the program. Once a procedure has executed, the flow of program control returns to the calling routine, for example `Writeln` and `Readln`. Although they may appear

Where Is a Procedure Placed?

A procedure must first be declared and is placed immediately following the **var** section of the main program. The keyword **procedure** is followed by a name and then a semicolon. The body of this miniprogram is completed with an **end** followed by a semicolon.

How Are Procedures Used?

Suppose a program is to perform a number of statements over and over again. You can rewrite the statements each time they are required or call a miniprogram or procedure to accomplish the same task. When the task is completed, the program proceeds. For example, examine the following program, which contains two mathematical functions, one for addition and one for multiplication:

```
Program MathAce;
```

```
Var
```

```
  num1, num2, sum, product : real;
```

```
Procedure Add;
```

```
begin {start of procedure Add}
```

```
  sum := num1 + num2;
```

```
  writeln ( 'The sum of your numbers is ', sum:10:2 );
```

```
end; {end of procedure Add}
```

```
Procedure Mult;
```

```
begin {start of procedure Mult}
```

```
  product := num1 * num2;
```

```
  writeln ( 'The product of your numbers is ', product:10:2 );
```

```
end; {end of procedure Mult}
```

```
begin {start of the main program}
```

```
  write ( 'Enter a number: ' );
```

```
  readln ( num1 );
```

```
  write ( 'Enter another number: ' );
```

```
  readln ( num2 );
```

```
  writeln; writeln;
```

```
  writeln ( 'We are now computing the addition...' );
```

```
  Add;
```

```
  writeln; writeln;
```

```
  writeln ( 'We are now computing the multiplication...' );
```

```
  Mult;
```

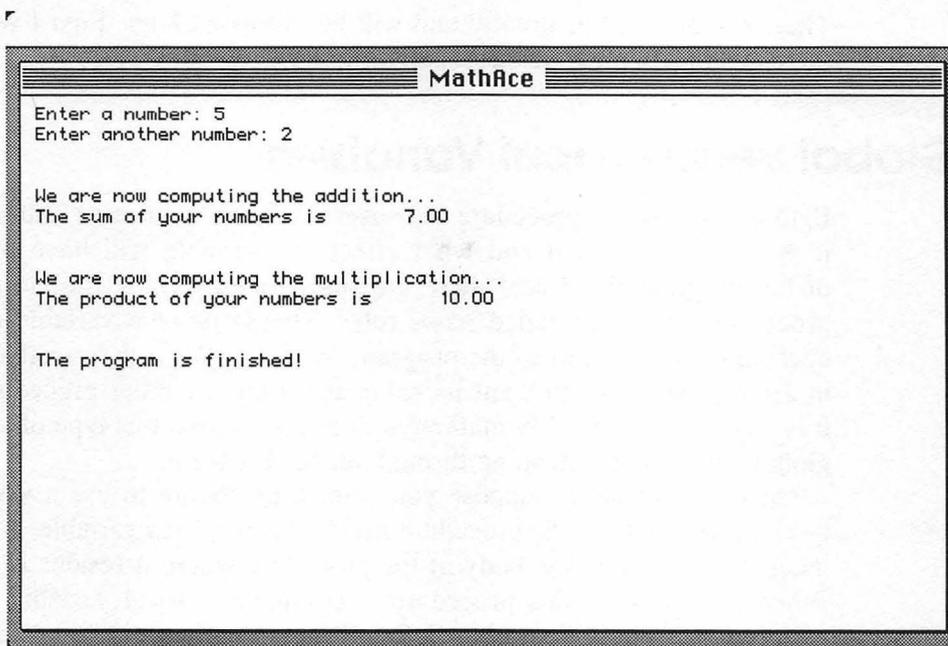
```
writeln; writeln;  
writeln ( 'The program is finished!' );  
readln;  
end.    {end of main program}
```

I think it's important that you understand the flow of program control in this example. Take a look step by step.

When you execute the program above, you are asked to enter two numbers. This is accomplished by the two `Write` and `Readln` combinations in the main routine. Next a message informs you that the addition is about to be performed. At this point the routine `Add` is invoked and control jumps to this procedure. The `Add` routine has all of its statements enclosed within a **begin-end** pair. The first statement executed is the calculation of the sum. Next the sum is printed out and the procedure ends. Then control returns to the main routine and a couple of blank lines are printed via two `Writeln`s without parameters and a message indicates that the multiplication is about to be performed. As you can see, when the program left the main routine, it was about to perform this sequence of `Writeln` statements and as soon as it returns, they are executed. At this point `Mult` is invoked and control is passed to it, just as it was to `Add`. First the product is calculated and then it is displayed. This procedure then ends and control is transferred back to the main program routine, where the message "This program is finished!" is displayed. A sample run of this program is shown in Figure 6.2.

Note two important points. First, the names of the procedures `Add` and `Mult` appear at the beginning of those procedures just as the program name appears at the beginning of the program. Second, the procedure's statements are enclosed within a **begin-end** pair just like those of the main routine, and compound statements appear within **begin-end** pairs just as in the main program routine.

Writing such programs is easily accomplished using top-down design. The first step in writing the code involves breaking down the main program into manageable miniprograms. The logic of the main program is developed first, as in the previous example all the initializing input/output statements are in the body of the main program. Next break the building blocks of the program into separate smaller tasks, or procedures, such as to do addition and its output, another to do multiplication and its output, and so on. Whenever such a task is required, call on that particular procedure again. Remember, you can call a procedure from any point in the main program as many times as you want. You can even have one procedure call another procedure. This should be easy to visualize, since the main program is nothing more than a special procedure itself. The key to allowing one procedure to call another is



```
MathAce
Enter a number: 5
Enter another number: 2

We are now computing the addition...
The sum of your numbers is      7.00

We are now computing the multiplication...
The product of your numbers is   10.00

The program is finished!
```

Fig. 6.2.

that the called procedure must be placed above the calling procedure, so the compiler knows where to find it; otherwise, the called procedure is flagged as an unknown identifier.

A procedure may contain a miniprogram that completes a more complex task than adding or multiplying two numbers. However, the concept is the same. It is important that you understand the logic behind structured programming with a top-down design, in which large programs are really a collection of smaller ones. Later in this book I will introduce applications that use this very important feature.

Using Variables with Procedures

Understanding and defining a procedure is simple. However, there are a few additional rules to discuss. In the previous example, I created a program that transmitted information from a procedure to the main program. All the variables were global to the program, and so any procedure was permitted to modify them at any time. I made global changes in these variables with the procedures `Add` and `Mult`. However, variables can be passed from one procedure to another, and changes therefore need not be global. There are two methods of passing values to a procedure, by value and by address.

These methods differ greatly and will be discussed later. First I will discuss the concept of global versus local variables.

Global versus Local Variables

Before you write a procedure that uses variables, you must determine how it is going to be used and what effect the variable will have on the rest of the program. In Pascal there are rules for governing the variables in a procedure; these are called *scope* rules. The *scope* of a variable defines its accessibility to the rest of the program. For example, if the variable *a* is used in a procedure, do you want its value accessible by other procedures? If so, it is a *global* variable. My mathematics program used this type of variable. A global variable has meaning throughout the program.

On the other hand, suppose you want a procedure to use a variable that has no effect outside the procedure itself. This is a *local* variable, one that has meaning only while the body of the procedure where it resides is executing. When execution of this procedure is complete, a local variable no longer exists. How do you identify or define the scope of a variable? As you recall, all variables must be declared before they are used. This rule applies whether the variable is used in the main procedure or another. Where you place the variable declaration defines its scope. If the variable is declared after a procedure, it may be accessed only by that procedure. If the variable is declared outside of a procedure block, as our variables were, it is accessible by the entire program.

Be careful if you declare a local variable with the same name as a global one because once the procedure transfers control to the calling routine, the local value of the variable will be lost and the calling routine will use the global value. The value assigned to the variable will be the last global value assigned from the main declaration. The result may not be what you expected.

How to choose between a local variable and a global one? Good programming practice dictates that whenever possible you should use a local variable instead of a global one. It's wise to use a procedure to complete its task and then go about your business with the rest of the calling routine. Why? A practical question, since our first example used global variables.

A global variable can wreck a good program. The reason is simple. In a large program you may wish to use a variable again. A global variable retains its value and can produce undesirable side effects later in your program. A local variable, however, has no effect on the main program and can be used repeatedly without unexpected results.

Perhaps the best reason to use a local variable in a procedure is for flexibility when making multiple calls to the same routine. Each time a call

to a procedure is made, the variables start from scratch. If global variables are used, the values left over from the previous execution of the procedure are still in effect.

Regardless of the reason, most experienced Pascal programmers agree that unnecessary global variables often lead to bugs. Such bugs are difficult to find even for the most seasoned programmer. As a rule of thumb you should use local variables whenever possible.

I just made a case for using local variables, but I haven't described the method for exchanging information with a procedure when local variables are used. This process requires a mechanism called a parameter.

Passing Information with Parameters

Exchanging information between a procedure and the main program is efficiently accomplished using a parameter, a variable that allows a value to be passed between a procedure and the main program or another procedure. Parameters are listed in the procedure definition statement. For example, a procedure named `MathSolution` might be written as

```
Procedure MathSolution ( Var x, y : Real );
```

where the variable parameters `x` and `y` are declared to be real data types. The reserved word **procedure** is followed by its identifier `MathSolution`, and the parenthetical declaration is followed by a semicolon. This example defines `x` and `y` as variable parameters (`var`). It is important to note that placement of the parenthetical list declares the variables local, and no other declaration for these variables is allowed in the procedure.

A parameter may be either `var` (*variable*) or a value, which is the default. The difference between the two involves how information is passed between a procedure and the calling routine. A `var` parameter is passed by reference and acts as a location pointer in memory. If it is a value parameter, the information is passed by value, and only a copy of the value is passed. The example above is a variable parameter, which I discuss next.

Variable Parameters

Calling a procedure using variable parameters is slightly different from calling a procedure without them. Suppose I want to pass the values of the variables `num1` and `num2` from the main routine to the routines `Add` and `Mult`. My program would look like this:

```
Program MathAce;

Var
  num1, num2 : real;

Procedure Add ( Var aNum1, aNum2 : real );

Var
  sum : real;    {declare sum local to Add}

begin  {start of procedure Add}
  sum := aNum1 + aNum2;
  writeln ( 'The sum of your numbers is ', sum:10:2 );
end;    {end of procedure Add}

Procedure Mult ( Var mNum1, mNum2 : real );

Var
  product : real;    {declare product local to Mult}

begin  {start of procedure Mult}
  product := mNum1 * mNum2;
  writeln ( 'The product of your numbers is ', product:10:2 );
end;    {end of procedure Mult}

begin  {start of the main program}
  write ( 'Enter a number: ' );
  readln ( num1 );
  write ( 'Enter another number: ' );
  readln ( num2 );
  writeln; writeln;
  writeln ( 'We are now computing the addition...' );
  Add ( num1, num2 );
  writeln; writeln;
  writeln ( 'We are now computing the multiplication...' );
  Mult ( num1, num2 );
  writeln; writeln;
  writeln ( 'The program is finished!' );
  readln;
end.    {end of main program}
```

Notice that num1 and num2 are the only variables declared globally; I removed the global declarations for sum and product. Also, the procedures Add and Mult now have parameters in both their calls and their definitions.

For instance,

```
Procedure Add ( Var aNum1, aNum2 : real );
```

declares Add as a routine with two parameters, aNum1 and aNum2, both reals and passed as var parameters.

Now when I invoke Add, I place the values of num1 and num2 within the parentheses in the call. This is identical to the way I have been calling writeln and readln all along; I simply place the parameters in the order in which the procedure expects them. The rules outlined for Add apply to the procedure Mult. Remember, the variables sum and product are no longer declared globally. They are now local to the routines in which they are used. That is, now only Mult can access the variable product and Add is the only procedure that may access the variable sum.

We refer to this as passing parameters by reference primarily because the value passed to the called routine is not the value of the variable but the address of the variable. For example, if the variable num1 held the value 12, the item is passed to Add and Mult is not 12 but rather the location in the Macintosh's memory where the value of num1 is stored. Recall that variables have their own memory locations set up when they are declared; passing a parameter by var indicates that the called routine is to know the location of the variable so that it can modify it. Since both Add and Mult know the addresses of num1 and num2, they can change the value of either of them by changing either aNum1/aNum2 (for Add) or mNum1/mNum2 (for Mult). For example, I'll modify the program a bit so that the values of num1 and num2 are changed within Add like this:

```
Program MathAce;
```

```
Var
  num1, num2 : real;
```

```
Procedure Add ( Var aNum1, aNum2 : real );
```

```
Var
  sum : real;    {declare sum local to Add}
```

```
begin    {start of procedure Add}
  sum := aNum1 + aNum2;
  writeln ( 'The sum of your numbers is ', sum:10:2 );
  (*****
  (* Now let's modify the values of num1 and num2 via *)
```

```

(* aNum1 and aNum2... *)
(*****)
aNum1 := aNum1 + 1.0;
aNum2 := aNum2 + 1.0;
end;    {end of procedure Add}

Procedure Mult ( Var mNum1, mNum2 : real );

Var
  product : real;    {declare product local to Mult}

begin    {start of procedure Mult}
  product := mNum1 * mNum2;
  writeln ( 'The product of the new numbers is ', product:10:2 );
end;    {end of procedure Mult}

begin    {start of the main program}
  write ( 'Enter a number: ' );
  readln ( num1 );
  write ( 'Enter another number: ' );
  readln ( num2);
  writeln; writeln;
  writeln ( 'We are now computing the addition...' );
  Add ( num1, num2 );
  writeln; writeln;
  writeln ( 'We are now computing the multiplication...' );
  Mult ( num1, num2 );
  writeln; writeln;
  writeln ( 'The program is finished!' );
  readln;
end.    {end of main program}

```

By modifying the values of aNum1 and aNum2 in Add, I change the values of the globally declared num1 and num2. A sample run of this program is shown in Figure 6.3.

You can see that the values of num1 and num2 were changed in Add by the fact that the resulting product in Mult is based upon the new values. One other point I would like to mention here is that the type declarations for parameters in the procedure declaration line *must* match up with the types of parameters being passed to the routines. For example, the declarations for aNum1 and aNum2 in Add are set up as reals. When I invoke the Add routine, I pass the variables num1 and num2, which are declared globally as reals also; we could *not* have passed a variable of any other type. This rule holds true for parameters passed by value, which is our next topic.

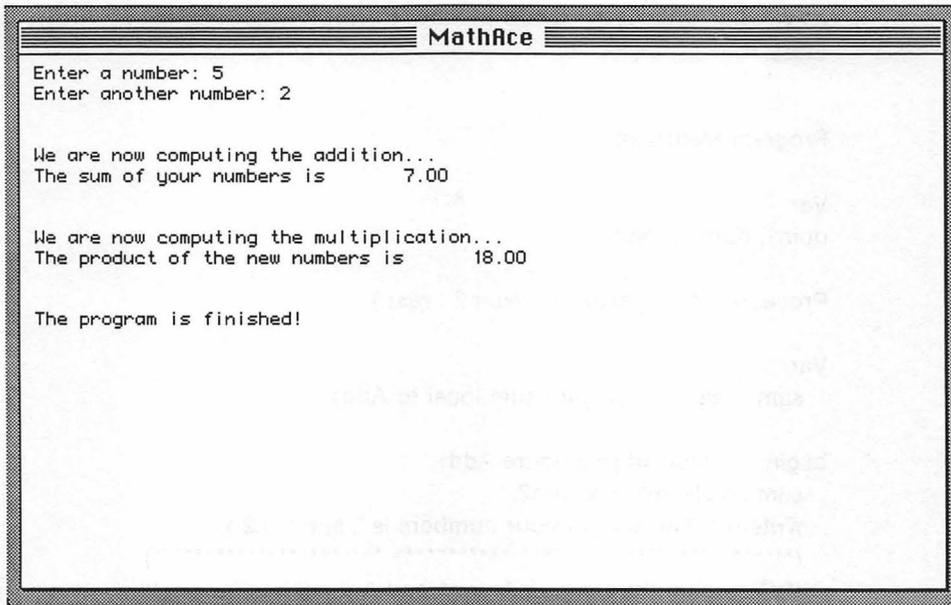


Fig. 6.3.

Value Parameters

A second way to pass parameters is to use value. Value parameters pass to the called procedure a copy of the value of the parameter. A value parameter has its memory location inside the procedure, and its original value cannot change. Only the *copy* is changed. Perhaps a simple distinction between a value parameter and a variable parameter is that a value parameter is fixed, like a constant. A variable parameter can take on any number of new values; only the address is fixed.

When you declare value parameters, eliminate the key word **var** from the procedure definition statement. For example, a procedure named Continue might be written as

```
Procedure Continue ( x : char );
```

where the parameter list contains a single formal parameter, x. The identifier x is declared as a Char data type. Of course the parameter list may include any number of formal parameters as long as they are matched in number and data type with the variables in the calling statement. A valid statement to invoke Continue might be

```
Continue ( answer );
```

where the variable “answer” represents a value parameter previously declared as a Char data type. Look at how to implement MathAce with value parameters:

```
Program MathAce;
```

```
Var
num1, num2 : real;
```

```
Procedure Add ( aNum1, aNum2 : real );
```

```
Var
  sum : real;    {declare sum local to Add}
```

```
begin  {start of procedure Add}
  sum := aNum1 + aNum2;
  writeln ( 'The sum of your numbers is ', sum:10:2 );
  (*****
  (* Changing the value of Anum1 and Anum2 has no      *)
  (* effect on num1 and num2.                          *)
  (*****
  aNum1 := aNum1 + 1.0;
  aNum2 := aNum2 + 1.0;
end;    {end of procedure Add}
```

```
Procedure Mult ( mNum1, mNum2: real );
```

```
Var
  product : real;    {declare product local to Mult}
```

```
begin  {start of procedure Mult}
  product := mNum1 * mNum2;
  writeln ( 'The product of your numbers is ', product:10:2 );
end;    {end of procedure Mult}
```

```
begin  {start of the main program}
  write ( 'Enter a number: ' );
  readln ( num1 );
  write ( 'Enter another number: ' );
  readln ( num2 );
  writeln ; writeln;
  writeln ( 'We are now computing the addition...' );
  Add ( num1, num2 );
  writeln; writeln;
  writeln ( 'We are now computing the multiplication...' );
```

```
Mult ( num1, num2 );  
writeln; writeln;  
writeln ( 'The program is finished!' );  
readln;  
end.    {end of main program}
```

As you can see, the declarations for Add and Mult look the same as before except that the parameters are not preceded by `var`. As a result the modification of `aNum1` and `aNum2` in Add has no effect on the values of `num1` and `num2`; you can see this by the resulting product displayed via Mult.

You need not choose between variable and value parameters. The examples we have seen above show the use of only one kind at a time, but one of the parameters could be a variable and the other a value. For example, the declaration for Add could be

```
Procedure Add ( aNum1 : real; Var aNum2 : real );
```

where `aNum1` is a value parameter and `aNum2` is a variable. Also, keep in mind that this

```
Procedure Add ( Var aNum1 : real; aNum2 : real );
```

declares `aNum1` as a var parameter, but `aNum2` is a value parameter; each individual var parameter must be preceded by the var identifier. However,

```
Procedure Add ( Var Anum1, Anum2 : real );
```

declares both `aNum1` and `aNum2` as var parameters because they are grouped together via a multi-variable declaration by using the comma as a separator.

Finally, for those of you interested in the internal workings of the Macintosh with Turbo Pascal, parameters in Turbo are passed via the stack. A *stack* is a block of memory used by the computer to keep track of various items like where to go after a procedure is finished and the value of a procedure's parameters. When you call a procedure, the Macintosh notes where it was when it encountered the procedure call and pushes that instruction's address on the stack. Then each procedure's parameters are placed on the stack so that the procedure knows where to look for them. When a value parameter is passed, the value of that variable is generally placed on the stack, and when a variable parameter is passed, the address of that variable is placed on the stack. I say generally the value of a nonvariable parameter is placed on the stack because Turbo Pascal uses a 4-byte pointer at the address if the variable is larger than 4 bytes. For example, given `String[50]` as a value pa-

parameter, instead of placing all 51 bytes on the stack, Turbo sets a pointer, just as it would if the same `String[50]` were a variable parameter.

Now that you have a good deal of knowledge about procedures, let's look at another type of subroutine that is used specifically to return a value, the function.

Introducing Functions

Pascal offers an additional program structure called a **function**. A function, similar to a procedure, is a subroutine, a set of statements that performs some task. Sounds just like a procedure, doesn't it? They even look similar. Both a procedure and a function have a heading and a body. In the case of a function, the heading substitutes the reserved word **function** for **procedure**. The single important difference between functions and procedures is that a function is designed to return a specific computed value to the calling routine. Let's look at what this means.

If you have mastered the procedure, the function will be easy to grasp. The function looks something like a procedure. However, the heading includes the reserved word **function**, the function name, a parameter list (if any), and the *type of function*. To illustrate, suppose I want to write a function that will return a Boolean value of `TRUE` if the character passed to it is a lowercase letter and `FALSE` otherwise. A function would do this:

```
Function IsItLower ( inChar : char ) : boolean;
begin
  if ( ( inChar >= 'a' ) AND ( inChar <= 'z' ) ) then
    IsItLower := True
  else
    IsItLower := FALSE;
end;
```

This defines a function named `IsItLower` followed by a single parameter, `inChar`. Notice the heading of the function declares the type of value returned by the function to be a Boolean. The body of the function, like the procedure, follows the function definition statement. Here is a `begin-end` pair and an `if` statement that checks the value of the `inChar` parameter to see if it is a lowercase character. I might use this function in a program like this:

```
Program LookAtInput;
```

```
Var
  aChar   :char;
  lowerCase : boolean;

Function IsItLower ( inChar : char ) : boolean;

begin   {start of function IsItLower}
  if ( ( inChar >= 'a' ) AND ( inChar <= 'z' ) ) then
    IsItLower := TRUE
  else
    IsItLower := FALSE;
end;    {end of function IsItLower}

begin   {start of main program routine}
  repeat
    write ( 'Enter a character: ' );
    readln ( aChar );
    lowerCase := IsItLower ( aChar );
  until ( LowerCase = TRUE );
  readln;
end.    {end of main program routine}
```

IsItLower is invoked by the statement

```
lowerCase := IsItLower( aChar );
```

In short, IsItLower is called, and within the function the value returned is either TRUE or FALSE based on the if statement within it. This is an important point about functions; you can (and should) assign a value to the function itself to be used by the calling routine. The reason I say that you should assign a value to it is that this is what makes a function different from a procedure. In this case I could have used a procedure like this:

```
Procedure IsItLower (inChar : char; Var aLower : boolean );

begin
  if ( ( inChar >= 'a' ) AND ( inChar <= 'z' ) ) then
    aLower := TRUE
  else
    aLower := FALSE;
end;
```

The parameter aLower could then be looked at by the calling routine to see if the character was lowercase or not. This sort of a subroutine is

better suited for use as a function than as a procedure because it returns one parameter. You are, of course, free to use either type of implementation you deem appropriate.

As with procedures and their parameters, you need to make sure that the function's parameter types match up with the actual variable types that are passed. In addition, you need to ensure that the declared type of the function matches up with the type of the variable you assign to it. For example, `IsItLower` is Boolean and the `lowerCase` variable its result is assigned to is also Boolean.

You should also notice that the function's name appears on the left side of the assignment operator (`:=`) only. It makes no sense to say something like this in our `IsItLower` function:

```
saveBool := IsItLower;
```

where `saveBool` is defined to be a Boolean variable. Be sure to make assignments only to the function name and not with the name in your functions. Let's take a look at a way to rewrite the example using `IsItLower` more economically:

```
Program LookAtInput;
```

```
Var
```

```
  aChar : char;
```

```
Function IsItLower ( inChar : char ) : boolean;
```

```
begin  {start of function IsItLower}
  if ( ( inChar >= 'a' ) AND ( inChar <= 'z' ) ) then
    IsItLower := TRUE
  else
    IsItLower := FALSE;
end;  {end of function IsItLower}
```

```
begin  {start of main program routine}
  repeat
    write ( 'Enter a character: ' );
    readln ( aChar);
  until ( IsItLower(aChar) = TRUE );
  readln;
end.  {end of main program routine}
```

In this version I remove the Boolean variable `lowerCase` and call `IsItLower` implicitly within the `until` portion of the `repeat...until` statement. Calling `IsItLower` at this point tells Turbo Pascal to use the returned value from

the function as the condition in the repeat...until loop. This implicit check can also be written like this:

```
until (IsItLower(aChar));
```

Turbo views that statement as a Boolean expression and if it is TRUE, exits the loop. Boolean expressions such as this are abbreviated like that very often. For example, the if statement

```
if (done = TRUE) then
```

can be abbreviated as

```
if (done) then
```

This shorthand notation is often quite confusing to the novice, but with a little experience you will be using it too to shorten programs and save keystrokes.

Again, for those interested in the inner workings of Turbo, when you invoke a function, space is saved on the stack to hold the result of the function. This stack space is determined by the function type. For the simple data types like Boolean, Char, Integer, and enumerated types, either 2 or 4 bytes are reserved on the stack for the function result. One-byte types like the Boolean and Char are padded out to 2 bytes because the 68000 microprocessor works on even-number byte boundaries. This is a common occurrence on 68000-based machines and you should keep it in mind later when looking at type sizes. Just remember that odd-number-length types are generally padded with an extra byte to keep everything on an even boundary. For other function types, like strings, a pointer to the returned value is placed on the stack for the same reason you should pass larger data types as variable parameters even though Turbo Pascal will generally pass a pointer to value parameters larger than 4 bytes anyway: stack space and processing time considerations.

Now that you are familiar with functions, take a look at some ways to control the Turbo Pascal compiler from within your programs via compiler directives.

Compiler Directives

All of the compiler directives available to you in Turbo Pascal are described in detail in Appendix C of the manual you received with the compiler. I feel it is appropriate to discuss many of the more popular ones here so that you will understand them when you encounter them in this book and in other programs and books.

To start with, a compiler directive is a statement that allows you to alter the way Turbo Pascal goes about compiling and linking your program. Compiler directives are specified in a comment line by placing a dollar sign immediately after the `*` or `{` that opens a comment. Some compiler directives are flags that are set on or off by placing a plus or minus sign after the flag name (designated by a character) in the directive. For example, the line

```
{SD+}
```

tells the compiler to generate debug symbols in the program. This allows you to look at procedure and function names when using a debugger, a tool that allows you to follow the operations of a program and look at its internal execution. (Debugging will be discussed in detail in Chapter 12.) In order to set this flag on, use the directive like this:

```
Program Test;
{SD+}

Var
  aFlag : boolean;
.
.
  {further variable declarations}
begin
.
.
end.
```

The opposite of `{SD+}` is `{SD-}`, which tells the compiler *not* to generate debug symbols when compiling the program. Instead of using curly brackets to embed a compiler directive, I could use `(*` and `*`). For example, `{SD+}` could be written as `(*SD+*)`. The flag-type compiler directives all have a default value that sets the flag on or off whether you specify or not. For example, the default value for the directive to generate debug symbols is off, or `{SD-}`. I have not been stating it at the beginning of these programs, and I could have placed the `{SD-}` directive in each of them with no difference.

Another very useful compiler directive allows you to check I/O results after any I/O activity. This flag directive uses the character `T` and is either `{SI-}` or `{SI+}`. As in `readln`, if you accidentally enter a string value when an integer is expected, you will wind up with the system error bomb on your screen. One of the simple ways around this is to change your code from

```
Program RiskyIO;

Var
  anInt : integer;

begin
  write ( 'Enter an integer: ' );
  readln ( AnInt );
end.

to

Program GoodIO;
{$I-}

Var
  anInt : integer;

begin
  repeat
    write ( 'Enter an integer: ' );
    readln ( AnInt );
  until ( IOResult = 0 );
end.
```

In the second example of integer input, `{$I-}` turns off Turbo's internal I/O error checking; the default value for this directive is `{$I+}`, and because of this, any time we don't manually turn the option off, a system error will result in bad input. Instead of requesting the input via `write` and `readln`, I have incorporated a `repeat...until` loop that continues until `IOResult = 0`. What does this mean? `IOResult` is built in to Turbo Pascal so that you can determine if an I/O error occurred on your last request, for example on a `readln`. A value of 0 from `IOResult` indicates that the input was OK. A complete list of `IOResult` codes is provided in the Turbo Pascal manual in Appendix B: Error Messages and Codes. At this point we're interested only in a value of 0, since this means it is OK to proceed with the next step in the program. We will see a good deal more of `{$I-}` and `IOResult` in upcoming programs that deal with input through the keyboard as well as with I/O with disk files.

The next compiler directory is not a flag but a command that tells Turbo to include another file with the one being compiled. Let's say you are working with some rather large programs that share the exact same constants, types, and variables. Rather than entering these declarations for each program, you

can use the include-file directive. Say you have just entered the declarations into the Turbo editor like this:

Const

```
HiValue      = 100;
LoValue      = 1;
MaxStudents  = 30;
MinStudents  = 1;
```

Type

```
Week = ( Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
ClassType = ( Mathematics, Science, English, SocialStudies );
```

Var

```
numberOfStudents : MinStudents...MaxStudents;
aDay : Week;
whichClass : ClassType;
```

I usually want to have more declarations before using an include file, but this example is valid. Next I can save this file on the disk just as I can save any other Turbo file; notice that this is not a complete program and therefore cannot be compiled on its own. However, I choose to save it with the filename GENDECL.INC. I use the .INC extension to signify that this is a Pascal Include file. I can read it in other programs like this:

```
Program School;
{$I GENDECL.INC}
```

begin

```
write ( 'Enter the number of students: ' );
readln ( NumberOfStudents );
case NumberOfStudents of
  1 : writeln ( 'Wow! What a small class!' );
  2 : writeln ( 'I can"t believe there"s only two of you! ' );
  .
  .
  .
end;
end.
```

The {\$I GENDECL.INC} directive tells the Turbo compiler to get the named file (GENDECL.INC) and treat it as if that entire file appeared at that point in the program. This sort of include-file use is not limited to declarations.

You can keep some commonly used procedures or functions in include files and pull them into different programs via the include-file directive. However, this is more commonly performed via the unit, which I will discuss later in this chapter.

The next compiler directive is \$L, which allows you to link other files to the one you are compiling and linking. I haven't discussed linking much because it is always implied in the compiling options. For example, when I run a program via the Compile menu, first the program is compiled and then any necessary external modules are linked in to generate an executable program. These external modules are generally created by other compilers or assemblers such as the MDS assembler. Instruct Turbo to link in a module with the \$L directive like this:

Program Test;

```
{ $L SCRNRNNS.REL }
{ $L FILERTNS.REL }
Var
.
.
.
begin
.
.
.
end.
```

This example shows how to link in the files SCRNRNNS.REL and FILERTNS.REL. As shown, \$L must appear before the **begin** statement that starts the main program routine.

If you have compiled any programs to disk, you have probably noticed that the name of the executable icon on the desktop is the name in the **program** statement. For example, if your program starts with

Program Test;

when you compile to disk, the double-clickable icon that results is called Test. This naming convention may be altered via the next directive, \$0. If I place a \$0 directive in our program like this:

Program Test;

```
{$0 SimpleOutput}  
begin  
  writeln ( 'Hi there.' );  
  readln;  
end.
```

and compile to disk, the double-clickable file generated by Turbo will be called SimpleOutput instead of Test.

The next directive is one you will not use until the topic of arrays comes up in Chapter 9. This directive, \$R, is used to turn on and off the internal range checking on arrays. You have already been introduced to one form of array, the string. It is possible to refer to a particular character within a string by placing the position within two brackets after the string name. For example, given a string declared like this:

```
Var  
  name : String[20];
```

I can refer to the first character in the string like this:

```
name[1]
```

It is possible to try to look at an invalid index within a string; suppose I try to look at name[30]. I am not informed of any error if the \$R option is off. Therefore, any time you work with arrays, turn \$R on like this:

```
Program Test;  
{$R+}  
begin  
.  
.  
.  
end.
```

so that any improper index references are flagged by a system error. This concept may not appear clear or important right now, but once we get into arrays, it will be used always.

The next directive also uses \$R but instructs the compiler that you wish to refer to a resource file. Explained in the next chapter, resource files are used to hold information about windows, menus, dialog boxes, and so forth that you set up and use in your programs. Resource files are compiled with the utility RMaker into a form directly usable by Turbo Pascal. In short, you

write a resource file using the Turbo editor and save it with the extension `.R`. Compile it with RMaker, and a file is produced with the extension `.RSRC`. This `.RSRC` file may then be put into your Turbo Pascal programs via `$R`:

Program Test;

```
{ $R MYWINDOW.RSRC }  
begin  
.  
.  
.  
end.
```

You may then refer to the resources defined in the file `MYWINDOW.RSRC`. As I have said, I'll get more into resources in the next chapter, so don't worry if you don't understand the specifics.

The last directive I would like to discuss is `$U`, which specifies whether or not you want to use standard Turbo Pascal units. A *unit* is a group of related declarations in Turbo Pascal. A unit may consist of blocks of `const`, `type`, `var`, `procedure`, and `function` declarations. The Turbo Pascal unit allows you to break programs up into separately linkable modules. This is in contrast to include files, which are pulled into the program and compiled with it.

Turbo Pascal comes with several interface units so that you can work with all the ROM-based routines for graphics, sound, and so on. These interface units are described in Appendix D of your Turbo Pascal manual. The directive `$U` tells Turbo to pull in the standard I/O units `PasInOut` and `PasConsole`. This directive is always on by default—`{ $U+ }`—so that you may always refer to routines such as `ClearScreen` and `GoToXY`, which will clear the output screen and place the cursor in a specified location respectively. Later you will want to turn this directive off to do special Macintosh-style I/O with windows and so on. In order to demonstrate units and their uses and importance in Turbo Pascal, I start with a fundamental description of the unit.

The Turbo Pascal Unit

A Turbo Pascal *unit* is a collection of declarations of constants, types, variables, procedures, and/or functions. In your Turbo Pascal manual at the `QuickDraw` unit there are several pages of these declarations. A unit, however, may contain more than these declarations. It may also contain Pascal code for procedures and functions as well as private declarations for each of them. The Turbo Pascal unit consists of two sections: *interface* and

implementation. The interface section contains all the externally available declarations. For example, in PasConsole the procedure ClearScreen appears after the reserved word **interface** and therefore is in the interface section. In fact, all the items listed in any standard Turbo Pascal unit in the manual are declared in the interface section. There may, however, be some procedure and function blocks in the implementation section that are not in the units. In addition, you may have noticed that most of the procedure and function declarations in these units are followed by something like this: inline \$Axxx, where xxx is a hexadecimal number. This inline \$Axxx specification is a special pointer that tells Turbo where the code for these routines may be found in the Macintosh's ROM. There are also several routine declarations followed by the reserved word **external**. These declarations mean that the actual routines were written in assembly language and that their code appears in another file linked with the unit.

The structure of a unit is

```
unit NAME(#);  
interface  
  (LIST OF ACCESSIBLE DECLARATIONS)  
implementation  
  (LIST OF UNACCESSIBLE DECLARATIONS AND ROUTINES)  
end.
```

NAME is an identifier that specifies how the unit will be referred to by other modules. The # is any 2-byte value you choose, as long as it doesn't match any other unit number used by the program. Notice that the Turbo Pascal units defined in your manual use negative unit numbers, and with 2 bytes available, you have a wide range of possibilities for your unit numbers.

Compile a unit just as you would compile any other Turbo Pascal program. When you compile a unit to disk, the icon representing the compiled unit doesn't look like those of other compiled modules; compiled units have icons of miniature briefcases on the desktop.

The Uses Clause

You may have noticed that each unit defined in your manual contains a statement similar to this:

```
Uses MemTypes, QuickDraw, OsIntf;
```

This statement, which appears in ToolIntf, says that this unit uses three other units, MemTypes, QuickDraw, and OsIntf. Look at OsIntf. It says it uses

MemTypes and QuickDraw; QuickDraw says it uses MemTypes; MemTypes has no uses clause, so it doesn't require any other units. When you write programs, you must specify any units via the uses clause. These declarations should be set up so that any units called up are already declared. Above, for example, OsIntf refers to both MemTypes and QuickDraw, so they must be declared before it. In addition, QuickDraw calls up MemTypes, so MemTypes must be declared before QuickDraw.

In earlier examples I could have placed the line

```
Uses PasInOut, PasConsole;
```

after the program declaration; this would have told Turbo to have available the routines listed in these units. \$U is always on by default. Take a look at a few of the routines available via PasConsole.

PasConsole Information

PasConsole allows you to perform many simple I/O routines similar to the routines you might use on another computer. In fact, PasConsole allows you to set up your Macintosh screen to show 80 characters horizontally and 25 lines vertically to simulate even the simplest of output devices.

A simple call to the routine ClearScreen will erase the entire Macintosh display so that you can start with a clean slate. In addition, the routine GoToXY, which takes horizontal and vertical position parameters, will allow you to place the cursor anywhere on the screen so that you can start I/O from there. We'll be working with many programs that use these very simple and effective I/O routines, so you will become quite familiar with them.

PasConsole also provides a couple of useful input-specific routines: KeyPressed and ReadChar. ReadChar may be used to read in a single character of input like this:

```
aChar := ReadChar;
```

where aChar is defined to be a char variable. This sort of input is quite useful for input on menu selections. For example, if you display five options and ask the user to select one, he need only press the number of the option; there is no need to press the enter key. As soon as the Macintosh detects a key depression it sends it back to your program and you can continue with the next Pascal statement. On the other hand, KeyPressed is a Boolean function that returns the value TRUE when a key has been pressed. With this function,

you can set up a continuous loop that doesn't stop until the user presses a key like this:

```
while NOT ( KeyPressed ) do;
```

You will see extensive use of these routines, so it is necessary that you understand how they work. Now take a look at how to use Unit Mover, which allows you to modify existing units.

Using UnitMover

UnitMover, which appears on your Utilities disk, allows you to move units from one module to another or to delete them entirely. Before I discuss the workings of UnitMover, you must understand that the units listed in your Turbo Pascal manual are part of the Turbo program. That is, the modules that make up PasConsole, QuickDraw, and the rest are in the Turbo program that runs the compiler. That is why the Turbo program is so large; if you remove any units you won't need for a while—from your working disk, *not* the original Turbo disk—you can save a lot of RAM space. As of release 1.1 of the Turbo compiler, UnitMover could not copy a unit to a nonexistent file in order to create a new unit. The folks at Borland have told me that this option may be in a future release of UnitMover, but for now all you can do is remove units or copy them from one unit file to another.

When you invoke UnitMover, the screen shown in Figure 6.4 is displayed.

Click on the open button to see what files are available via UnitMover. The dialog box shown in Figure 6.5 is displayed when you click on the open button.

Select a unit. For example, I decide to remove some of the units I won't need. Look at how to remove MacPrint and FixMath.

Once I have selected the unit, I use the scroll bar to select the units to be removed. I scroll to MacPrint and FixMath, drag over them, and release the mouse button. Information about these two units is displayed at the bottom of the UnitMover window as shown in Figure 6.6.

This additional window area shows in bytes the number and size of the units you have selected. Click on the remove button, and the units are deleted from Turbo. Alternatively, you may open another unit in the top right window in UnitMover to specify the output unit file. This activates the copy button so that you can copy these units to the output unit file. Again, you should be working with UnitMover only on the copies of your master Turbo Pascal disks. If you remove a unit from your master disk and don't have a backup, it is gone for good!

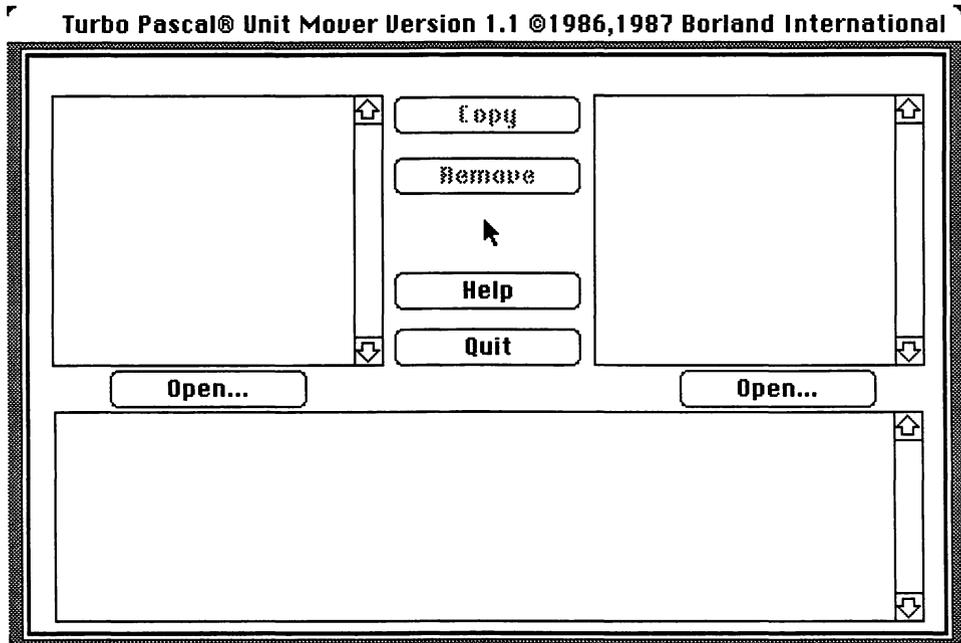


Fig. 6.4.

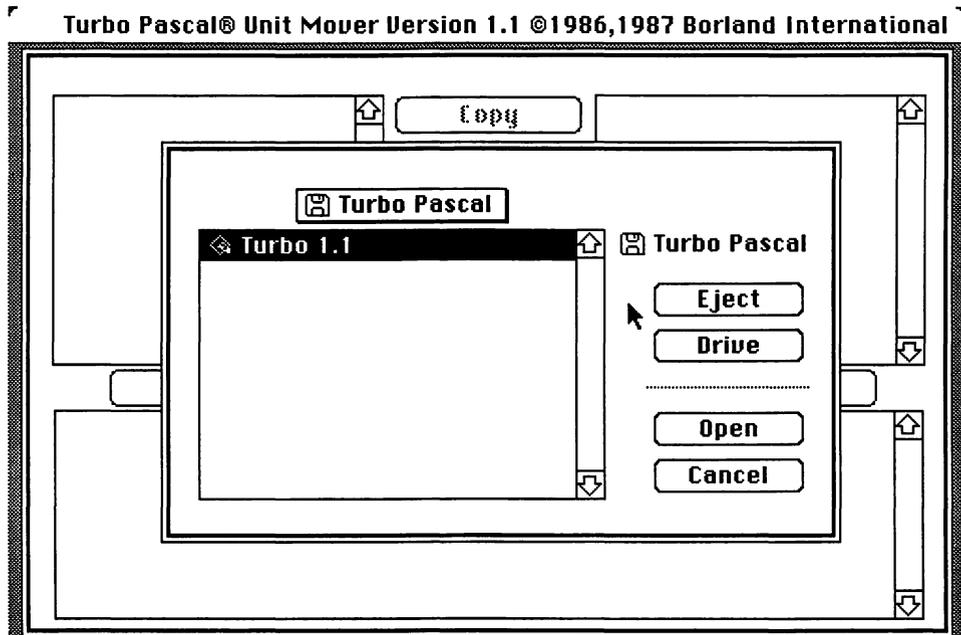


Fig. 6.5.

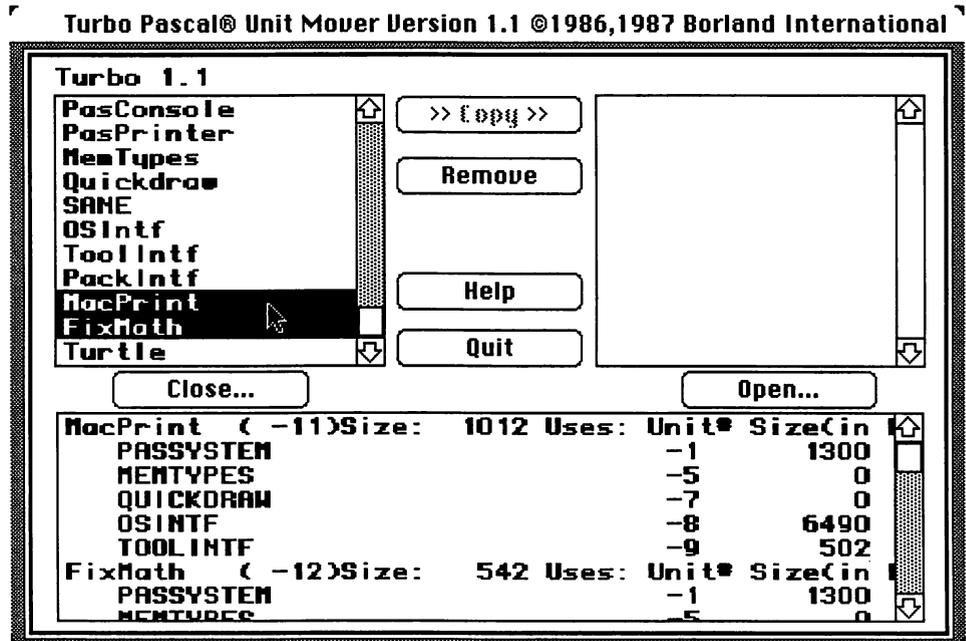


Fig. 6.6.

You have completed a very important chapter on Pascal programming. You learned how to define and to write procedures and functions. You now know the difference between a global variable and a local variable. You learned how to pass information with value and address parameters that use pointers. You have also learned about some of the more important and useful compiler directives as well as had an introduction to the world of units and UnitMover. Take a few minutes, give yourself a pat on the back, and review what you have learned. You'll discover you can write some interesting programs with what you have learned so far.

Review Summary

1. Structured programming is accomplished by breaking large programs down into smaller and smaller subprograms, or modules. The modules are divided until each submodule consists of a manageable block of statements, each performing a specific task.
2. Top-down design, sometimes called stepwise refinement, is a structured program design in which the original macro problem is represented by a main module. The main module is broken down into a set of specific modules that solve specific tasks. The final program con-

sists of a collection of submodules and appears as a hierarchical tree whose main module can be depicted at level 0, the next set of modules at level 1, and so on.

3. A procedure identifies a module or subprogram. It consists of a collection of statements that resembles a full-blown program. To access a procedure, you call it by name from some point in another module.
4. A procedure consists of a heading and a body and is analogous in appearance to a standard program. In the procedure heading the reserved word **procedure** replaces the reserved word **program**. The last **end** of the procedure body is followed by a semicolon instead of a period.
5. A procedure is placed immediately following the **var** section of the main program.
6. Variables can be used with procedures and are either global or local. A global variable has meaning throughout the program, while a local variable has meaning only while the procedure is executing.
7. A parameter can be used for passing information between a procedure and the main program. A parameter can be either variable or value. A variable parameter passes information by reference and acts as a location pointer in memory. When information is passed by a value parameter, only a copy of the value is passed.
8. In a procedure definition statement the reserved word **var** is used for variable parameters, while no modifier is used with value parameters.
9. Functions, like procedures, have a heading and a body. However, a function by definition returns a single computed value to the module that called it.
10. Functions, like procedures, may include parameter lists with either value or variable parameters.
11. Turbo Pascal offers several compiler directives set up via a comment line with a dollar sign—as the first character inside the comment. These directives may set up a flag to perform compilation in a certain manner or perform some other request, such as calling in a Pascal include file.
12. Turbo Pascal offers the concept of units to aid in the development of structured programming. Units are defined with a special heading statement and consist of the sections interface and implementation. Interface is the section visible to calling routines. It contains the declarations for special constants, types, vars, procedures, and functions. Implementation is the portion of the unit that contains the code for the internally represented routines declared in the interface section.
13. Turbo Pascal offers several standard units, including PasConsole,

which allows you to perform I/O via the Macintosh screen as though it were a standard 80-by-25 display.

14. UnitMover allows you to remove or copy units from one file to another.

Quiz

1. What is the difference between a procedure and a function?
2. What is the difference between a global and a local variable?
3. What is the difference between passing a parameter by value and passing by var?
4. What do `{I+}` and `{I-}` allow you to do?
5. What is a unit?
6. What are the two sections in a unit and how do they differ?
7. What does UnitMover allow you to do?

Turbo Pascal Library Features

How to Avoid Reinventing the Wheel
The Central Library
Library Routines
Review Summary
Quiz

In this chapter you will learn:

- What the library is.
- How to use library routines for mathematics, strings, bit manipulation, and so forth.

How to Avoid Reinventing the Wheel

When you begin writing programs, you soon realize that it would be nice if the computer could magically perform some common tasks you use in different programs. Suppose, for some reason you need to check and see whether an integer variable is odd or even. You might develop a function like this:

```
Function Odd ( number : integer ) : boolean;
```

```
begin
  if ( ( Number MOD 2 ) = 0 ) then
    Odd := FALSE
  else
    Odd := TRUE
end;      {function Odd}
```

Any time you invoke this function it returns FALSE if the number is even or TRUE if the number is odd. Assuming inNumber is an integer, you can refer to this function in your main program like this:

```
readln ( inNumber );
if ( Odd ( inNumber ) ) then
begin
  {logic for Odd returning TRUE}
end
else
begin
  {logic for Odd returning FALSE}
end;
```

It may have taken you only a few minutes to develop a function like `Odd`, but if you already had one, it would be a waste of time. There is a function very similar to our `Odd`, in the Turbo Pascal library, but before I discuss it and several other library routines, take a closer look at what is meant by library.

The Central Library

If you're an avid book reader, you know your local library has a wealth of books on topics from A to Z, you know that by taking advantage of its offerings, you may save hundreds of dollars by not buying them.

The standard Turbo Pascal library is very similar to your local library in that it holds a wealth of procedures and functions for everything from mathematics to window manipulation. Because of this the Pascal library can save you time, just as your library can save you money, since you don't have to write these routines yourself. All you need to do is invoke them and follow a couple of simple rules:

1. If the library routine is a function, make sure that the variable you assign it to is the same type as the function itself.
2. If the routine passes parameters, make certain that the order of parameters is identical to that of the routine and that their types match.

This is very similar to the discussion of units in Chapter 6, but these library routines are not part of a unit. Rather, they are standard routines available in almost every edition of Pascal on any type of computer.

Once you understand a routine, all you need to do is call it up as if it were your own; there is no need to worry about scope, since library routines are treated as global declarations. The next several sections of this chapter present some of the more popular routines with a discussion and example of each.

Library Routines

Mathematical Routines

1. *Squaring*

This function may be used to determine the square of a numeric value— $2^2 = 4$ —with the format

Sqr (Number)

where Number and the resulting value types are always identical. For example, if an integer is passed to Sqr, an integer is returned. An example of this function is

```
writeln (Sqr ( 12 ) );
```

which displays the value 144.

2. *The Square Root*

This function may be used to determine the square root of a real expression, for example, the square root of 4 is 2, with the following format:

Sqrt (Number)

where Number is a real expression and the function returns a real value.

An example of this function is

```
writeln ( Sqrt ( 144.0 ) );
```

which displays the value 12.

3. *Absolute Value*

This function may be used to determine the absolute value of a given number—the absolute value of 4 is 4; the absolute value of -7 is 7—with the format

Abs (Number)

where Number is a numeric parameter and the value returned by the function is the same type as Number.

An example of this function is

```
writeln ( Abs ( -456 ), Abs ( 456 ) );
```

which displays 456 twice.

4. *Odd*

This is the library function I discussed earlier in this chapter. It determines whether a given integer is even or odd and has the format

Odd (Number)

where Number is an integer and the function returns a Boolean value, TRUE for odd, FALSE for even.

An example of this function is

```
bool1 := Odd ( 3 );  
bool2 := Odd ( 4 );
```

where bool1 and bool2 are declared as type Boolean. Bool1 will be assigned the value TRUE and Bool2 will be FALSE.

5. *Truncation*

This function may be used to strip the fraction off a real number so that only the whole number remains; after a truncation 5.7 is 5 and 3.6 is 3. The format is

Trunc (Number)

where Number is Real and the function returns a value of type long integer.

An example of this function is

```
writeln ( Trunc ( 3.1 ), Trunc ( 3.9 ) );
```

which displays two 3s.

6. *Rounding*

As opposed to truncation, this function does not merely strip off the fractional portion of a number; it rounds the number to the nearest integer value; after rounding 4.6 reads 5, whereas 4.4 reads 4. The function has the format

Round (Number)

where *Number* is a real value and the function returns a value of type long integer.

An example of this function is

```
writeln ( Round ( 3.1 ), Round ( 3.9 ) );
```

which displays 3 and 4 respectively.

7. *Int*

Like *Trunc*, this function may be used to strip off the fractional portion of a real number so that only the whole number remains; after *int*, 5.7 reads 5.0 and 3.6 reads 3.0. Notice that the result returned is a Real as opposed to a long integer as with *Trunc*. The format is

Int (Number)

where *Number* is a real and the function returns a real value as noted above.

An example of this function is

```
writeln ( Int ( 3.1 ), Int ( 3.9 ) );
```

which results in 3.0 being displayed twice.

8. *The Cosine*

This trigonometric function returns the cosine of an angle expressed in radians with the format

Cos (Angle)

where *angle* is a Real type and the function returns a Real value.

An example of this function is

```
writeln ( Cos ( 0.0 ), Cos ( 22.0 / 7.0 ) );
```

which results in 1 and -1 being displayed. (Note: 22/7 approximates the mathematical constant pi).

9. *The Sine*

This trigonometric function returns the sine of an angle expressed in radians with the following format:

Sin (Angle)

where *Angle* is a real type and the function returns a real value.

An example of this function is

```
writeln( Sin ( 0.0 ), Sin ( 22.0 / 14.0 ) );
```

which results in a 0 and a 1 being displayed.

10. Exp

This function may be used to determine the value of the natural number e raised to a power. The syntax is

Exp (Number)

where Number and the value returned are both real types.

An example of this function is

```
writeln ( Exp ( 1.0 ) );
```

which displays a 2.7, or approximately the value of e .

11. ln

ln, or natural log may be used to return the natural logarithm of a specified value. It has the syntax

ln (Number)

where Number and the value returned again are real types.

An example of this function is

```
writeln ( ln ( 5.0 ) );
```

which displays the value 1.6.

12. ArcTan

This function may be used to determine the arctangent of an angle specified in radians. Its syntax is

ArcTan (Number)

where Number and the returned value are both real types.

An example of this function is

```
writeln ( ArcTan ( 0.0 ) );
```

where the value 0.0 is displayed.

String Routines

1. *Length*

This function may be used to determine the length of a string variable. Its format is

Length (String)

The value returned by the function is an integer type.

An example of this function is

```
reply := 'Hi Joe';  
writeln ( reply, ' ', Length ( Reply ) );
```

which results in the following output:

```
Hi Joe      6
```

since the string "Hi Joe" is 6 characters long.

2. *Concat*

This function concatenates, or joins, two or more strings into one. The function has the format

Concat (Strng1, Strng2, ...)

where Strng1 and Strng2 are string type and the value returned by the function is also a string. As shown by the elipsis, you can have more than two strings in the parameter list.

An example of the function is

```
st1 := 'Hi';  
st2 := 'there';
```

```
st3 := 'Kelly';  
writeln ( Concat ( st1, st2, st3 ) );
```

which would display:

```
Hi there Kelly
```

Note the blanks at the end of the “Hi” and “there” strings when they are assigned to St1 and St2; this is necessary, since Concat does not insert spaces between strings. Without them this would be displayed:

```
HithereKelly
```

3. Copy

This function may be used to copy a portion of a string into another string. Its format is

Copy (Stng, Start, Count)

where Stng is the string from which to copy, Start is the index of the character you wish to start at, and Count is the number of characters including Start you wish to copy. Start and Count are both integers and the value returned by the function is a string.

An example of this function is

```
st1 := 'Oliver';  
st2 := Copy ( st1, 2, 4 );  
writeln ( St2 );
```

which displays

```
live
```

The function does not change the value of the string parameter (st1); it copies Count—4—characters starting at and including the Start—2nd—character.

4. Delete

This is the first procedure discussed in this chapter; it may be used to remove a portion of a string. You should note that the string parameter is actually modified to contain the new value. The format is

Delete (Stng, Start, Count);

where Count characters beginning at Start are deleted from Stng and this new value is placed in Stng.

An example of this procedure is

```
st1 := 'playground';  
Delete ( st1, 5, 6 );  
writeln ( st1 );
```

where this is displayed:

```
play
```

5. Insert

This procedure may be used to insert a string into another string. As with Delete, the string is modified to hold the new value. The procedure has the syntax

Insert (Stng1, Stng2, Start)

where Stng1 is inserted into Stng2 beginning at the Start character of St2.

An example of this procedure is

```
st1 := 'ground';  
st2 := 'plays';  
Insert ( st1, st2, 5 );  
writeln ( st2 );
```

which displays

```
playgrounds
```

6. Position

This function determines where a particular substring is located within another string.

The function's format is

Pos (Stng1, Stng2)

where Stng1 is the substring you are searching for in Stng2. The value returned by the function is an Integer type and specifies the index within Stng2 where Stng1 begins.

An example of this function is

```
st1 := 'ground';  
st2 := 'playground';  
location := Pos ( st1, st2 ); {Location is of type integer}  
writeln ( location );
```

where the following is displayed:

5

If the substring cannot be found, the value 0 is returned.

Other Types of Routines

1. *Ord*

This function determines the ordinal value of an ordinal item. This function should be reviewed after you finish reading Chapter 9, which further discusses ordinal types. Its format is

Ord (Item)

For example, with the declarations

```
Type  
FruitType = (Orange, Apple, Banana);
```

```
Var  
snack : FruitType;
```

you could make the assignment

```
snack := Orange;
```

and display

```
writeln ( Ord ( snack ) );
```

which results in

0

displayed on the screen. Note: When dealing with ordinal types, Pascal numbers the first element 0, the next element 1, and so on.

In the example above, the value returned is an Integer type. However, we can derive the Ord of a pointer and the value returned is a long integer. Another function, Ord4, returns a long integer regardless of whether the parameter is ordinal or a pointer. Again, this function should be reviewed after Chapter 9.

2. *Chr*

This function converts an integer into its corresponding value in the Macintosh Character Set.

The function has the format

Chr (Number)

where Number is some integer between 0 and 255.

An example of this function is

```
writeln ( Chr ( 100 ) );
```

which displays a lowercase d, the 100th character in the Macintosh Character Set.

3. *Pred*

This function seeks out the preceding ordinal item. It has the following format:

Pred (Item)

Given the declarations

Type

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

Var

```
day, yesterday : Week;
```

and this assignments:

```
day := Tuesday;  
yesterday := Pred ( day );
```

the value Monday is assigned to “yesterday.” The value returned by the function is one of the items specified in the type declaration. The Pred of any day as I have them set up in the Week type is the day before, with the exception of Sunday. Don’t expect Pascal to know that the day preceding Sunday is Saturday. Warning: be very careful never to do a Pred on the first item in the type; a safe way to avoid doing this is to check to make sure that the item does not have an Ord equal to 0.

4. *Succ*

This function is the complement to Pred; with it you can determine the succeeding item of a particular type. The function has the following format:

Succ (Item)

If we look at the Type declaration for Week and the Var declaration for day and tomorrow, the assignments

```
day := Wednesday;  
tomorrow := Succ ( day );
```

result in tomorrow being assigned the value Thursday. As with Pred, don’t expect Pascal to know the Succ of Saturday. A safe way around this problem is to know how many items there are in the type and make sure that the Ord is less than that.

5. *SizeOf*

This function determines the size of either a variable or a type identifier. SizeOf has the format

SizeOf (Identifier)

where the result returned is an integer. The declaration

```
Var  
  aNumber : integer;
```

plus the statement

```
writeln ( SizeOf ( aNumber ), SizeOf ( integer ) );
```

writes 2 on the screen twice.

6. *Hi and HiWord*

This function retrieves the high-order byte of an integer. It has the syntax

Hi (Number)

where the value returned is a 1-byte signed integer. If the integer variable `anInt` contains the value 256—or \$0100—the statement

```
writeln ( Hi ( anInt ) );
```

displays the value 1, since the high-order word for 256 is \$01. `HiWord` works identically to `Hi` except that it uses a long integer and returns an integer type.

7. *Lo and LoWord*

As compared with `Hi`, `Lo` returns the low-order byte of an integer with the following syntax:

Lo (Number)

where `Lo` is an integer. In the example above, `AnInt` contained the value 256 (\$0100). The statement

```
writeln ( Lo ( anInt ) );
```

displays a 0, since the low-order byte of 256 is \$00. `LoWord` works identically to `Lo` except that it uses a long integer and returns an integer.

8. *Swap and SwapWord*

This function swaps the high- and low-order bytes of an integer with the following syntax:

Swap (Number)

If `AnInt` contains the value 256, the statement

```
writeln ( Swap ( anInt ) );
```

displays the value 1 (\$0001). The parameter `Number` and the returned value are both Integer types. `SwapWord` works identically to `Swap` except that it uses a long integer parameter and returns a long integer type.

Review Summary

1. A library is a central gathering of commonly used routines; the Pascal library has routines for mathematics, string manipulation, bit manipulation, and so on.
2. Using the routines in the standard Pascal library as well as any other library saves time.

Quiz

1. What are the results of the following functions?

- A. Sqr(10)
- B. Sqrt(25.0)
- C. Abs(12)
- D. Odd(900)
- E. Trunc(1.99)
- F. Round(1.99)

2. What would the following lines of Pascal display on the screen?

```
st1 := 'I like Pascal';
st2 := 'The funniest thing happened to me ...';
st3 := 'How is your dog?';
st4 := Copy ( st1, 8, 6 );
st5 := Copy ( st2, 5, 3 );
st6 := Copy ( st3, 5, 2 );
writeln ( Concat ( st4, ', ', st6, ', ', st5 );
```

3. Given the declarations

```
Type
  Week = (Sunday, Monday, Tuesday, Wednesday,
          Thursday, Friday, Saturday);
Var
  aDay, anotherDay : Week;
```

what is the final value of ADay in this statement?

```
anotherDay := Wednesday;
aDay := Pred ( Succ ( Suc ( Pred ( anotherDay ) ) ) );
```



APPLICATIONS AND ADVANCED CONCEPTS

8

Programmer's Corner

The Entire Picture: Complete Pascal Programs
Metric Conversion Program
Guess-a-Number Program
Decimal-to-Hexadecimal Conversion Program
Tape Counter Program
Review Summary
Quiz

In this chapter you will learn:

- How complete Pascal programs operate on the Macintosh.
- How to modify existing programs to suit your needs.
- Why comments are a critical part of programming.
- The importance of proper error handling.
- Four programs for figuring how many songs will fit on a tape, doing metric conversions, guessing numbers, and converting to hexadecimal notation. All are documented with line-by-line, variable, and type explanations.

The Entire Picture: Complete Pascal Programs

Now that you have learned all the individual elements of Pascal programming, it is time to put everything together and discuss a few application programs. These programs are designed to tie together all the topics I have discussed as well as to prepare you for further study of file handling, graphics and sound, and pointers and linked lists. Take a look at a program that performs conversions to the metric system.

Metric Conversion Program

Enter the following program into your Macintosh, paying close attention to spacing within quotes ' ' and the use of semicolons:

Program Metric;

```
(*****
(* This interactive program will calculate various *)
(* metric conversions for you. *)
(*****)
```

Var

```
done : boolean;           {are we finished yet?}
valid : boolean;         {is this valid input?}
choice : char;           {user's choice from menu}
```

Procedure InchToMeter;

Const

```
MeterConvConst = 39.37;           {conversion constant of inch to meter}
```

Var

```
numOfInches : real;           {how many inches to express in meters}
numOfMeters : real;           {how many meters}
```

begin {procedure InchToMeter}

```
ClearScreen;
writeln;
write ('How many inches? ');
readln (numOfInches);
numOfMeters := numOfInches/MeterConvConst;
writeln;
writeln (numOfInches : 8 : 2, 'inches equals', numOfMeters : 8 : 2,
'meters.');
```

```
writeln;
write ('Press any key to continue. . ');
choice := ReadChar;
end; {procedure InchToMeter}
```

Procedure CubInchToLiter;

Const

```
LiterConvConst = 61.02;           {conversion constant of cub. inch to liter}
```

Var

```
numOfCubInches : real;           {how many cubic inches}
numOfLiters : real;           {how many liters}
```

```
begin                                     {procedure CubInchToLiter}
  ClearScreen;
  writeln;
  write ('How many cubic inches? ');
  readln (numOfCubInches);
  numOfLiters := numOfCubInches / LiterConvConst;
  writeln;
  writeln (numOfCubInches : 8 : 2, 'cubic inches equals',
    numOfLiters : 8 : 2, 'liters. ');
  writeln;
  write ('Press any key to continue. . ');
  choice := ReadChar;
end;                                     {procedure CubInchToLiter}

Procedure OunceToGram;

Const
  GranConvConst = 0.035;                {conversion constant}

Var
  numOfOunces : real;                   {how many ounces}
  numOfGrams  : real;                   {how many grams}

begin                                     {procedure OunceToGram}
  ClearScreen;
  writeln;
  write ('How many ounces? ');
  readln (numOfOunces);
  numOfGrams := numOfOunces / GramConvConst;
  writeln;
  writeln (numOfOunces : 8 : 2, 'ounces equals', numOfGrams : 8 : 2,
    'grams. ');
  writeln;
  write ('Press any key to continue. . ');
  choice := ReadChar;
end;                                     {procedure OunceToGram}

begin                                     {program Metric}
  done := FALSE;
  while NOT done do
  begin
    valid := FALSE;
    while NOT valid do
```

```
begin
  ClearScreen;
  writeln ('Which do you wish to do:');
  writeln;
  writeln ('  1. Convert from inches to meters');
  writeln ('  2. Convert from cubic inches to liters');
  writeln ('  3. Convert from ounces to grams');
  writeln ('  4. QUIT');
  writeln;
  write ('PLEASE ENTER 1,2,3 OR 4 ');
  choice := ReadChar;
  case choice of
    '1' : begin
      InchToMeter;
      valid := TRUE;
      end;

    '2' : begin
      CubInchToLiter;
      valid := TRUE;
      end;

    '3' : begin
      OunceToGram;
      valid := TRUE;
      end;

    '4' : begin
      valid := TRUE;
      done := TRUE;
      end;

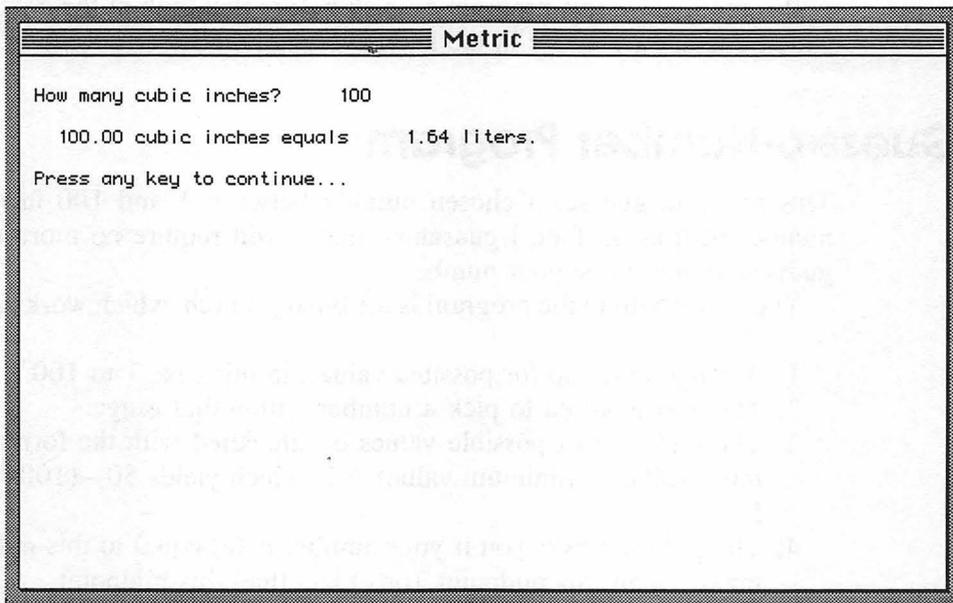
    otherwise
      begin
        writeln;
        writeln ('INVALID RESPONSE . . . PLEASE TRY AGAIN!');
        writeln;
        end;
      end;
      end;
      end;
      end;
      end.
      {case choice of}
      {while NOT valid do}
      {while NOT done do}
      {program metric}
```

Figure 8.1 shows a sample run of this program, which allows you to convert inches to meters, cubic inches to liters, and ounces to grams. You may continue to make conversions without having to rerun the program, and the modular conversion procedures let you easily add more options.

Program Explanation

I begin, of course, with the program name, `Metric`. The global variables are declared, and immediately thereafter come the blocks containing the three conversion procedures `InchToMeter`, `CubInchToLiter`, and `OunceToGram`. Finally, as the rules of Pascal dictate, the last block of code is the main program logic. Now let's look at each block in detail in order to determine exactly how they fit together.

The main program block begins by setting the Boolean `done` to `FALSE`, and the main loop starts with `while NOT done`. Then the Boolean `valid` is initialized to `FALSE` and a nested loop, `while NOT valid`, is started. This loop will be directly affected by valid input by the user. `PasConsole` procedure `ClearScreen` is invoked and a menu shows the user four options. As shown



```
Metric
How many cubic inches?    100
    100.00 cubic inches equals    1.64 liters.
Press any key to continue...
```

Fig. 8.1.

in the successive `writeln` statements, the options include each of the three conversions and the option to `QUIT` the program. `ReadChar` (`PasConsole`) is executed, and the menu choice is used in a case statement to determine the appropriate action. As the case statement reads, if the user entered a 1, 2, or 3, the appropriate procedure is invoked and `valid` is set to `TRUE` because the user's response was acceptable. If a 4 was entered, the Boolean variables `valid` and `done` are set to `TRUE` because the user wishes to `QUIT` and the response was acceptable. The `OTHERWISE` clause of the case statement displays an error message, and no Boolean flags are changed.

When you look at the two previous `while` loops, it should be obvious that the outer one—`while NOT done`—terminates only when a 4 is entered, whereas the inner one—`while NOT valid`—terminates when any value from 1 to 4 is entered.

The three procedures are quite similar in that they all request an amount—`inches`, `cubic inches`, or `ounces`—and then convert that amount into its equivalent in the metric system—`meters`, `liters`, or `grams`—through a conversion constant—`MeterConvConst`, `LiterConvConst`, or `GramConvConst`. You may also have noticed that each procedure uses the formatted `writeln` statement, for example `numOfInches:8:2`, where 8 specifies the minimum number of characters to be printed and 2 specifies how many digits will be displayed to the right of the decimal point.

Be sure to save this program on disk before you look at the next program, which guesses a number in the least average number of attempts.

Guess-a-Number Program

This program guesses a chosen number between 1 and 100 in the lowest number of tries. In fact, I guarantee that it will require no more than eight guesses to determine your number.

The secret behind the program is the binary search, which works as follows:

1. A range is set up for possible values, in this case 1 to 100.
2. The user is asked to pick a number within that range.
3. The midpoint of possible values is calculated with the formula (maximum value + minimum value) \div 2, which yields 50— $(100 + 1) \div 2 = 50$.
4. The program asks you if your number is (a) equal to this midpoint, (b) greater than this midpoint, (or c) less than this midpoint.
5. If it is equal to your number, the program stops. If your number is greater, the values 1 through 50 are thrown out and we make the

midpoint, or first guess, the lowest possible value. Actually, midpoint + 1 is the lowest possible value, but integer truncation lets us ignore +1. We go back to step 3. If your number is less than the guess, the top half of possible values are thrown out and the midpoint becomes the highest possible value. We go back to step 3.

This loop from step 5 to step 3 will continue until your number has correctly been identified, and the given range of 1 to 100 will take no more than seven attempts. Seven is not a magic number that holds true for all ranges; rather, this figure is the lowest power of 2 that exceeds the range size. Take a moment to review the following table of powers of two:

Power	Expression	Equivalent
0	2^0	1
1	2^1	2
2	2^2	4
3	2^3	8
4	2^4	16
5	2^5	32
6	2^6	64
7	2^7	128
8	2^8	256
9	2^9	512
10	2^{10}	1024

The lowest power of 2 that results in a figure equal to or greater than the range size 100 is 7. Therefore, it will take at most seven guesses before the number is correctly picked. Two is the base number because the list of possibilities halves with each guess.

Now that you understand how the program works, enter it into your Macintosh as it appears below:

Program FindNum;

```
(*****
(* This program shows how an interactive program can be written *)
(* to guess a number via a binary search.                          *)
(*****)
```

Const

```
    MaxGuess = 7;                                {Maximum guesses the program will
                                                make}
```

Var

```

goAhead : char;           {Input character from user}
found    : boolean;      {Has the program found your number?}
guess    : integer;      {The program's actual guess}
done     : boolean;      {Are you finished with the game?}
maxNum   : integer;      {The current maximum range of guesses}
minNum   : integer;      {The current minimum range of guesses}
count    : integer;      {The number of guesses the program has
                           made}

valid    : boolean;      {Did you provide a valid response?}

```

begin

```

done := FALSE;
while NOT done do
begin
  ClearScreen;
  minNum := 1;           {We will set our range from 1 to 100}
  maxNum := 100;
  writeln ('Pick a number between 1 and 100 . . .');
  writeln ('Press any key to continue . . .');
  goAhead := ReadChar;
  found := FALSE;
  guess := (maxNum + minNum) DIV 2;
  count := 1;
  while (NOT found) AND (count <= MaxGuess) do
  begin
    writeln;
    writeln ('Is your number less than (L), greater than (G),');
    writeln ('or equal to (E)', Guess : 3, '?');
    goAhead := ReadChar;
    case (goAhead) of
      'E', 'e':
        found := TRUE;
      'L', 'l':
        begin
          maxNum := guess;
          guess := (maxNum + minNum) DIV 2;
          count := count + 1;
        end;
      'G', 'g':
        begin

```


Program Explanation

This program contains no procedures or functions and is therefore easily read from top to bottom. After the constant and variable declarations the first thing to do is set the Boolean `done` to `FALSE`. I then begin the main loop, while `NOT done`, which will continue until the user decides to stop playing. I do a `ClearScreen` and assign values to the `minNum` (1) and `maxNum` (100). Through `writeln` and `ReadChar` I ask the user to pick a number between 1 and 100 and wait for any key to be pressed to continue. I set `found` to `FALSE` and use the formula for finding the midpoint between two numbers to determine the first guess; the formula adds `maxNum` and `minNum` and divides the sum by 2 to arrive at the midpoint of the two numbers. The integer variable `count` is set to 1 and through the next while loop—while `(NOT found) AND (count <= maxGuess)`—is not permitted to exceed 7. Rather than saying

```
while (NOT found) AND (count <= 7) do
```

we chose to assign a constant (`MaxGuess`) to represent 7 for two reasons:

1. It makes the statement more readable.
2. It lends itself to a more obvious change should anyone choose to modify the program to allow the range of numbers to be 1 to 1000. Remember, from the power of 2 chart above, you have to make the maximum number of guesses equal to 10, and you may forget what the 7 represented, but with the constant name `MaxGuess` it should be more meaningful.

So this while loop executes until it either finds the user's number or makes the maximum number of guesses. After starting this loop the user is asked to say whether the number is less than, greater than, or equal to the calculated guess. A case is then performed on the response.

1. If the guess is equal to the number, `found` is set to `TRUE` so that the `while (NOT found) AND (count <= MaxGuess)` loop will terminate.
2. If the number is less than the guess, the top half of possible numbers is thrown away (by assigning `maxNum := guess`) and a new guess is calculated.
3. If the number is greater than the guess, the bottom half of possible numbers is thrown away (by assigning `minNum := guess`) and a new guess is calculated.

4. If an invalid response (the OTHERWISE clause) is entered, a message is displayed.

(Note that both uppercase and lowercase letters are used in the case statement so that the user can have the caps lock key down or up and the input will be valid.)

Next, when the program has either found the number or used the maximum number of guesses, if the number was guessed (meaning that `count <= maxGuess` is TRUE), a message says either that it took only `Count` guesses to determine it or that the maximum number of guesses has been made and since the number was not found, the user must have missed it. When either of these events is complete, the other nested while loop—while NOT valid, is executed. This loop is used to ask if the user would like to play again, and based upon the input, the flags `valid` (if a Y, y, N, or n is entered) and `done` (if an N or n is entered) are set to TRUE. As with other inputs of this nature, if an invalid response is entered (the OTHERWISE clause), an error message is displayed.

Decimal-to-Hexadecimal Conversion Program

The next program converts any decimal (base 10) integer into its hexadecimal (base 16) equivalent. If you are familiar with hexadecimal notation, enter the program and follow its instructions to perform conversions. For those who are unfamiliar with hexadecimal, or hex, numbers, there follows a brief explanation of their use and significance in the computer world.

Everyone uses the base 10 numbering system; the digits 0 through 9 represent any number. Look at the number 425. This number represents four hundreds, two tens, and five ones; the digit 4 is in the hundreds position, the digit 2 is in the tens position, and the digit 5 is in the ones position. Multiply and add these figures to get

$$\begin{array}{r} 4 \times 100 = 400 \\ 2 \times 10 = 20 \\ 5 \times 1 = \underline{5} \\ 425 \text{ total} \end{array}$$

Note that in base 10 each digit is multiplied by 10 raised to its position minus 1, or:

$$\text{decimal place value} = \text{digit} \times 10^{(\text{position} - 1)}$$

The second decimal place's value is

$$2 \times 10^{(2-1)} = 2 \times 10^1 = 2 \times 10 = 20$$

This discussion may seem trivial, but keep in mind that the same rules apply for hexadecimal numbers, except that they have 16 unique digits and each hexadecimal place in the number is multiplied by a power of 16. First of all, the 16 digits for hexadecimal notation are

Hex	Decimal Equivalent
\$0	0
\$1	1
\$2	2
\$3	3
\$4	4
\$5	5
\$6	6
\$7	7
\$8	8
\$9	9
\$A	10
\$B	11
\$C	12
\$D	13
\$E	14
\$F	15

The digits 0 through 9 are equivalent and the values 10 through 15 are represented by the first six letters in the alphabet. Hex numbers are usually preceded by a dollar sign to differentiate them from a base 10 value.

Now let's look at the hex number \$1A9. What is this in base 10? If we use our rules from above, we would multiply and add to get the following:

$$\begin{array}{r}
 1 \times 16^{(3-1)} = 1 \times 16^2 = 256 \\
 10 \times 16^{(2-1)} = 10 \times 16^1 = 160 \\
 9 \times 16^{(1-1)} = 9 \times 16^{(0)} = \underline{9} \\
 \hline
 425 \text{ total}
 \end{array}$$

The hex values are represented in boldface to show how they fit into the computations; compare them to the base 10 values from the earlier example to understand the similarity figuring the total. Also recall that \$A is equal to 10 in base 10.

Now that you know how to translate a hex number into base 10, let's see what's involved in converting from decimal to hex; remember to divide by 16 and write the remainder. For example, to convert 425 into hex, first divide it by 16:

$$425 \div 16 = 26 \text{ with a remainder of } 9$$

Divide the resulting dividend (26) by 16:

$$26 \div 16 = 1 \text{ with a remainder of } 10$$

Divide the resulting dividend (1) by 16:

$$1 \div 16 = 0 \text{ with a remainder of } 1$$

Since my dividend is 0, I stop. I have determined that 1, 10, and 9 are the remainders, and if I convert them to their respective hex equivalents, I find that the hex equivalent of 425 is \$1A9.

You now know how to go from decimal to hex and vice versa, but you may still be wondering how to use this knowledge. In computer applications, numbers are quite often expressed in systems other than base 10, and hexadecimal notation is one of the most popular of these systems. Therefore, it is worth your while to try a few conversions on your own and check them, either by converting back to the original system or by using the following program.

Program Conversion;

```
(*****)
(* This program converts a decimal number from 0 to*)
(* 32,767 to its hexadecimal equivalent.          *)
(*****)
```

Type

```
ConvType = String[6];
```

Var

```
done, valid : Boolean;           {Are we finished; Is this valid input?}
answer      : char;             {The user's response}
```

Procedure FlipFlop (Var inTemp : ConvType);

```

Var
  newString : ConvType;           {Temporary holder of flipped string}
  i          : integer;          {Loop counter}
begin                             {procedure FlipFlop}
  for i := 1 to 6 do
    newString[i] := inTemp[6 - i + 1];
  for i := 1 to 6 do
    inTemp[i] := NewString[i];
end;                               {procedure FlipFlop}

```

Procedure DecToHex;

```

Var
  inDecimal      : integer;       {the number entered by user}
  outHex         : ConvType;      {decimal input; hex output}
  thisLong       : 0..6;         {string length}
  i, m           : 0..5;         {loop counters}
  allNumeric     : boolean;      {is the input all numeric?}
  actualNumber,
  tempNumber     : integer;       {real number; temp. holder}
  nextHex        : char;         {actual hex value}

```

Function FigureHex (Var tempNumber : integer): char;

```

Var
  tempRmdr : 0..15;              {remainder after div 16}

begin                             {function FigureHex}
  tempRmdr := (tempNumber MOD 16);
  case (tempRmdr) of
    0: FigureHex := '0';

    1: FigureHex := '1';

    2: FigureHex := '2';

    3: FigureHex := '3';

    4: FigureHex := '4';

    5: FigureHex := '5';

```

```
6: FigureHex := '6';
7: FigureHex := '7';
8: FigureHex := '8';
9: FigureHex := '9';
10: FigureHex := 'A';
11: FigureHex := 'B';
12: FigureHex := 'C';
13: FigureHex := 'D';
14: FigureHex := 'E';
15: FigureHex := 'F';
end;                                {case (tempRmdr)}
tempNumber := (tempNumber DIV 16);
end;                                {function FigureHex}

begin                                {procedure DecToHex}
  valid := FALSE;
  while NOT valid do
    begin
      writeln;
      writeln ('Please enter a decimal integer in the range of 0 to 32767');
      write ('without commas or decimal points. ');
      readln (inDecimal);
      if (inDecimal >= 0) AND (inDecimal <= 32767) then
        valid := TRUE;
      if NOT valid then
        begin
          writeln;
          writeln ('INVALID ENTRY...PLEASE TRY AGAIN!');
          writeln;
        end;
    end;                                {if not Valid}
  end;                                {while not Valid}
  tempNumber := inDecimal;
  m := 1;
  outHex := '000000';
```


Program Explanation

This program is composed of the main program block, two procedures, and one function nested within a procedure. If you look down to the main program block, where the begin statement for the main program Conversion is located, you can see that we set done to FALSE and perform a while loop based on the value of done that will continue until the user wishes to quit the program. Immediately inside this while loop the program executes ClearScreen and then DecToHex, has a nested function called FigureHex. Starting at the begin statement for DecToHex, the Boolean variable valid is set to FALSE and a while loop is started based upon the value of valid. A message asks the user to enter a decimal integer from 0 to 32767, and this value is read into the variable InDecimal of type integer. A check is performed on InDecimal to determine if it is indeed within the range of 0 to 32767; if this is true, valid is set to TRUE. The if statement is used to display an invalid-entry message if an invalid number was entered. This signifies the end of the while NOT valid loop.

The statements `tempNumber := inDecimal; m := 1; and outHex := '000000';` are used to set local variables for the following reasons:

1. `tempNumber` now holds the value of `inDecimal`, so that I may manipulate (namely, repeatedly divide by 16 and determine the remainder) without destroying the original value, which still resides in `inDecimal`.
2. `m` is used as a counter to determine which digit the program is figuring; it starts with the first digit, so `m` is set to 1.
3. `outHex` is a string and will hold the final converted hex value.

A while loop based on `tempNumber` not equal to 0 now begins; it is directly related to dividing the decimal value by 16 and using its remainder until the quotient equals 0. This loop figures each hex digit and then increases `m` to prepare for the next digit. The nested function `FigureHex` is called for each iteration of the while loop; `tempNumber` is its only parameter.

At the `FigureHex` declaration the program assigns the remainder from dividing `tempNumber` by 16 to the local variable `tempRmdr`. A case statement performed on the value of `tempRmdr` will assign hex value to the function `FigureHex`. If the first remainder is 13, `FigureHex` will be set equal to D. Finally, `tempNumber` is replaced by its present value divided by 16. Note: This final statement is the reason `tempNumber` is listed as a var parameter. Otherwise I would not be able to change the value of `tempNumber`.

Having finished with the function `FigureHex`, return to the statement after it was invoked—but where was that? Remember? It was back in the `while (tempNumber) <> 0` loop in `DecToHex`. Again, this loop continues to determine the next digit for the hex conversion until `tempNumber` equals 0. So when this loop is complete, `outHex` will contain the converted hex value with one minor problem: it is backwards. If the converted number is \$3A7, it is stored in `outHex` as follows:

```
outHex[ 1 ] 7
outHex[ 2 ] A
outHex[ 3 ] 3
```

so if I try to write the line `(outHex[1], outHex[2], outHex[3])`; I will display \$7A3. The next procedure, `FlipFlop`, takes care of this problem; I pass it `outHex`, it will flip the characters to the proper sequence.

`FlipFlop`, the first procedure in the program, shows that for each of the elements of `outHex`—or `inTemp`, as it is called locally—the opposite element of `newString` is set equal to it. If you work through the loop, this is what is happening.

```
newString[ 1 ] := inTemp[ 6 ];
newString[ 2 ] := inTemp[ 5 ];
newString[ 3 ] := inTemp[ 4 ];
newString[ 4 ] := inTemp[ 3 ];
newString[ 5 ] := inTemp[ 2 ];
newString[ 6 ] := inTemp[ 1 ];
```

The value of `newString` is put into `inTemp` and `FlipFlop` returns control to `DecToHex`, where three output statements display the original base 10 value and its hex equivalent. `DecToHex` is complete and control returns to the main program.

The first statement after the call to `DecToHex` sets `valid` to `FALSE` and executes a `while` loop based upon the value of `valid`, that is, until the user enters a valid entry. The user is asked if he or she would like to run the program again and the input (`answer`) is used in a case statement to determine if the input is valid and the program done. Otherwise, in the event of an invalid response, a message is displayed.

Figure 8.2 shows how the conversion looks on your Macintosh.

Look at one more program, which allows you to figure out exactly how many songs you can fit on a recording tape.

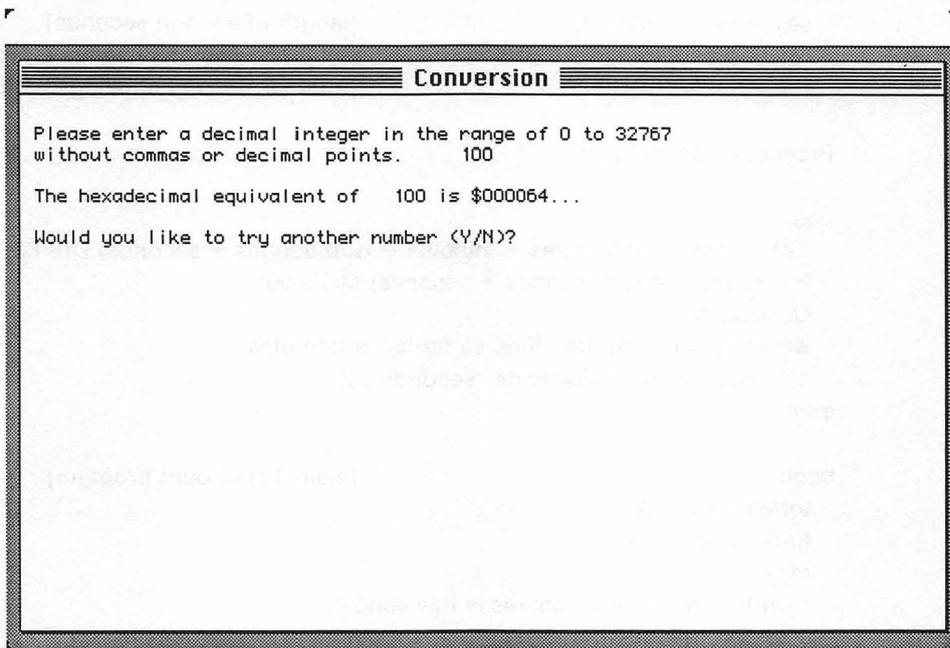


Fig. 8.2.

Tape Counter Program

If you have ever taped any of your phonograph albums, you are familiar with the problem of trying to figure out exactly how many songs you can get on one side of a tape. Many of us have spent some time adding minutes and seconds to get the most out of a blank tape. If you take a few moments to use the following program, your tapes will be a snap to make.

All you have to do is enter each song's length in minutes and seconds, and this short program will keep a running total of the amount of time necessary to make the recording.

Program TapeCount;

```

(*****
(* This short program allows the user to enter song lengths *)
(* in minutes and seconds to see exactly how many songs will *)
(* fit on a tape. *)
(*****

```

Var

```

minutes : integer; {length of song in minutes}

```

```

seconds   : integer;           {length of song in seconds}
totMinutes : integer;         {total minutes of all songs entered}
totSeconds : integer;         {total seconds of all songs entered}

Procedure CalcNDisp;

begin
  totMinutes := totMinutes + minutes + (totSeconds + seconds) DIV 60;
  totSeconds := (totSeconds + seconds) MOD 60;
  ClearScreen;
  writeln ('Total required time so far is:', totMinutes,
    'minutes and', totSeconds, 'seconds...');
end;

begin                               {main TapeCount program}
  totMinutes := 0;
  totSeconds := 0;
  repeat
    write ('How many minutes is this song?');
    readln (minutes);
    if (minutes <> 0) then
      begin
        write ('How many seconds is this song?');
        readln (seconds);
        CalcNDisp;
      end;
  until (Minutes = 0);                {zero entry for minutes means STOP}
end.                                  {program TapeCount}

```

Program Explanation

The tape counter program has one main repeat...until loop that continues until the user enters a 0 value for the number of minutes in a song. This loop requests the number of minutes and if minutes is not 0, the number of seconds. Then CalcNDisp is called to figure the running total of minutes and seconds thus far. TotMinutes is calculated by adding the previous totMinutes to the latest song's number of minutes. This amount is then added to the remainder after adding the totSeconds to the latest song's seconds and dividing that sum by 60 as follows:

```

totMinutes := totMinutes + minutes + (totSeconds + seconds)
  DIV 60;

```

TotSeconds is figured by using the remainder, or modulus, of adding the totSeconds to the latest song's seconds and dividing by 60. Again, the MOD operator is used to determine the remainder of dividing \times by y ($\times \text{MOD } y$).

The cumulative totals are displayed by CalcNDisp, and control returns to the main routine, where the repeat...until loop continues until a 0 minutes entry is made.

Review Summary

1. In a binary search the list of possible items is reduced by half with each successive guess.
2. Base 10 is the common numbering system using the 10 digits 0 through 9 to represent all possible values.
3. The hexadecimal system is the computer-oriented numbering system that uses the 16 digits 0 through F to represent all possible values.

Quiz

1. Why weren't minNum and maxNum (originally 1 and 100 respectively) declared as constants in the program Guess-a-Number?
2. In the decimal-to-hexadecimal conversion program, a portion of the code in the procedure DecToHex reads:

```
if ( inDecimal >= 0 ) AND ( inDecimal <= 32767 ) then
  valid := TRUE;
if NOT valid then
begin
  writeln;
  writeln ( 'INVALID ENTRY...PLEASE TRY AGAIN!' );
  writeln;
end;
```

How could this code have been simplified?

3. How could the procedure FlipFlop (in the decimal-to-hex conversion program) have been omitted?
4. Do any numbers have the same representation in both hexadecimal and base 10 notation?

Advanced Data Structures

Advanced Numeric Types: Long Integers and Extended Real Numbers

What's in a Type

Simple Arrays

Parallel Arrays

Records and Files

Sets

Using Arrays, Records, and Sets in an Application

Review Summary

Quiz

In this chapter you will learn:

- More about data types and how to construct your own.
- The concept of arrays and how they are used in programs.
- About record and file types.
- About sets and their value and readability in Pascal.

Advanced Numeric Types: Long Integers and Extended Real Numbers

The range of integers acceptable to Turbo Pascal is -32768 to 32767 . You may sometimes need an integer value that exceeds this range. To this end Turbo Pascal provides the data type long integer, or `LongInt`. The range for long integers -2147483648 to 2147483647 , which should accommodate most of your needs. A long integer is declared similarly to other data types:

```
var  
    number : LongInt;
```

The variable `number` is declared as a long integer. The values of a single integer and a long integer can be mixed when performing arithmetic; however,

you should use long integers only when it is absolutely necessary because they require twice as much storage space (32 bits versus 16 bits) as do integers.

I have presented some simple mathematics that your Macintosh can perform for you. For a greater degree of accuracy in complex arithmetic, you may want to use the two additional real data types, double and extended integers. The differences between the three are the number of significant digits and the range of numbers allowed. For example, a standard real variable provides an accuracy of 7 to 8 decimal places and can be stored in a range of 1.5×10^{-45} to 3.4×10^{38} , while a Double real variable provides accuracy of 15 to 16 decimal places with a range of 5.0×10^{-324} to 1.7×10^{308} . Similarly, the Extended type provides accuracy of 19 to 20 decimal places and a range of values from 1.9×10^{-4951} to 1.1×10^{4932} . Indeed, Double and Extended data can be used to provide a much greater degree of accuracy than a standard real type. The drawback is that as with long integers, the larger the range of values, the more memory must be set aside. The following table shows the number of bytes occupied by each of these data types:

<u>Type</u>	<u>Size in bytes</u>
Real Single	4
Double	8
Extended	10

Note that all three are compatible and can be mixed in mathematical computations. In fact, all real numbers are converted to extended and then back to the type needed. If you try to calculate a value that falls outside the range of the data type being used, an error will occur.

Before leaving my discussion of real numbers, I must point out that a fourth type computes numbers without decimal points. The Comp (for computational) data type allows calculations of exact integer values without any rounding errors. Type Comp accepts values in the range of -9.2×10^{18} to 9.2×10^{18} .

When using real number data types, be as prudent as possible. What you save in memory space, you may lose in speed. Remember, any real-number data are converted to extended, calculated, and then converted back to the original type. If you always use Extended types when working with real data, you will eliminate conversion and therefore save processor time. However, if memory is a consideration and speed is not, you may wish to use simple real numbers, which occupy only 4 bytes rather than the 10 bytes occupied by an extended number.

What's in a Type?

You are already familiar with many predefined types of data such as integer, char and Boolean), which are at your disposal any time in Pascal. To declare a variable of one of these types, you need only set up the necessary statements in the Var section of the routine.

These predefined types are usually sufficient for very simple programs, but you may wish to create your own types for some advanced programs. In fact, Pascal does allow you to make customized types, which can make your program easier to develop as well as easier for others to read and follow. I spoke briefly about these types in Chapter 3; now it is time to take a closer look at them.

For example, take a look at the following declaration:

Type

```
ColorType = ( White, Yellow, Blue, Brown, Black );
```

Var

```
myFavoriteColor : ColorType;
```

I have created an enumerated ColorType. This statement means that any variable declared to be of type ColorType may take on *only* the stated values, in this case white, yellow, blue, brown, and black. The variable myFavoriteColor may carry these values and only these values. Hence I can make the assignment statement

```
myFavoriteColor := Blue;
```

or

```
myFavoriteColor := Yellow;
```

and because I have made all the necessary declarations, Pascal understands. I am not limited to the names of colors, though. Take a look at the following:

Type

```
CarType = ( Sedan, Convertible, Stationwagon, Coupe );
```

Var

```
myCar : CarType;  
yourCar : CarType;
```

I can use these declarations to define these statements:

```
yourCar := Convertible;  
myCar := Coupe;
```

If, however, I accidentally say

```
myCar := Chevrolet;
```

I receive an error from the compiler because “Chevrolet” is not one of the items listed in the declaration `CarType`. Although you and I may agree that a Chevrolet is a type of car, Pascal knows only what you tell it, and according to my instructions, the only valid types of car are sedan, convertible, stationwagon, and coupe.

Be extra careful about punctuation. Do not make this sort of assignment based upon the above declarations:

```
myCar := 'Coupe';
```

because the value “coupe” is enclosed in quotes, hence a string, whereas `myCar` is an enumerated variable, which may take on certain nonstring values.

It is interesting to note that the Macintosh does not place the value “Convertible” in the memory location specified by `myCar`. Rather, an enumerated type is a group of items each of which is assigned an integer value, beginning with 0. For example, Pascal remembers `ColorType` as follows:

```
MacPascal's value:  0    1    2    3    4  
Color Type = ( White, Yellow, Blue, Brown, Black)
```

When you assign yellow to `myFavoriteColor`, Pascal places the value 1 in that slot in memory. This saves space in memory, since the number 1 takes up less room than the word yellow. This should also reinforce the meaning of the values of `Ord`, `Succ`, and `Pred`, described in Chapter 7; the order of yellow is 1, the successor (`Succ`) of Yellow is blue, and the predecessor (`Pred`) of yellow is white.

As a final note on enumerated types, try to think of the Boolean type having a hidden declaration—

```
Type  
boolean = ( FALSE, TRUE );
```

—since a Boolean variable may be assigned only the value TRUE or FALSE and since, most versions of Pascal read the values of FALSE and TRUE as 0 and 1 respectively. If you ever have any problem remembering how enumerated types operate, try to keep in mind this declaration of the Boolean type. Now take a look at a very useful data structure, the array.

Simple Arrays

Up to now I have presented the basic predefined types of Pascal, including integers, real numbers, characters, and strings—integer, real, char, string—as well as the concept of enumerated types. These types are quite useful indeed but would be somewhat limited for general programming without a mechanism to group similar variables. Take for example a program to display the standings of the six teams in baseball's National League East. I place these team names in variables by this declaration:

```
Var
  team1 : String;
  team2 : String;
  team3 : String;
  team4 : String;
  team5 : String;
  team6 : String;
```

Then I assign values like this:

```
begin
  team1 := 'Pirates';
  team2 := 'Cardinals';
  team3 := 'Phillies';
  team4 := 'Mets';
  team5 := 'Expos';
  team6 := 'Cubs';
end;
```

I could have created an array like

```
Var
  teamArray : array [1..6] of String;
```

and made the assignments like this:

```

begin
  teamArray[ 1 ] := 'Pirates';
  teamArray[ 2 ] := 'Cardinals';
  teamArray[ 3 ] := 'Phillies';
  teamArray[ 4 ] := 'Mets';
  teamArray[ 5 ] := 'Expos';
  teamArray[ 6 ] := 'Cubs';
end;

```

TeamArray is an array 1 to 6 of type String. In other words, teamArray may have up to six elements and each element is a String. Picture teamArray as a set of six slots whose values after assignment look like this:

```

6 [ _____ Cubs _____ ]
5 [ _____ Expos _____ ]
4 [ _____ Mets _____ ]
3 [ _____ Phillies _____ ]
2 [ _____ Cardinals _____ ]
1 [ _____ Pirates _____ ]

```

teamArray

I address each element of the array by specifying the name of the array and the element or index position: teamArray[5]. As the index [1..6] states, I may address any whole value from 1 through 6. However, it would be an error to attempt to access an index outside the defined range: TeamArray[7]. This is one of the most common errors encountered when working with arrays, so beware! The compiler directive `{R+/-}`, described earlier, shows how to receive an immediate error message on out-of-range subscripting with strings. This same directive monitors subscript errors with regular arrays.

Array elements may be assigned to one another just like other variable types:

```

teamArray[ 1 ] := 'Pirates';
teamArray[ 2 ] := teamArray[ 1 ];

```

This places the string 'Pirates' in both `teamArray[1]` and `teamArray[2]`. Arrays don't have to contain string types, nor do they have to be indexed by integers. For instance, this array declaration could hold a secret code:

```
Var
  codeArray : array [ A..Z ] of char;
```

I set up a code like this:

```
begin
  codeArray [ A ] := 'Z';
  codeArray [ B ] := 'Y';
  codeArray [ C ] := 'X';
  :
  :
  codeArray [ X ] := 'C';
  codeArray [ Y ] := 'B';
  codeArray [ Z ] := 'A';
end;
```

The `codeArray` may be indexed from A to Z and only character values may be assigned to any element.

In general the array's syntax is

```
varName : array [ Range ] of ArrType;
```

where `Range` is specified (for example 1..10) and `ArrType` is the predefined type of each element. Note that the range cannot be expressed in real numbers. This statement is illegal:

```
thisArray : array [ 1.00..10.00 ] of char;).
```

The most obvious reason for not allowing a real-number index is that it would indicate an infinite number of elements, since there are infinite real numbers between 1.00 and 10.00.

Parallel Arrays

One of the most popular applications of arrays uses the concept of parallel arrays, which are related to one another by index. Consider the situation of a teacher who has 30 students and who would like to write a program that

prompts for each student's name and grade and then sorts them in descending order. The major data structures for this problem might look like this:

```
Var
  nameArray : array [ 1..30 ] of String;
  gradeArray : array [ 1..30 ] of 0..100;
```

If I assign the first three elements of each array based upon

<u>Student</u>	<u>Grade</u>
Jackson	84
Mackowick	100
Craig	91

with the code

```
nameArray[ 1 ] := 'Jackson';
gradeArray[ 1 ] := 84;
nameArray[ 2 ] := 'Mackowick';
gradeArray[ 2 ] := 100;
nameArray[ 3 ] := 'Craig';
gradeArray[ 3 ] := 91;
```

the arrays would look like this:

3	[<u>Craig</u>]	3	[<u>91</u>]
2	[<u>Mackowick</u>]	2	[<u>100</u>]
1	[<u>Jackson</u>]	1	[<u>84</u>]
	nameArray		gradeArray

As you may have noticed these assignment statements, the students' names and grades are matched in the two arrays; (that is, Mackowick's name is in slot 2 of the name array and his grade is in slot 2 of the grade array.). This may seem trivial upon first inspection; however, when the items are sorted in descending grade order, it keeps the relationship intact. Otherwise there would be no way of knowing which name corresponds to which grade.

Records and Files

The student and grade example might better be solved using the data structure of records. A *record* is one or more different items or fields grouped for logic and convenience. For instance, this declaration sets up a record of students and grades:

```
Type
  StudentRec = record
    name : String;
    grade : 0..100;
  end;
```

In StudentRec I establish two fields, name and grade. In general, the declaration of a record looks like this:

```
Type
  RecType = record
    field1 : Type 1;
    field2 : Type 2;
    :
    :
    fieldN : TypeN;
  end;
```

I next declare a variable of this type in order to work with it in a program:

```
Var
  stRecVar : StudentRec;
```

In order to make an assignment to a record's fields, specify the record's name and follow with a dot and the field name. Here I assign values to stRecVar:

```
stRecVar.name := 'Craig';
stRecVar.grade := 91;
```

It's as simple as that! But if I assign new values to those fields—such as the next name and grade—the original values are lost. A simple solution is to declare an array of records like this:

Type

```
StudentRec = record
  name : String;
  grade : 0..100;
end;
```

Var

```
classArray : array [ 1..30 ] of StudentRec;
```

Now I can make my student and grade assignments like this:

```
classArray[ 1 ].name := 'Jackson';
classArray[ 1 ].grade := 84;
classArray[ 2 ].name := 'Mackowick';
classArray[ 2 ].grade := 100;
classArray[ 3 ].name := 'Craig';
classArray[ 3 ].grade := 91;
```

Instead of an array of records, I could have created a file, which is a collection of records:

Type

```
StudentRec = record
  name : string;
  grade : 0..100;
end;
```

Var

```
stRecVar : StudentRec;
stdntFile : file of stRecVar;
```

Files store records on disk for later retrieval. The better to understand the relationship between files, records, and fields, think of the filing cabinet in a doctor's office. The cabinet full of folders on different patients is a file. Each patient's folder is a record, and each item in that folder—name, address, height, weight—is a field.

Sets

The *set* allows you to designate a group of items and determine whether or not a particular item is a member of that group. For instance, say you want to determine whether or not an integer variable holds a single-digit

numeral. The only method covered so far would involve a fairly lengthy set of if statements that would start out like this:

```
if ( ( theNumber = 0 ) OR ( theNumber = 1 ) OR ( theNumber = 2 ) OR ...
```

I can declare a set that holds all 10 values and then check to see if theNumber is a member of that set:

```
Var
  numeralSet      : set of 0..9;
  theNumber       : integer;
  dummyCounter   : integer;
begin
  for dummyCounter := 0 to 9 do
    numeralSet := numeralSet + [ dummyCounter ];
    { this is where you assign the value of theNumber }
    if ( theNumber IN numeralSet ) then
      { then you know that theNumber is a single digit }
    else
      { theNumber is not a single digit }
  end;
end;
```

The for loop used in this example is needed to initialize the value of the set. Just like any other variable, if it is not defined, it may contain garbage. So the method used to assign elements to a set is the plus sign (+). An element is added to the set by listing the set name; an addition sign; and within [brackets], the element itself. Be careful *not* to use {braces} or (parentheses) when referring to set elements. An element may be removed from a set by using the subtraction sign like this:

```
numeralSet := numeralSet - [ 8 ];
```

which removes the value 8 from the numeralSet.

If I want to know whether a particular item is an element of a set, I ask if the element is IN that set, just as I did above with:

```
if (theNumber in numeralSet) then
```

In order to determine whether the item is not an element of the set we would precede the condition of the “if” statement with a NOT operator.

Sets are quite easy to work with and can add to the readability of a program by cutting out large segments of if statements. However, sets generate a

considerable amount of overhead, spent writing the assembly instructions required to manipulate them. In addition, a set may occupy up to 32 bytes of memory depending upon how many items it holds. So if memory space is a critical consideration, keep these points in mind when you consider using sets. Now look at a program that uses arrays, records, and sets to create a baseball team lineup.

Using Arrays, Records, and Sets in an Application

The following program requests information, including name, position, batting average, and spot in the batting order, for nine players on a baseball team. When this information is entered for all nine players, the array of records is scanned through to display the batting order of the team. Take a look at the program line by line:

Program BaseBall;

{R+}

```
(*****
(* This program shows the relationship between fields *)
(* and records as well as showing an application of sets. *)
(*****)
```

Type

PlayerRec = record	{record for each player}
name : String[20];	{player's name}
position : String[15];	{player's position (i.e. Short Stop)}
average : 0..1000;	{player's batting average (i.e. 300)}
batOrder : 1..9;	{player's spot in batting order}

end;

Var

teamArray : array[1..9] of PlayerRec;	{array of players (i.e. a team)}
orderSet : set of 1..9;	{the set of batting order positions}
aChar : char;	{used for "Press any key..."}

Procedure GetPlayerInfo;

Var

counter : integer;	{dummy counter}
valid : boolean;	{is this entry valid?}

```

begin
    for counter := 1 to 9 do
        orderSet := orderSet - [counter];
    for counter := 1 to 9 do
        begin
            ClearScreen;
            writeln;
            writeln;
            write ('What is this player"s name? ');
            readln (teamArray[counter].name);
            writeln;
            write ('What is this player"s position? ');
            readln (teamArray[counter].position);
            writeln;
            write ('What is this player"s batting average? ');
            readln (teamArray[counter].average);
            repeat
                valid := TRUE;
                writeln;
                write ('What is this player"s spot in the batting order? ');
                readln(teamArray[counter].batOrder);
                if (teamArray[counter].batOrder IN orderSet) then
                    begin
                        writeln;
                        writeln ("That spot in the order has already been used!");
                        writeln;
                        aChar := ReadChar;
                        valid := FALSE;
                    end
                else
                    orderSet := orderSet + [teamArray[counter].batOrder];
                until Valid;
            end;
        end;
    end;
end;

```

Procedure DisplayPlayerInfo;

Var

whichBatter, counter : integer; {dummy counters}

```

begin
    ClearScreen;
    writeln;
    writeln;

```

```
writeln ('Your batting order is as follows:');
writeln;
for whichBatter := 1 to 9 do
begin
  counter := 1;
  while (teamArray[counter].batOrder <> whichBatter) do
    counter := counter + 1;
  writeln ('Batter', whichBatter : 1, 'is', teamArray[counter].name,
    'who is playing', teamArray[counter].position);
end;                                     {for whichBatter := 1 to 9}
writeln;
write ('Press any key to continue...');
aChar := ReadChar;
end;                                     {procedure DisplayPlayerInfo}

begin                                     {program Baseball}
  GetPlayerInfo;
  DisplayPlayerInfo;
end.                                     {program Baseball}
```

Figure 9.1 shows a sample run of program Baseball.

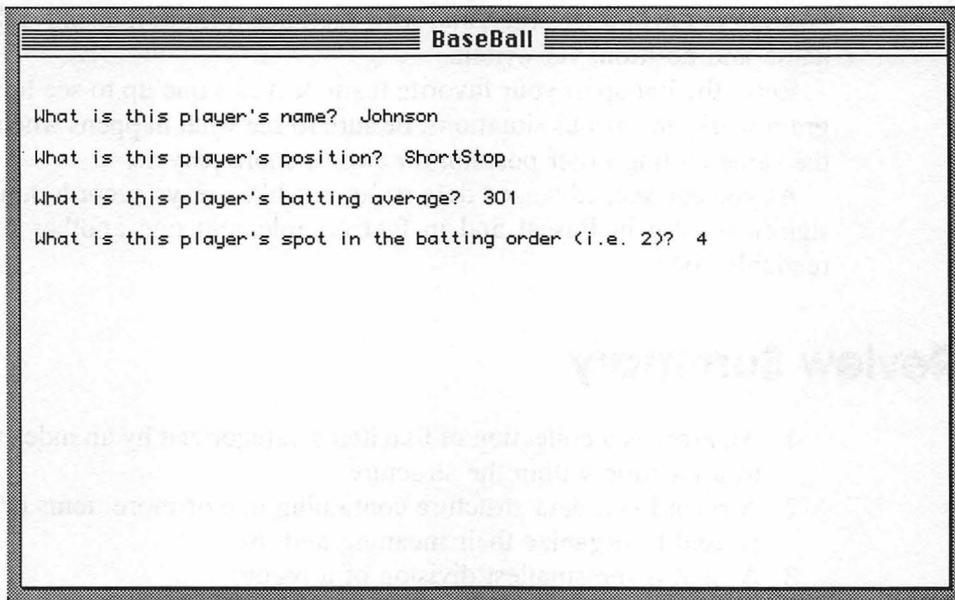


Fig. 9.1.

Baseball has two procedures, one for gathering player information (`GetPlayerInfo`) and another for displaying it (`DisplayPlayerInfo`).

The main program's declarations begin with a record for the players that contains fields for name, position, batting average, and position in the batting order. The team array is declared and may hold nine player records; the set for batting order is declared to hold elements from 1 to 9. The main program calls `GetPlayerInfo` and `DisplayPlayerInfo`.

The procedure `GetPlayerInfo` initializes the `orderSet` so that it has no entries by subtracting all the possible elements from the set. There is another more readable method:

```
orderSet := [ ];
```

The empty set, or null set, is denoted by brackets with nothing between them. You may use either method, but we recommend the empty set, which seems to be the norm.

Next in `BaseBall`, the name, position, average, and batting order of each player are requested. During the request for batting order the `orderSet` makes sure that no two players are accidentally assigned the same position in the batting order, which would make no sense. If the number entered for the batting order is already an element of the set, an error is displayed; otherwise, the entered number is added to the set.

The procedure `DisplayPlayerInfo` scans the `teamArray` for each successive position in the batting order, and with each one it finds, it displays the player's name and position, via `writeln`.

Enter the lineup of your favorite team or make one up to see how the program works in various situations. Be sure to see what happens when you enter the same batting order position for two or more players.

As you can see, advanced data structures like arrays, records, and sets have significant use in Pascal and in fact complement one another for smooth, readable code.

Review Summary

1. An array is a collection of like items categorized by an index that points to a location within the structure.
2. A record is a data structure containing one or more items or fields and is used to organize their meaning and use.
3. A field is the smallest division of a record.
4. A file is a collection of records.

5. A set is a group of elements that may be added to or subtracted from via Pascal operations.

Quiz

1. Describe the relationship between fields, records, and files.
2. Why does it not make sense to put as many fields as possible in a record declaration?
3. What was the purpose of the for loop in the BaseBall program that successively removed the items 1 through 9 in GetPlayerInfo?

Introduction to Advanced Concepts

**Recursion; or, Can I Call Myself?!
The Pointer and Handle Data Types
Almost Everything in Life Has a Point(er)!
Further Study—Stacks, Queues, and Trees
Review Summary
Quiz**

In this chapter you will learn:

- The concept of recursion and how to use it in Pascal.
- The purpose of dynamic storage allocation (DSA), or how to use pointers.
- What handles are and why they are useful on the Macintosh.
- How to work with linked lists and manipulate them in Pascal.
- About such areas as stacks, queues, and trees.

Recursion; or, Can I Call Myself?!

You have seen how procedures and functions can simplify the writing of lengthy programs by centralizing common operations and separating logical blocks of code. I have also explained that procedures and functions can be called by the main program, other procedures, and other functions, depending upon the scope of the program. One point I have not discussed, however, is that a procedure or function can call itself. For example, say I want to write a program that will calculate the product of a given integer and every integer between the given and 1; that is, given 4, the result is $4*3*2*1=24$. This mathematical concept is the factorial and is written as $4!$, which is read “four factorial.” The following program will accomplish this job:

Program Factorial;

```
(*****  
(* This program will calculate the factorial of a number *)
```



```

tempNum := facNum;
CalcFact;
writeln;
writeln ('The result of factorializing,' facNum : 5, 'is', result:10:0);
writeln;
write ('Would you like to calculate another number (Y/N)?');
again := ReadChar;
if (again = 'N') OR (again = 'n') then
  done := TRUE;
writeln;
end;                                     {while NOT done}
end.                                     {program Factorial}

```

The program accepts a number between 1 and 1750 and through the recursive procedure CalcFact calculates its factorial. CalcFact is said to be recursive because it calls itself at the end of the while loop. It makes sense to set up the program in this manner, since it repeatedly performs the same operation on a number. However, the same goal could have been accomplished with a while loop in CalcFact. In fact, most if not all recursive procedures can be eliminated by coding the routine differently. Try to avoid recursive programming for two major reasons:

1. It is very difficult for someone else and sometimes even for the programmer to follow the logic of a recursive routine.
2. Recursive routines are famous for crashing programs because they eat up a lot of RAM. For instance, I had to limit the Factorial program to 1750 because any number greater than that resulted in an error of too many recursive calls. It is difficult to say how many times you can call a recursive routine, but to give you an example, I tried the following program, which displays the number of times its recursive routine was called, and the highest number displayed was 46200 on a 512K Mac:

Program RecCount;

```

(*****
(* This program shows how many times you can call      *)
(* a recursive routine before you run out of memory.   *)
(*****

```

Var

```

  i : longInt;                                     {counts how many times recursion takes place}

```

Procedure RecurseAgain;

```
begin
  if (i MOD 100 = 0) then           {only display every 100 times}
    writeln (i);
    i := i + 1;
    RecurseAgain;
end;

begin                               {program RecCount}
  i := 1;
  RecurseAgain;
end.                                 {program RecCount}
```

This program is not very useful except for proving that recursive routines can run you out of memory if they are not used properly. When you run this program, a count is displayed every time *i* is evenly divisible by 100. When memory is exhausted, a system error is displayed, and you should press the resume key to return to the Turbo editor.

The Pointer and Handle Data Types

I have discussed integers, long integers, real numbers, characters, Boolean values, and so on. These data types all have something in common: when a variable is declared to be of any of these types, a certain portion of memory is allocated to hold the values assigned to the variable throughout the program. Different variable types require different amounts of memory; for instance, a long integer takes up more memory than an integer. I have discussed the memory requirements of these types as I introduced them; you need to know them. The Macintosh has a finite amount of memory, and Turbo Pascal is designed to allow no more than 32K of space for variables in any program. This limit is quite large for the simple programs I have been using, but large applications may eat that much memory in no time at all.

Although it is unlikely that you will run out of memory in your beginning programs, there is no doubt that sophisticated programs dealing with several variables may cause chaos.

One way to save or manage memory is to use pointer. The pointer is analogous to the index at the end of a textbook; the book index provides a page number, or address, where a topic may be found. A reference tells you where to find information. The Pascal pointer holds an address, or memory location, where the value of the variable may be found, just as with variable parameters for procedures and functions. You may recall that a variable parameter is a 4-byte pointer placed on a stack in lieu of a value itself. The type of pointer I will be referring to here is also a 4-byte value.

I can declare a variable like this:

```
Var                { CASE #1 }
  myString : String[255];
```

This sets aside a certain block of memory, 256 bytes, to be used only for the values assigned to myString. Instead, you could declare this pointer type and variable:

```
Type                { CASE #2 }
  StngPtr = String[255];
```

```
Var
  ptrToMyString : StngPtr;
```

This declares a type that points to a String[255] value. Notice that the up arrow (^) is used to designate a pointer type. At this point in each case I have declared a variable that can be used to hold the value of a string. One of the major differences is that CASE #1 has already eaten up 256 bytes of memory, whereas CASE #2 has nibbled only 4 bytes, a fraction of the amount used in CASE #1. I have not yet allocated any memory to hold a string value for ptrToMyString, but I must before I try to assign it a value. The standard Pascal library procedure New allocates memory. Its format is

```
new ( PointerVariableType );
```

where PointerVariableType in CASE #2 is ptrToMyString. Now I can begin to develop a program to use pointers like this:

```
Program SamplePointer;
```

```
Type
  StngPtr = ^String[255];
```

```
Var
  ptrToMyString : StngPtr;
```

```
begin      {program SamplePointer}
  new ( ptrToMyString );
  {now you assign a value to ptrToMyString}
end.
```

Assigning a value to a pointer is slightly different from assigning to a normal variable. To assign your name to the variable in CASE #1 you would say

```
myString := 'Joe Wikert';
```

In order to accomplish the same result with the pointer, you would say

```
ptrToMyString^:= 'Joe Wikert';
```

which says to put 'Joe Wikert' in the memory space you allocated for ptrToMyString. Notice again the use of the up arrow (^) in assignment statements for pointer variables. Our sample program now looks like this:

```
Program SamplePointer;
Type
  StngPtr = ^String[255];

Var
  ptrToMyString : StngPtr;

begin
  {program SamplePointer}
  new ( ptrToMyString );
  ptrToMyString ^ := 'Joe Wikert';
  .
  .
  .
end.
```

At this point there is no advantage to using pointers as opposed to conventional variables in this program, since I have had to perform a new and set aside as large a chunk of memory for this variable as with myString in CASE #1. Actually, with the pointer method I allocated 4 bytes more than with the standard string method because I had to declare the pointer itself. Once finished with the pointer, however, I can free its memory space with the procedure dispose, whose format is identical to new. Here's how it would look:

```
Program SamplePointer;

Type
  StngPtr = ^String[255];
```

```

Var
  ptrToMyString : StngPtr;

begin
  {program SamplePointer}
  new ( ptrToMyString );
  ptrToMyString ^ := 'Joe Wikert';
  {you might want to display the string with a writeln statement here}
  dispose ( ptrToMyString );
  {Now any memory originally set aside for ptrToMyString is available
   for use by other pointer variables, etc.}
end.

```

Keep in mind that the statement

```
ptrToMyString ^ := 'Joe Wikert';
```

assigns a value to what `ptrToMyString` points at and not to the value of `ptrToMyString` itself (since `ptrToMyString` is merely a location in memory). There are two ways to assign a value to `ptrToMyString`. First, you can set it equal to another pointer—

```
ptrToMyString := another Ptr;
```

—where `anotherPtr` is also declared to be of type `StngPtr`.

Or you can assign it the predefined value `nil`, which essentially sets it equal to nothing. Note: It is an error to try to manipulate what a pointer points at if its value is `nil`. This sequence of statements is *not* legal:

```
ptrToMyString := nil;
anotherPtr ^ := ptrToMyString ^;
```

`MemTypes` defines a blind pointer, `Ptr`, which may define var pointers. These can subsequently be assigned to each other even though they may not point to the exact same data type. Another item in `MemTypes` worth mentioning is `Handle`, defined like this:

```
Handle = ^Ptr;
```

What does this mean? A handle is a pointer to a pointer. The Macintosh has a fairly sophisticated memory management system that allows you to compact blocks of free memory. Memory may become fragmented, split into pieces too small for your program to use, and there may be several pieces of

nearly contiguous memory separated only by variables in use. Handles allow all the memory in use to be pushed together, creating larger contiguous blocks of unused memory. Handles are used throughout Macintosh applications for this and other reasons. If you have declared `ourHndle`, `ourHndle^` refers to the pointer `ourHndle` points to and `ourHndle^^` refers to the data pointed to by `ourHndle^`. The handle is a nested pointer.

These are the fundamentals of pointers and handles. To demonstrate their usefulness, I will discuss a practical application of pointers and linked lists.

Almost Everything in Life Has a Point(er)!

Nearly everyone has a daily ritual to perform before work or school. A common sequence of events might look like this:

Wake Up--->Take Shower--->Eat Breakfast--->Brush Teeth--->Leave Home

This series can be referred to as a linked list, which is one item followed by another in a well-defined manner. Each event (except “Leave Home”) points to another event which could be called its next event. For example, the next event after taking a shower is eating breakfast. For a program to keep track of this schedule the obvious choice for a data structure is an array like this:

```

_____
1 [ _____ Wake Up _____ ]
2 [ _____ Take Shower _____ ]
3 [ _____ Eat Breakfast _____ ]
4 [ _____ Brush Teeth _____ ]
5 [ _____ Leave Home _____ ]

```

So array element number 2, Take Shower, immediately precedes element number 3, Eat Breakfast; the general rule for going from one event to the next is to add one to the array index. What if I need to insert an event in this array? Say after I enter the five events, I realize that I read the paper after I wake up and before I take a shower; that is, between elements 1 and 2 of the array. In order to maintain the general rule of going from one event to

the next by adding one to the array index, I have to shift all of the elements from index 2 through 4 down one spot and insert the new step. The array now looks like this:

```

_____
1 [   Wake Up   ]
2 [   Read Paper ]
3 [   Take Shower ]
4 [   Eat Breakfast ]
5 [   Brush Teeth ]
6 [   Leave Home ]

```

The next event after waking up is to read the paper and the next event after that is to take a shower, and so on.

The same sort of logic applies if you want to delete an item from the list. For example, to remove “Eat Breakfast” you have to move all the items below it up one slot in the array. As you can see, writing a Pascal program to solve this problem would be quite complex for the beginner. One major problem is to determine how large an array to declare. If a program with pointers handles most of the work, the job is much simpler, as the following program illustrates:

Program Pointers;

```

(*****)
(* This program illustrates the use of Pascal pointers. *)
(*****)

```

Type

```

StepPointer = ^Step;           {pointer to step record}
Step = record                  {step record of action and next step}
  action   : String;           {what you must do}
  nextStep : StepPointer;      {what's next?}
end;

```

```

Var;
  firstAction  : StepPointer;           {the first thing you must do}
  currentTask  : StepPointer;           {what you are currently doing}
  done         : Boolean;               {are we finished adding jobs?}
  tempString   : String;               {holds step to search for}
  found        : Boolean;               {have we found the entry?}
  saveTemp     : StepPointer;           {temporary pointer}
  addTemp      : StepPointer;           {temporary pointer}
  aChar        : char;                 {used for "Press any key..."}

begin
  new (firstAction);
  new (currentTask);
  new (saveTemp);
  new (addTemp);
  currentTask^.action := 'Wake Up';
  firstAction := currentTask;
  done := FALSE;
  while NOT done do
  begin
    new (currentTask^.nextStep);
    currentTask := currentTask^.nextStep;
    write ('What is your next step? (press <RETURN> to stop) ');
    readln (currentTask^.action);
    writeln;
    if (currentTask^.action = ") then
      done := TRUE;
    end;                               {while not Done}
    currentTask^.nextStep := nil;       {last step points to nothing}
    currentTask := firstAction;
    ClearScreen;
    writeln;
    writeln ('Your steps are as follows:');
    while (currentTask^.nextStep <> nil) do
    begin
      writeln ('    ', currentTask^.action);
      currentTask := currentTask^.nextStep;
    end;                               {while currentTask^.nextStep <> nil}
    writeln;
    write ('After which step do you wish to insert? ');
    readln (tempString);
    currentTask := firstAction;
    found := FALSE;
    while (currentTask^.nextStep <> nil) AND (NOT found) do

```

```

if (currentTask^.action = tempString) then
  found := TRUE
else
  currentTask := currentTask^.nextStep;
if (NOT found) then
begin
  writeln;
  writeln ('Sorry, that's not in the list!!');
  writeln ('Try the program again and write down your steps...');
end
else
begin
  writeln;
  write ('What do you wish to insert? ');
  readln (addTemp^.action);
  writeln;
  saveTemp := currentTask^.nextStep;
  currentTask^.nextStep := addTemp;{make previous point to add}
  addTemp^.nextStep := saveTemp;
  currentTask:= firstAction;          {go back to beginning}
  ClearScreen;
  writeln ('Here"s your new list:');
  while (currentTask^.nextStep <> nil) do
  begin
    writeln (' ', currentTask^.action);
    currentTask := currentTask^.nextStep;
  end;
end;          {while (currentTask^.nextStep <> nil)}
writeln;
write ('Press any key to continue...');
aChar := ReadChar;

dispose (firstAction);
dispose (currentTask);
dispose (saveTemp);
dispose (addTemp);
end.          {program Pointers}

```

Let's look closer at the Pointers program by first studying our pointer type declaration:

```

StepPointer = ^Step;
Step = record
  action : String;
  nextStep : StepPointer;
end;

```

This declaration is read as “StepPointer points to a Step record, which consists of two fields. The first field is called action and is a String type, and the second field is called nextStep and is a StepPointer type.” Two new items have been introduced to you in this declaration:

1. Declaring StepPointer to be of type Step, normally would generate an error message, since Step is not yet declared. However, if I declare Step first, I will also have declared nextStep as type StepPointer, which has not yet declared. This appears to be Catch-22, but Pascal realizes this, and by convention you should always declare the pointer first, and then if necessary declare the type to which it points immediately afterward. I have done so by declaring first the type StepPointer and then Step, the type to which StepPointer points.
2. Within the pointer declaration of Step, I declared another pointer—nextStep : StepPointer. As you will see shortly, this permits a record for each of daily morning task and a field in the record that points to the next task.

The program begins by performing a new on four pointer types, which are explained in the comments. The next step is to assign Wake Up to currentTask^.action; that says that the current task is Wake Up. The FirstAction pointer is set equal to currentTask to provide a reference point to the first task; that is, after I change the value of currentTask, I will still have firstAction pointing to the beginning of the series of events.

Done is initialized to FALSE, setting off a while loop. Within this while loop the program will continue adding items to the list until a Return is pressed with no data. Next a new currentTask^.nextStep provides the currentTask with a location that holds the value of the nextStep. Assuming this is the first time through the while loop, I now have a pointer to the first event (firstAction); WakeUp is the currentTask; and currentTask will point to the next event, currentTask^.nextStep. Next a request is made for the user to enter the next step; you might enter Read Paper. If nothing is entered but Return is pressed in response to currentTask^.action = ”, the user is finished and done set to TRUE. When this while loop is complete, currentTask is set to nil, so the last event in the series, Leave Home, points to nothing. The current task is then set equal to firstAction, the beginning of the list, to display the steps for the user. A while loop displays currentTask^.action and then sets currentTask equal to currentTask^.nextStep. The display keeps on moving through the list until currentTask^.nextStep is nil.

The lines of code up to this point in the program are all you need to build a *linked list*, which has a definite order defined by pointer values to each next item in the list. The remaining code shows how to insert an item in the

linked list. `TempString` prompts the user to enter the name of the item, after which he or she wishes to insert another item. `currentTask` is made to point at the beginning of the list by assigning it the value of `firstAction`, and the Boolean `found` is set to `FALSE`. A while loop continues to search the linked list for the item, and if it is located, `found` is set to `TRUE`; otherwise the loop continues to search until either the end of the list is encountered or the item is located. If the item is not in the list, a message suggests running the program again. If the item is located, the user is asked what to insert. Notice that `currentTask` now points to the item before the insert. So a temporary variable, `saveTemp`, mimics `currentTask^.nextStep` to preserve this value and make `currentTask^.nextStep` point to the new item. Finally, the new item is made to point at the previous value of `currentTask^.nextStep` by assigning it the value of `saveTemp`. The three statements

```
saveTemp := currentTask^.nextStep;  
currentTask^.nextStep := addTemp;  
addTemp^.nextStep := saveTemp;
```

are the fundamental statements used to manipulate the linked list so that a new item may be inserted. Study them carefully and be sure to understand their significance before you do any work with linked lists.

Finally, by moving `currentTask` to the beginning of the list again, the program displays the updated linked list. A dispose is performed on all the pointer variables, making available any memory they were using. This last step has little significance in the program `Pointer`, since it takes place at the end of the program, but if there were more statements after the disposes, there would be more memory available to other variables. This is the foundation of dynamic storage allocation: use only the memory absolutely necessary for that portion of the program. Now take a look at some other item-handling methods: stacks, queues, and trees.

Further Study—Stacks, Queues, and Trees

There are several different ways of handling items for processing by a program. I have shown how to store elements in arrays and go through the array to find a particular element. I have also shown how to use pointers to keep track of events in your morning ritual. The first two processing strategies mentioned above, stacks and queues, are quite similar in that their ordering is linear; you can go straight down a sequence of items and determine which one is next to be processed.

A stack is just like the tray bin at your local cafeteria: data, or trays, are

processed, or used, in order from the top down. When people go through the line of a cafeteria they take the next available tray off the top of the stack. When clean trays are returned to the stack, they are placed at the top of the stack. This leads to what is commonly referred to as a last-in-first-out, or LIFO, traffic because the last tray put onto the stack is the next one taken off.

In contrast, a queue is a first-in-first-out, or FIFO, strategy, analogous to the line in a bank, where the first customer in line is the first one served. When new items are placed into the queue they are placed at the end and are processed after all previously entered items.

These two methods of processing data may be implemented using an array. To use an array for a stack, place each new item at the next available index and take each item to be processed off the top of the stack. This means that you must know the top item's index at all times; an integer variable can be used for this purpose. If an array is used to simulate a queue, things get a bit more complicated. Place new items at the next available index location, but when you take an item from the queue, you must remove it from the bottom, or index 1, and slide all the remaining items down the array so that index item 2 gets moved to index 1, index item 3 get moved to index 2, and soon. Two common problems in this sort of programming:

1. Trying to remove an item from an empty stack or queue.
2. Placing one too many items in a stack or queue.

Both of these problems may be avoided by carefully coding and using arrays, but they could more logically be avoided with pointers and linked lists.

The final topic of this chapter is binary trees. You saw in Chapter 8 how a binary search can quickly deduce a number between 1 and 100. The logic behind a binary tree is very similar to that of the binary search. A binary tree may be used to arrange items so they are more easily retrieved for the processing. For example, I wish to sort these numbers: 47, 3, 17, 29, 81, 1, and 12 so I can quickly say whether a particular number is in our list. I start out by placing the first number, 47, at the top of the tree and examining the next number to determine whether it is less than or greater than the first number; since it is 3, it is less, and so I place the 3 below and to the left of the 47 as shown in Figure 10.1.



Fig. 10.1.

The next number, 17, is less than 47, so I go to the left of 47 and find 3. Since 17 is greater than 3, I go to the right of 3 and find nothing there. I place the 17 below and to the right of the 3 as shown in Figure 10.2.

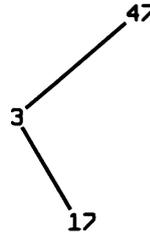


Fig. 10.2.

The next number, 29, is less than 47, so I look to the left and again find the 3. Since 29 is greater than 3, I look to the right of 3 and find 17. Again, 29 is greater than 17, so I look to the right of 17. Finding nothing, I place 29 below and to the right of 17 as shown in Figure 10.3.

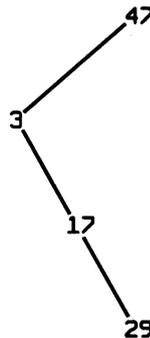


Fig. 10.3.

I use the same process to place 81, 1, and 12 on the tree so that the final tree looks like Figure 10.4.

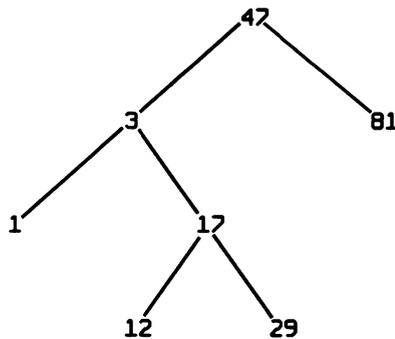


Fig. 10.4.

In this binary tree 47 is the base from which all sorting and searching begins. The simple pattern for inserting numbers is to compare the new number to the current base and look for a new base or empty slot to the left if the new number is less than the base and to the right if the new number is greater than the base. I continue to search for a place to insert the new number until I find an empty slot. You can see that the final configuration of numbers looks very much like an upside-down tree or the roots of a tree.

Each number in the tree has two nodes associated with it. For 47 the left node is 3 and the right node is 81. Some of the numbers in the trees have no value for either node. For example, 1 has no values for its left or right nodes. It is also possible for a number to have only one node with a value, although none of the numbers in this tree fall into this category.

To code a Pascal program to create a binary tree, this pointer type is very appropriate:

```
Type
  BinaryPointer = ^BinPtrType;
  BinPtrType = record
    number : integer;
    leftNode : BinaryPointer;
    rightNode : BinaryPointer;
  end;
```

This declares a variable, `BinaryPointer`, to represent current location on the tree. I need another variable of this type to point to the base of the tree. To insert a number, compare it against the current spot in the tree and go either left or right to check further. When you find `LeftNode` to be nil (assuming the new number is less than the current number) or `RightNode` to be nil (assuming the new number is greater than the current number), make that directional node point to the new number, since the current number points to nothing in that direction. Continue to search for a number until you find a nil, which means that the number is not in the list.

Stacks, queues, and trees are only some of the techniques used to arrange and sort items for processing. With the assistance of pointers and linked lists you can use them effectively, but be extra cautious to avoid the errors that may arise when using this dynamic storage type; watch out for disposing of a necessary pointer and for trying to access what is pointed at by a nil pointer.

Review Summary

1. Recursion is the act of a procedure or function calling itself repeatedly. One of the most popular examples is the mathematical factorial function.
2. A pointer is useful for dynamically allocating memory space, or occupying memory only when necessary.
3. A handle is a pointer to a pointer. Handles permit the compression of occupied memory.
4. A linked list is organized so that the first item points to the second, the second to the third, and the final item points to nothing, or nil.
5. Last-in-first-out, or LIFO, is an item organization method in which the last item in the group is the first to get processed, similar to the stack of trays in a cafeteria.
6. First-in-last-out, or FIFO, is an item organization method in which the first item in the group is the first to get processed, similar to the line of customers in a bank or grocery store.

Quiz

1. Why is it not a good idea to incorporate recursive routines in Pascal programs?
2. In the daily ritual linked list, why was it easier to use a linked list with pointers than an array to represent the list?
3. How would the following sequence of numbers be organized in a binary tree: 1, 3, 4, 5, 2?
4. What is the difference between a stack and a queue?

More on Records and Files

Fields, Records, and Files
Files versus Arrays of Records
File-Handling Library Procedures
Using a File in a Phone-Book Program
Sorting and Merging Records
Variant Records
Review Summary
Quiz

In this chapter you will learn:

- About the organization of fields within records and records within files.
- The similarities between files and arrays of records.
- Some of the most useful file-handling library routines.
- How to sort and merge records within a file.
- What variant records are and how to use them.

Fields, Records, and Files

Every day people deal with files of many different types. The phone book is one of the most obvious examples of a file. It consists of numerous entries containing the names, addresses, and phone numbers of local residents. Below is a portion of a sample phone book:

Craig, Janet E.	451 Emerson Rd.	555-1200
Johnson, Ralph	1121 Elm Place	555-1201
Mackowick, Paul	1487 Altaview Ave.	555-1202
Zimmerman, Larry	1234 Maple Ave.	555-1203

The entire book is a file, each entry is a record, and each record has three fields: name, address, and phone number. The following type could represent a phone-book record:

```
PhoneBookType = record
  name   : String;
  address : String;
  number : String[8];
end;
```

If `phBkVar` were declared to be of type `PhoneBookType`, the fields of the record would read like this:

```
write ( 'What is the person's name? ' );
readln ( phBkVar.name );
write ( 'What is their address? ' );
readln ( phBkVar.address );
write ( 'What is their phone number? ' );
readln ( phBkVar.number );
```

These would go into a Pascal file, but since you don't know how to do this yet, look at an alternative.

Files versus Arrays of Records

If I wanted to work with several phone-book records within a program, I could declare a variable like this:

```
Var
  phBkArray : array[ 1..10 ] of PhoneBookType;
```

This allows me to read in up to 10 records like this:

```
for i := 1 to 10 do
begin
  write ( 'What is the person's name? ' );
  readln ( phBkArray[ i ].name );
  write ( 'What is their address? ' );
  readln ( phBkArray[ i ].address );
  write ( 'What is their phone number? ' );
  readln ( phBkArray[ i ].number );
end;
```

This array can be sorted alphabetically by last name, by phone number, or by street address. The main problem arises when saving the information for future use; when the program ends, the array's contents are forever lost.

Although you could have declared a larger array, you will always be limited by the size of the array if you want to work with many more records.

The Pascal file type can simplify this problem. I declare a file within the program such as this:

```
Type
  PhoneBookType = record
    name   : String;
    address : String;
    number : String[8];
  end;

Var
  phoneFile : file of PhoneBookType;
```

This declaration will later allow me to place multiple PhoneBookType records on a disk so that I can later retrieve and manipulate them. The file is the obvious choice for storing phone book records, since the information may be retrieved after the program has been terminated and since the number of records is usually limited only by the available disk space. The manipulation of files involves several major Pascal procedures: rewrite, reset, read, write, and close.

File-Handling Library Procedures

In order to open a file for rewriting, use this procedure:

```
rewrite ( ProgFileName, DiskFileName );
```

ProgFileName is the file variable that calls up the file, and DiskFileName is the name of the file as it appears on the disk or on the Macintosh desktop.

In order to open a file for subsequent reading, the reset routine is used as follows:

```
reset ( ProgFileName, DiskFileName, [BuffSize] );
```

The first two parameters represent the same items as in rewrite above, and the optional (as denoted by the brackets) parameter BuffSize specifies the size of the buffer to use for reading (usually 512 bytes) and is used only for TEXT files.

Once you have opened a file, you may read or write to it using these procedures:

```
read ( ProgFileName, Component );  
write ( ProgFileName, Component );
```

Again, ProgFileName is the file variable and Component is the item you wish to read or write from or to the file.

When you are finished with a file, always close it with this procedure:

```
close ( ProgFileName );
```

Always use a corresponding close upon every opened file. If you do not follow that simple rule, you may end up with many problems in your programs.

When reading from a file, do not attempt to read beyond the last record, since there is nothing there. The Boolean EOF (End Of File) function is used to detect this situation and may be expressed like this:

```
while NOT EOF ( phoneFILE ) do  
  {perform your reading, etc.}
```

EOF (phoneFile) will return TRUE if you have reached the end of the file, or FALSE if there is still at least one more record in the file.

The file-handling routines described up to this point are the standard ones in almost any version of Pascal on other machines. Turbo Pascal offers several other routines that may be used in their place. Take a look at the most useful of these Turbo-based file routines.

Instead of using reset or rewrite to open files, Turbo offers the functions FSOpen and Create, which use MemTypes, QuickDraw and OSIntf, so you have to specify them in a Uses statement. The syntax for FSOpen is

```
result := FSOpen ( FName, FNum, RefNum );
```

where the result is an OSErr, which is defined in OSIntf as an integer. OSIntf also declares several constants, for example FNFErr, which may be checked against result for the file status. The parameter FName is the name of the file in string format; FNum is the number to associate with the file; and RefNum is the reference number returned from FSOpen. This number is used for identification in subsequent accesses to the file.

If you try to open a file via `FSOpen` and the result is `FNFErr` (file not found error), you'll need to Create it like this:

```
result := Create ( FName, RefNum, Creator, FileType );
```

`FName` and `RefNum` are the same parameters as in `FSOpen`; `Creator` is a four-character ID you specify as the file creator; and `FileType` is a four-character ID you specify as the file's type. These last two parameters may be set as you deem appropriate. For example, in a program that writes telephone numbers to a file I have specified the `Creator` as "PHNE" and the type of file as "NUMS."

When you open a file, you find yourself at the beginning of it. If you want to append it, use the function `SetFPos`, whose syntax is

```
result := SetFPos ( RefNum, Mode, Offset );
```

where `RefNum` is the reference number and `Mode`, the type of offset you specify, is one of four constant values:

<code>FSAtMark</code>	0
<code>FSFromStart</code>	1
<code>FSFromLEOF</code>	2
<code>FSFromMark</code>	3

`Offset` derives from what you have specified as the `Mode`. For example, to find the end of a file, do it like this:

```
result := SetFPos ( refNum, FSFromLEOF, 0 );
```

Once you have the file pointer correctly situated, you can write to it via the function `FSWrite`, whose syntax is

```
result := FSWrite ( RefNum, Size, BufPtr );
```

where `RefNum` is as defined previously; `Size` is the size of the buffer to be written; and `BufPtr` is the address of the item to be written. If I have properly opened a file and wish to write a record to it, I can do so like this:

```
recSize := SizeOf ( phoneRecord );  
result := FSWrite ( refNum, recSize, @phoneRecord );
```

where `phoneRecord` is defined like this:

```
Type
  PhoneType = record
    firstName : String[ 10 ];
    lastName  : String[ 10 ];
    number    : String[ 8 ];
  end;

Var
  phoneRecord : PhoneType;
```

You may wonder what the `@` before `phoneRecord` means. This tells the Turbo compiler to pass the address of `phoneRecord` and not the variable itself; remember that the final parameter (`BufrPtr`) of `FSWrite` is an address. You can specify the address of any variable by placing the `@`, or address operator, before its name.

If you wish to read from the file rather than write to it, use the function `FSRead`, whose syntax is

```
result := FSRead ( RefNum, Size, BufrPtr );
```

Each of these parameters is the same as those listed with `FSWrite`.

When finished with a file, close it via `FSClose` with the following syntax:

```
result := FSClose ( RefNum );
```

If you just finished writing to the file, you should always call `FlushVol` *after* `FSClose` like this:

```
result := FlushVol ( StrngPtr, FNum );
```

where `StrngPtr` may be a nil pointer and `FNum` is the number specified in `FSOpen`.

As we have mentioned, the `Result` may be one of several constants declared in `OSIntf`, and it should be checked after any I/O but is most commonly viewed only after `FSOpen` to see if the file exists or not.

Using a File in a Phone-Book Program

The concepts described above are brought out in this program, which may be used to keep your own private phone book on a Macintosh disk:


```

Procedure SearchFile;

Var
  findName : String[10];           {name user wants to search for}
  found    : boolean;             {have we found the name yet?}

begin                               {procedure SearchFile}
  write ('What is the last name you wish to search for? ');
  readln (findName);
  anErr := FSOpen (userFileName, 0, refNum);
  found := FALSE;
  while (NOT found) AND (anErr <> EOFErr) do
  begin
    anErr := FSRead (refNum, recSize, @phoneRecord);
    if (phoneRecord.lastName = findName) then
      found := TRUE;
    end;
    writeln;
    if found then
    begin
      write (phoneRecord.firstName, ' ', phoneRecord.lastName, "S");
      writeln ('phone number is:', phoneRecord.number);
    end
    else
      writeln ('Your file contains no number for,' findName);
    anErr := FSClose (refNum);
    (*****
    (* Display info until user is finished reading it.          *)
    (*****
    writeln;
    writeln;
    writeln('  Press any key to continue');
    response := ReadChar;
  end;                               {procedure SearchFile}

begin                               {main program PhoneBook}
  recSize := SizeOf (phoneRecord);
  done := FALSE;
  write ('Enter your file"s name and press <RETURN>...');
  readln (userFileName);
  if userFileName = " then
    userFileName := 'Nameless';
  (*****
  (* Need to check to see if this file exists first.          *)
  (* If it doesn't, we need to create it.                      *)

```

```

(*****)
anErr := FSOpen (userFileName, 0, refNum);
if anErr = fnfErr then           {File not found error}
  anErr := Create (userFileName, refNum, 'PHNE', 'NUMS')
else
  anErr := FSClose (refNum);

while NOT done do
begin
  ClearScreen;
  writeln;
  writeln;
  writeln ('Which do you wish to do:');
  writeln;
  writeln ('  1. Add a record to the file');
  writeln ('  2. Search the file');
  writeln ('  3. QUIT');
  writeln;
  write ('PLEASE ENTER A 1, 2, OR 3 ');
  response := ReadChar;
  ClearScreen;
  case response of
    '1': AddRecord;

    '2': SearchFile;

    '3': done := TRUE;

    otherwise;
  end;
end;                               {case Response}
end;                               {while not done}
end.                               {program PhoneBook}

```

This program may be used to create your own disk-based phone book file, add records to it, and search for particular entries. Figure 11.1 shows an example of the program's interactive output.

PhoneType is a type of record written to the file; it has fields for first and last names and phone numbers. The program contains the main program and two procedures, AddRecord and SearchFile, which place an entry at the end of the file and find a particular entry within the file respectively.

The main program asks for a file name (userFileName) to use in the remainder of the program. If the user presses Return with no name, the default "Nameless" is assigned to the file. The program menu, including the options to add a record, search the file, and quit the program, is displayed. Depending

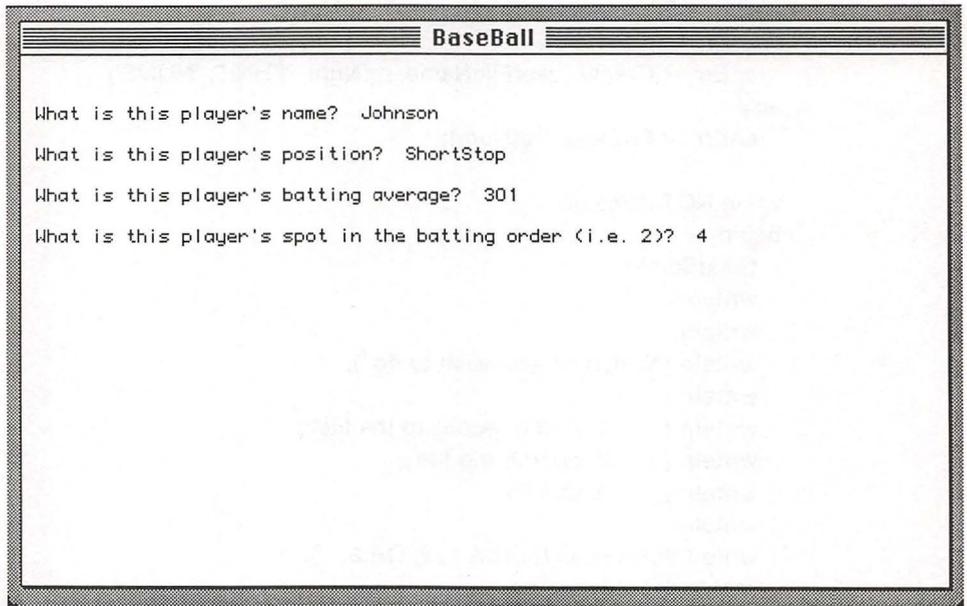


Fig. 11.1.

upon the selection either `AddRecord` or `SearchFile` will be invoked, or `Done` will be set to `TRUE` and the program will be terminated.

When `AddRecord` is invoked, the procedure requests the first name, last name, and phone number of the record to be added to the file. The file is opened and the end of the file is located via this code:

```
anErr := SetFPos ( refNum, FSFromLEOF, 0 );
```

After the end of file is found, the record is written (`FSWrite`), the file is closed (`FSClose`), and the output is flushed (`FlushVol`).

When `SearchFile` is called, the user is asked which last name he or she wishes to locate in the file. The file is opened, and the search is begun through a while loop that continues until either the last name is found or end of file is reached (`EOFErr`). If the name is found, the entire name is displayed along with its phone number; otherwise a message explains that the name is not in the file. In either situation the file is closed (`FSClose`) before returning to the main program.

The phone-book program creates a very simple file; there is no real order to the records within the file. When a new record is added to the file it is placed at the end, not in alphabetical order like your local phone book. This should not create a problem for a small file that is infrequently used, but

if there are quite a few entries in the file and you search it often, it would be nice to not have to look at almost every record in the file to find the one you want. If the file is stored in alphabetical order, there is a noticeable difference in speed when accessing a particular record. Before you do this, I must discuss the concepts of sorting and merging records.

Sorting and Merging Records

The phone book is a list of names and numbers in alphabetical order. If the book were not in any order, it would take hours to find the number of a friend. My phone book program has just that problem, but since the computer can search files so quickly, this time goes almost unnoticed unless the file is quite large. In order to make my phone book program create a file similar to the phone book, I must sort the records in the file before actually writing to and closing it. If I need to add a record that would be the last record in the file, I simply add it to the end as in the original program. But when I need to insert the record at the beginning or between two existing records, I must make room for the new entry and merge it with the existing records. The following program has the modifications necessary to create an alphabetical phone-book file;

Program SortPhoneBook;

{R+}

```
(*****
(* This program illustrates the use of sorted files.      *)
(*****)
```

Uses

MemTypes, QuickDraw, OsIntf;

Type

```
PhoneType = record                                {phone-book file type}
  firstName : String[10];                          {entry's first name}
  lastName  : String[10];                          {entry's last name}
  number   : String[8];                            {entry's phone number}
end;
```

Var

```
userFileName : String[10];                        {user's phone-book file name}
phoneArray   : array[1..50]of PhoneType;         {array for sorting}
```

```

response      : char;           {menu selection}
done          : boolean;       {are we finished?}
phoneRecord   : PhoneType;     {var for writing/reading PhoneFile}
index, counter : 0..50;       {PhoneArray index}
anErr        : OSErr;         {errors returned from file routines}
refNum       : integer;       {file path reference number}
recSize      : longint;      {tells file routines how large record is}

Procedure InsertAndSort;

Var
  locatedSlot : boolean;       {have we found the spot to insert
                                record?}
  dummy      : 0..50;         {used to slide array elements down}

begin                               {procedure InsertAndSort}
  locatedSlot := FALSE;
  counter := 1;
  while (counter <= index) AND (NOT locatedSlot) do
  begin
    if (phoneArray[counter].lastName < phoneRecord.lastName) then
      counter := counter + 1
    else
      locatedSlot := TRUE;
  end;                               {while (counter <= index) AND (NOT
                                    locatedSlot)}

  for dummy := index downto counter do
    phoneArray[Dummy + 1] := PhoneArray[Dummy];
  phoneArray[counter] := phoneRecord;
end;                               {procedure InsertAndSort}

Procedure AddRecord;

Var
  dummy : 0..50;              {used as index for writing to file}

begin                               {procedure AddRecord}
  index := 1;
  anErr := FSOpen(userFileName, 0, refNum);
  while (anErr <> EOFErr) do
  begin
    anErr := FSRead (refNum, recSize, @phoneArray[index]);
    index := index + 1;
  end;
  anErr := FSClose (refNum);

```



```

if found then
begin
  write (phoneRecord.firstName,' ', phoneRecord.lastName, "'s");
  writeln ('phone number is:', phoneRecord.number);
end
else
  writeln ('Your file contains no number for', findName);

(*****
(* Allow user to read screen before proceeding.          *)
*****)
write (' Press any key to continue...');
response := ReadChar;
anErr := FSClose (refNum);
end;                                     {procedure SearchFile}

begin                                     {main program SortPhoneBook}
  recSize := SizeOf (phoneRecord);
  done := FALSE;
  write ('Enter your file"s name and press <RETURN>...');
  readln (userFileName);
  if userFileName = " then
    userFileName := 'Nameless';

(*****
(* Need to check to see if this file exists first.        *)
(* If it doesn't, we need to create it.                   *)
*****)
anErr := FSOpen (userFileName, 0, refNum);
if anErr = fnfErr then                    {File not found error}
  anErr := Create (userFileName, refNum, 'PHNE', 'NUMS')
else
  anErr := FSClose (refNum);

while NOT done do
begin
  ClearScreen;
  writeln;
  writeln;
  writeln ('Which do you wish to do:');
  writeln;
  writeln (' 1. Add a record to the file');
  writeln (' 2. Search the file');
  writeln (' 3. QUIT');
  writeln;

```

```

write ('PLEASE ENTER A 1, 2, or 3 ');
response := ReadChar;
writeln;
case response of
  '1' : AddRecord;

  '2' : SearchFile;

  '3' : Done := TRUE;

end;                                {case response}
end;                                {while NOT done}
end.                                {program SortPhoneBook}

```

This new program uses a `phoneArray` that can hold up to 50 records and that is used to sort the file before writing it to disk. The procedure `AddRecord` has been changed to read the file into the array and then call `InsertAndSort`. This procedure finds where the insertion should be made, slides all subsequent records down one position in the array, and places the new record in its proper position. Finally, the procedure `SearchFile` no longer simply searches the file until it finds either the record or the end of the file; it knows that the file is arranged alphabetically, so once it gets past the point where the record should be, it quits. For example, if you are searching for the name Boyle and you come to the name Carlson, you have gone too far and the name is not in the file.

You probably will not notice much difference in this version of the program, but to see how the file itself is completely different, use the following program to display the contents of your phone-book file.

Program `DisplayPhoneBook`;

```

(*****
(* This program may be used to display the phone-      *)
(* book file created via the PhoneBook program        *)
(* presented earlier.                                  *)
(*****

```

Uses

`MemTypes`, `QuickDraw`, `OSIntf`;

Type

```

PhoneType = record                                {phone-book file type}
  firstName : String[10];                          {entry's first name}

```

```

        lastName : String[10];           {entry's last name}
        number   : String[8];           {entry's phone number}
end;

Var
    phoneRecord : PhoneType;           {var for writing/reading PhoneFile}
    userFileName : String[10];         {name of user's file}
    fileExists   : boolean;           {did the user enter an existing file?}
    anErr        : OSErr;             {errors returned from file routines}
    aChar        : char;              {used for "Press any key to..."}
    refNum       : integer;           {file path reference number}
    recSize      : longint;          {tells file routines how large record is}

begin                                     {main program DisplayPhoneBook}
    fileExists := FALSE;
    while NOT fileExists do
    begin
        ClearScreen;
        write ('Enter your file's name and press <RETURN>...');
        readln (userFileName);
        if userFileName = " then
            userFileName := 'Nameless';
        (*****
        (* Does the file exist? *)
        (*****
        anErr := FSOpen (userFileName, 0, refNum);
        if anErr = fnfErr then           {File not found error}
        begin
            writeln ('That file doesn't exist...');
            writeln ('Press any key to continue...');
            aChar := ReadChar;
        end
        else
            fileExists := TRUE;
    end;
    writeln;
    recSize := SizeOf (phoneRecord);
    while (FSRead(refNum, recSize, @phoneRecord) <> EOFErr) do
        writeln ('      ', phoneRecord.firstName, ' ',
            phoneRecord.lastName, ' ', phoneRecord.number);

    (*****
    (* Now await input to continue. *)
    (*****

```

```
writeln;
write ('   Press any key to continue...');
aChar := ReadChar;
anErr := FSClose (refNum);
end.                                {program DisplayPhoneBook}
```

With a good understanding of records and how they may be written to disk files, take a look at a special type of record, the variant record.

Variant Records

Now that you know how to write files to disks, say you want to write a program to keep some statistics on your favorite athletes. Maybe you have four favorite sports: baseball, basketball, football, and hockey. Further, all you wish to maintain is the player's name, what sport he plays, and some statistics about offensive play, so you come up with a structure like this:

Type

```
SportType = (Baseball, Basketball, Football, Hockey);
```

```
SportRecord = record
```

```
  playerName      : String[40];
  whatSport       : SportType;
  atBats          : integer;           {baseball stats}
  hits            : integer;
  walks           : integer;
  runs            : integer;
  twoPtFieldGoals : integer;         {basketball stats}
  threePtFieldGoals : integer;
  freeThrows      : integer;
  touchDowns     : integer;         {football stat}
  goals           : integer;         {hockey stats}
  assists         : integer;
```

```
end;
```

You can subsequently define a variable to be of type `SportRecord` and use it in a program, having read in the name, sport, and statistics. The size of this record is 64 bytes: `String[40]` takes 42 including the length indicator (plus-pad byte); `whichSport` takes 1 plus the pad byte; and the statistics require 10 2-byte integers. So every time you write one of these records to a file, you are writing 64 bytes. What if most records are for football players? You need to write only 46 bytes of information, since the only fields used for football

where `EnumeratedItem` is one of the possible values of the case type; for example, `Baseball` is one of the values for the `SportType`. The list of variables associated with that variant portion is then placed within parentheses. Finally, the structure is completed with the `end` statement. The functionality of the record structure is not changed; if `playerRec` is declared as a `SportRecord`, you still refer to each field like this:

```
playerRec.atBats
```

or

```
playerRec.whichSport
```

The most significant task so far accomplished is that the total record size has been decreased from 64 bytes to 52 bytes without sacrificing any function. The record size is smaller because the separate sport blocks overlay each other so that no space is reserved for empty categories. The largest block is the one for `Baseball`, which has four 2-byte integers. The total length of `SportType` is the sum of the fixed fields and the longest variant portion:

<code>playerName</code>	42 bytes
<code>whichSport</code>	2 bytes
<code>Baseball</code>	<u>8 bytes</u>
	52 fixed bytes

`BaseBall` is the longest variant portion at four 2-byte fields.

When using a variant record, it is the programmer's job to make sure always to look at the correct variant block. For instance, don't look at `atBats` when working with a hockey player. This is no different from working with the original `SportType`. Note that if you know the value of `whichSport`, you can determine exactly which variant block to use.

You have just completed one of the most important chapters in this book, all about working with disk-based files and record structures to write even the most sophisticated of programs. Be sure to take another look over the major topics of this chapter before moving on to our next subject, debugging.

Review Summary

1. The EOF (filename) function is the end-of-file status, which is set to `TRUE` when there are no more records in the file.
2. Turbo Pascal supports the standard Pascal file I/O routines such as `reset`, `close`, `read`, and `write`, but more important, it offers several other

I/O routines that permit more flexibility and better error handling. The most useful of these routines are Create, FSOpen, FSClose, SetFPos, FSWrite, FSRead, and FlushVol.

3. The operator @ may be used to determine the address of a variable by placing it before the variable's name; for instance, address := @myVar.
4. Sorting is the act of organizing records in a particular order, for example alphabetically or numerically.
5. Merging is the act of combining a set of records with existing records so that the entire new file is properly sorted.
6. A variant record structure allows you to overlay mutually exclusive blocks of fields within a record.

Quiz

1. Draw an analogy between a file to and collection of musical albums.
2. In the program PhoneBook, what was the purpose of this statement:

```
anErr := SetFPos ( refNum, FSFromLEOF, O );
```

3. What is the most important advantage of using variant record structures?

Debugging and File Analysis

Debugging
File Analysis
Encipher/Decipher Program
MacsBug
TMON
Review Summary
Quiz

In this chapter you will learn:

- Some tips on debugging programs with Turbo Pascal on the Macintosh.
- A file-dump utility that will allow you to view entire contents of files.
- A file-encryption program to protect your files from unwanted viewing.
- How to use the debuggers available to you with Turbo Pascal on the Macintosh: MacsBug and TMON.

Debugging

Program logic up to now has, I trust, been very easy for you to follow. Programs only a few dozen lines long can be written and debugged in a fairly short time. *Debug* means to fix problems in programs. A debugger is a utility that assists the programmer in fixing the problems by allowing him to step through his program line by line and look at and modify variable values. If you run into a problem with a program, with the help of a debugger you may be able to single out a variable and see why it's causing you headaches.

The Turbo Pascal package includes a debugger from Apple called MacsBug. This debugger is difficult to use with Turbo for several reasons; for one, it is very difficult to locate code within it. MacsBug is a 68000-based debugger, so if you are not familiar with 68000 assembly code, it will be even harder to work with.

As an alternative, the TMON debugger from ICOM Simulations, Inc., is an excellent 68000-based debugger you can purchase separately. I will briefly discuss each of these debuggers later in the chapter, but for now look at methods of debugging your programs if you don't want to get into assembly code. The following few debugging tips will probably come in handy some day, and you will be able to solve some problems without a debugger.

First of all, closely monitor your program for such things as disk activity and screen display. For instance, you might notice that one string is making it to the screen but the next one is not, so you compare the code for the two strings to see what code executed between them may be causing a problem. Or maybe you're working with a file-based program that is supposed to read and write a file on one of your disks. If the disk never starts moving before the program bombs out, look at the code before the disk command statements. With just these small bits of knowledge you can more easily locate your problem.

If you're having problems with a particular variable and you're not sure what value it is holding, it is a good idea to use a few `Writeln` statements to display the variable at strategic locations. This is called echo printing because the value of the variable echoes on the screen. Although this process requires an extra compilation or two, it can save hours of distress. In fact, in writing this book I relied heavily on echo printing to solve several strange problems!

If you start working with very lengthy programs, you may find it convenient to have a debug routine you can call at any time to look at particular variables. This is a rather awkward way to fix problems, but once you get the procedure written, you insert calls to it only where you need them; you do not have to worry about duplicating large blocks of code and remembering to remove them once the problems are fixed.

If you are working with a file-building program and are able to achieve some disk activity, you can get some idea of whether something was actually written out by using the option `Get Info...` from the `File` menu on the desktop. Sometimes this is all you need to see whether anything was written to the file. On the other hand, you may wish to find out exactly what was written instead of just the size of the file. For this reason I have developed a file-dump program that allows you to look at most of your files to see exactly what they contain.

File Analysis

Let's take a look at this program:

Program HexDump;

{\$R+}

{\$I-}

Uses

MemTypes, QuickDraw, OsIntf, ToolIntf;

Var

i, j	: integer;	{counters}
fileName	: String[15];	{file to be dumped}
wait	: char;	{used in "Press any key..."}
inChar	: char;	{each character processed in file}
asciiArr	: array [1..16] of char;	{the array of characters to be printed}
hexArray	: array [0..15] of char;	{array of hex digits}
theFile	: text;	{the actual file to be read}

Procedure PrintHex;

Type

OverlayType = record	{variant for longint and char}
case boolean of	
TRUE: (aLong : longint);	{the longint portion}
FALSE: (aChar : char;	
bChar : char;	
cChar : char;	
dChar : char);	{the four-character portion}
end;	

Var

anOverlay	: OverlayType;	{the var used for overlaying}
loNib, hiNib	: char;	{the low and high nibbles to print}

begin {procedure PrintHex}

 anOverlay.aLong := BitAnd (\$00000000, \$00000000); {set all bits off}

 anOverlay.bChar := inChar;

 (*****)

 (* BitAnd so that only get lower 4 bits *)

 (*****)

 loNib := hexArray [BitAnd (\$0000000F, anOverlay.aLong)];

```

(*****
* BitShift 4 bits to get upper 4 bits *
(*****
anOverlay.aLong := BitAnd ($000000F0, anOverlay.aLong);
hiNib := hexArray [anOverlay.aLong shr 4];

(*****
* Now write them out *
(*****
write (hiNib, loNib, ' ');
end;                                     {procedure PrintHex}

begin                                     {main routine of HexDump}
hexArray[0] := '0';                       {initialize HexArray to hex digits}
hexArray[1] := '1';
hexArray[2] := '2';
hexArray[3] := '3';
hexArray[4] := '4';
hexArray[5] := '5';
hexArray[6] := '6';
hexArray[7] := '7';
hexArray[8] := '8';
hexArray[9] := '9';
hexArray[10] := 'A';
hexArray[11] := 'B';
hexArray[12] := 'C';
hexArray[13] := 'D';
hexArray[14] := 'E';
hexArray[15] := 'F';
write ('Please enter the file name:');
readln (fileName);
if (fileName = "") then
  fileName := 'Nameless';
reset (theFile, fileName, 1);             {buffer size of 1}
(*****
* If the file doesn't exist, display message *
(*****
if IOResult <> 0 then
  writeln (' NON-EXISTENT FILE ERROR!')
else
begin
  for j := 1 to 16 do
    asciiArr[j] := '.';                   {initialize array}

```



```

        close (theFile);
    end;                                     {else}
    writeln;
    writeln;
    write ('    Press any key to continue...');
    inChar := ReadChar;
end.                                         {program HexDump}

```

The hex-character dump program is fairly straightforward, but it introduces a few new concepts: text files, the function `BitAnd`, and the operator `shr`. You may have noticed that `TheFile` is declared as `text`. This is an easy way of declaring a file pointer to almost any type of file. I use `text` here because I want to be able to dump different types of files and I look at them on a byte-by-byte basis. You may have noticed the standard Pascal file routines `reset`, `EOF`, `read`, and `close`. This shows that programs can be written with Turbo Pascal using the standard procedures and can be ported to other Pascal compilers and interpreters. The function `BitAnd` simplifies converting characters to hexadecimal notation for output. `BitAnd` takes two parameters, both `longInt` types, and performs a logical AND on them. The result of a logical AND is determined by turning on the bits in both the parameters. For example, if I perform an AND on 7 and 14, I compare the binary values as follows:

$$\begin{array}{r}
 00000111 \quad \{7\} \\
 \underline{\text{AND } 00001110} \quad \{14\} \\
 00000110 \quad \{6\}
 \end{array}$$

The result is 6 because only the second and third rightmost bits are on in both 7 and 14.

In the program `HexDump` I use `BitAnd` to mask out 4 bits at a time from the byte in question. If I do a logical AND against the hex value `$0000000F`, the result will always be the 4 low-order bits of the value I AND it against. In this manner I can isolate the high- and low-order nibbles (a nibble is half a byte) of the byte we are converting.

The operator `shr` (shift/right) moves bits to the right in a value. Once I have isolated the high-order nibble in a character using `BitAnd` with `$000000F0`, I need to shift the bits to the right four slots so as to finish the conversion. The index:

```
[ anOverlay.aLong shr 4 ]
```

takes the current value of `anOverlay.aLong` and returns a new value in which all the on bits are shifted 4 positions to the right. The opposite of this is `shl`

(shift left), which may be used to shift bits left by a specified number of positions.

When you run the program, you are prompted to enter the name of the file to examine. The file is displayed in both ASCII and the character translation. For example, if you enter the file

```
Hi there.
This is what a file dump looks like.
As you can see, even numbers look kind of funny!
1234567890
But, this is a very good debugging tool for files.
```

Its file dump looks like this:

```
48 69 20 74 68 65 72 65 2E 0A 54 68 69 73 20 69 Hi there..This i
73 20 77 68 61 74 20 61 20 66 69 6C 65 20 64 75 s what a file du
6D 70 20 6C 6F 6F 6B 73 20 6C 69 6B 65 2E 0A 41 mp looks like..A
73 20 79 6F 75 20 63 61 6E 20 73 65 65 2C 20 65 s you can see, e
76 65 6E 20 6E 75 6D 62 65 72 73 20 6C 6F 6F 6B ven numbers look
20 6B 69 6E 64 20 6F 66 20 66 75 6E 6R 79 21 0A kind of funny!.
31 32 33 34 35 36 37 38 39 30 0A 42 75 74 2C 20 1234567890. But,
74 68 69 73 20 69 73 20 61 20 76 65 72 79 20 67 this is a very g
6F 6F 64 20 64 65 62 75 67 67 69 6E 67 20 74 6F ood debugging to
6F 6C 20 66 6F 72 20 66 69 6C 65 73 2E 0A ol for files..
```

Each line of the dump represents 16 bytes of the file, first in ASCII and second in the converted-character representation. Remember, ASCII is a method of representing characters that is recognizable by computers. The hexadecimal ASCII conversion values for printable characters are as follows:

<u>Hex Value</u>	<u>Character</u>	<u>Hex Value</u>	<u>Character</u>
20	(space)	2A	*
21	!	2B	+
22	"	2C	,
23	#	2D	—
24	\$	2E	.
25	1/2	2F	/
26	&	30	0
27	'	31	1
28	(32	2
29)	33	3

Continued

<u>Hex Value</u>	<u>Character</u>	<u>Hex Value</u>	<u>Character</u>
34	4	59	Y
35	5	5A	Z
36	6	5B	[
37	7	5C	/
38	8	5D]
39	9	5E	^
3A	:	5F	_
3B	;	60	`
3C	<	61	a
3D	=	62	b
3E	>	63	c
3F	?	64	d
40	@	65	e
41	A	66	f
42	B	67	g
43	C	68	h
44	D	69	i
45	E	6A	j
46	F	6B	k
47	G	6C	l
48	H	6D	m
49	I	6E	n
4A	J	6F	o
4B	K	70	p
4C	L	71	q
4D	M	72	r
4E	N	73	s
4F	O	74	t
50	P	75	u
51	Q	76	v
52	R	77	w
53	S	78	x
54	T	79	y
55	U	7A	z
56	V	7B	{
57	W	7C	
58	X	7D	}

Analysis of files with this program shows exactly how much information is written to the file on a byte-by-byte basis until the end of the file is reached. You may have noticed that there are several dots (':') in the character translation that were not in the original file. The dots represent nonprintable

characters and those that are not easily represented on the screen. If you are dumping a file that starts to scroll off the screen, you can press any key to freeze the screen and take a closer look at that portion of the dump. To continue, just press a key other than Q. If you do press Q while the screen is stopped, the program calls the procedure Halt. Halt stops the program and returns you to the Macintosh desktop or the Turbo Pascal editor, depending upon how your program was invoked.

As you can see, this dump utility is quite useful in many applications. Now look at a program that allows you to encipher data files so that nobody else can read them.

Encipher/Decipher Program

Try the following program:

Program CiphDecip;

{\$I-}

Var

selection	: char;	{user's selection}
valid	: boolean;	{is the input valid?}
inFileName	: String[15];	{name of the input file}
outFileName	: String[15];	{name of the output file}
inFile	: text;	{file pointer for input}
outFile	: text;	{file pointer for output}
aChar	: char;	{a character in the file}

Procedure Encipher;

```
begin                                     {procedure Encipher}
  ClearScreen;
  write ('What is the name of the text file?');
  readln (inFileName);
  if (inFileName = "") then
    inFileName := 'NoNameInput';
  writeln;
  write ('What is the name of the enciphered file?');
  readln (outFileName);
  if (outFileName = "") then
    outFileName := 'NoNameOutput';
  reset (inFile, inFileName, 1);          {buffer size of 1}
```

```

if IOResult <> 0 then                                {flag as an error}
begin
  writeln;
  writeln ('    NON-EXISTENT INPUT FILE!');
end
else
begin
  rewrite (outFile, outFileNamE);
  while NOT EOF (inFile) do
  begin
    read (inFile, aChar);
    aChar := CHR (ORD (aChar) + 1); {add one to encipher data}
    write (outFile, aChar);
  end;
  close (inFile);
  close (outFile);
  writeln;
  writeln ('    YOUR FILE HAS NOW BEEN ENCIPHERED!');
end;
writeln;
write ('    Press any key to continue...');
selection := ReadChar;
end;                                                    {procedure Encipher}

Procedure Decipher;

begin                                                    {procedure Decipher}
  ClearScreen;
  write ("What is the name of the enciphered file?");
  readln (inFileName);
  if (inFileName = "") then
    inFileName := 'NoNameOutput';    {name of nameless output in Encipher}
  writeln;
  write ("What is the name of the deciphered file?");
  readln (outFileName);
  if (outFileName = "") then
    outFileName := 'NoNameInput';    {name of nameless input in Encipher}
  reset (inFile, inFileName, 1);      {buffer size of 1}
  if IOResult <> 0 then
  begin
    writeln;
    writeln ('    NON-EXISTENT INPUT FILE!');
  end
  else

```

```
begin
  rewrite (outFile, outfileName);
  while NOT EOF (inFile) do
  begin
    read (inFile, aChar);
    aChar := CHR (ORD (aChar) - 1); {subtract one to decipher data}
    write(outFile, aChar);
  end;
  writeln;
  writeln('    YOUR FILE HAS BEEN DECIPHERED!');
  close (inFile);
  close (outFile);
end;
writeln;
write('    Press any key to continue...');
selection := ReadChar;
end;                                     {procedure Decipher}

begin                                     {program CiphDecip}
  valid := FALSE;
  while NOT valid do
  begin
    ClearScreen;
    writeln ('    Please select the option you would like to do:');
    writeln ('    1. Encipher a text file');
    write ('    2. Decipher an enciphered file ');
    selection := ReadChar;
    case selection of
      '1': begin
        valid := TRUE;
        Encipher;
      end;
      '2' : begin
        valid := TRUE;
        Decipher;
      end;
    otherwise
      begin
        writeln ('    INVALID RESPONSE...PLEASE TRY AGAIN!');
        writeln;
        write ('    Press any key to continue...');
        selection := ReadChar;
      end;
    end;
  end;
end;
```

```

        end;
    end;                                {case Selection of}
end;                                    {while NOT valid}
end.                                    {program CiphDecip}

```

The basic premise behind the encipher program is to add 1 to every byte in the file and write the results out to a file. In other words, all spaces appear as \$21, capital A appears as \$42, and so on. The output file will hold your enciphered file and no one will be able to read it, without the algorithm behind the encipher program. When you need to view your file, you run the decipher option. The program can easily be modified to allow you to delete the original file or a deciphered file to a file. For those who are interested in working with the 68000-based debuggers available for Turbo Pascal programs, the next sections discuss the use of MacsBug and TMON.

MacsBug

In the Misc Folder on the Turbo Pascal Utilities and Sample Programs disk is a file called MacsBug; this is the debugger Borland provides for use with Turbo Pascal. MacsBug is intended for use with assembly-language programs and compilers that provide an intermediate step between high-level languages and 68000, generating assembly-language listings. MacsBug can debug Turbo programs provided patience and a fairly good knowledge of 68000 assembly language.

To begin with, copy the MacsBug file into the System Folder of the boot disk you will use when you debug. When you boot from that disk, the regular "Welcome to Macintosh" screen will say at the bottom of the box that MacsBug is installed. From this point on you may never notice that you have installed MacsBug because no other screens are affected.

You may invoke the debugger in one of three ways: first, by pressing the interrupt button on the programmer's switch on the rear left side of the Macintosh; a second, by placing a call to MacsBug. In order to do this, you'll have to explain to Turbo that MacsBug is a routine defined like this:

```
Procedure MacsBug;    inline $A9FF;
```

If you place that declaration anywhere between the program declaration and the beginning of the main routine, you may invoke MacsBug simply by calling the procedure. The final method of invoking MacsBug is actually a safety measure; if a system error occurs while MacsBug is installed, MacsBug will automatically kick in to gear. This way, if your program does bomb out,

at least you can look around at variables and instruction paths to get some information on the crash.

Before you use MacsBug, compile with \$D, which generates debug symbols, and compile the program to disk besides. \$D+ allows you to look for your procedure and function names in the assembly listing while you are working in the MacsBug environment. This should help a bit in pinpointing specific instructions, but you still must have a good understanding of 68000 assembly code to see how the Turbo statements are translated. You should compile your program to disk and start it up by double-clicking on it because memory is set up differently when you run your program from within the Turbo environment.

When you do invoke MacsBug, your screen will clear and the MacsBug screen will take over. The instruction about to be executed will be displayed along with all 68000 registers—A0 through A7, D0 through D7, the program counter (PC), and the status register (SR). At this point you may start interacting with the debugger by entering the MacsBug commands listed in your Turbo Pascal manual. Below is a brief description of some of the more useful commands:

- ? Provides help with the MacsBug debugger.
- DV Displays the MacsBug version number.
- RB Reboots the system from within MacsBug.
- EA Exits to the application running when MacsBug was invoked.
- DM *[[address] number]* Displays memory at the address provided for the number of bytes specified.
- TD Provides a total display of the registers.
- BR *[address]* Sets a breakpoint at the address specified.
- G Continues execution of the program.
- CL Removes all breakpoints (or selective breakpoints if CL *[address]*).
- S Single-step execution of the program.
- HD Provides a complete dump of heap information.
- IL *[address [number]]* Disassembles memory beginning at specified address for number of bytes listed.
- Dn *[value]* The Dn (D0, D1, D2) by itself will display the value of the specified register. If a value is placed after the register name (for example D1 FFFFFFFF), that value is plugged into the register.
- An *[value]* Same as Dn *[value]* described above except works with address registers.

Several other instructions are available in MacsBug. They are briefly described in the Turbo Pascal manual. If you are not familiar with the

workings of 68000 assembly, I suggest you read a book or two on the subject (such as *The Complete Book of Macintosh Assembly Language Programming*, Volumes I and II from Scott, Foresman and Company). Even with a good understanding of assembly language, MacsBug is rather difficult to use. A more powerful debugger, TMON, may be purchased separately from ICOM Simulations, Inc. Take a look at some of TMON's capabilities.

TMON

A 68000-based debugger for the Macintosh, TMON may be purchased separately at a local computer store. If you have trouble finding it, you can order it, at this address:

ICOM Simulations, Inc.
648 S. Wheeling Road
Wheeling, IL 60090

or call (312) 520-4440.

Unlike MacsBug, TMON has its own small installation procedure, detailed in the manual that comes with the debugger. Once you have installed TMON, you may invoke it by pressing the interrupt button on the programmer's switch. This is where the real differences between MacsBug and TMON become apparent.

Window-based TMON also uses the Macintosh menu environment, albeit in a rather bare-bones manner. You may have several debug windows open at a time so as to look at assembly instructions, view the heap, set breakpoints, and so forth. Each window may be opened by selecting the appropriate item from the TMON menu bar. Rather than restricting you to the conventional debugger interaction of entering a command and awaiting a response from the computer so that you may enter another, TMON allows you to modify instructions, registers, and so on. To do so place the cursor via the mouse over the item to be changed, select it just like text in the Turbo Pascal editor, and modify it. This debugger truly uses the Macintosh-style interface and is a pleasure to use. In short, you can do everything in TMON that you can do in MacsBug, do a good deal more, and because it is so easy to use, probably go further with TMON than with MacsBug. TMON is an excellent package, used in many software houses that write Macintosh programs. You still need to know how 68000 assembly language works—a handy 68000 reference booklet is included with the TMON package—so there is no difference in this regard between MacsBug and TMON, but we believe TMON is a better product.

Review Summary

1. A debugger is a utility that may allow you to view your program's execution line by line, look at, and modify variables.
2. Echo printing is a debugging method showing variable values at strategic locations within your code.
3. The Turbo Pascal package includes the MacsBug compiler from Apple, but the TMON debugger is much easier to use and more powerful than MacsBug.

Quiz

1. Why is it sometimes beneficial not to have access to a sophisticated debugger?
2. What does the following ASCII string say?

43 20 49 53 20 46 55 4E

3. What is the result using BitAnd on the following values?
A. 14 AND 3 C. 15 AND 15
B. 8 AND 2 D. 3 AND 12

Graphics, Sound, and Resources

Graphics

Turtle Graphics

Standard Macintosh Graphics

Fun with the Mouse

Making Music In Turbo Pascal

Resources

Using RMaker

Event-Handling Programming with a Resource File

Review Summary

Quiz

In this chapter you will learn:

- How to work with some popular graphics routines available to you in Turtle and QuickDraw.
- About the unique environment used to work with QuickDraw routines in Turbo Pascal.
- How to design programs that use the mouse-oriented routines.
- How to write and store songs using a music library routine.
- How to use resource files and RMaker.
- How to write event-driven programs on the Macintosh.

Graphics

As you probably know, the Macintosh is well known for its graphics. In fact, the level of graphics attainable on the Macintosh is unsurpassed in the microcomputer industry today. One of the reasons is the high number of pixels, or dots, per square inch on the Macintosh screen; the more dots in a given area, the crisper the resulting picture.

In Turbo Pascal you have access to Turtle and QuickDraw, which are compilations of graphics routines. For various reasons these routines are much faster and probably easier to use than most graphics routines you could develop yourself. Take a look at the graphics available to you in Turtle.

Turtle Graphics

Turbo Pascal offers Turtle, which allows you to create simple graphics without much effort. Turtle's routines let you draw figures in a manner similar to the Logo language's turtle graphics. Unlike many versions of Logo, Turbo has no cute little turtle or triangle showing the drawing tool, or pen in Turbo Pascal. When you use Turtle, the screen is treated as a grid whose home position (0,0) is in the center. Positive values on the horizontal, or x, axis take you to the right across the screen. Positive values on the vertical, or y, axis take you up the screen. You may draw on your screen with Turtle routines by specifying a location or by directing the pen and saying how far you want it to travel in that direction. There are thirteen procedures and three functions available in the Turtle unit; here is a brief description of each;

Back(Distance)

This procedure moves the pen backward the distance specified. A negative distance will move the pen forward.

Clear

This procedure clears the window and moves the pen to the home position (0,0) in the middle of the screen.

Forwd(Distance)

The opposite of Back(Distance), it moves the pen in the direction it is pointing for a specified Distance. If distance is negative, the pen will move backward.

Direction := Heading

The function Heading returns an integer value in the range 0 to 359 that specifies the angle at which the pen is pointing. The value 0 means that the pen is pointing directly up, and any other value is the angle in degrees clockwise from the vertical.

Home

The procedure Home places the pen in the home position in the window and points it straight up.

NoWrap

NoWrap keeps the pen from wrapping around from one side of the window to the other when it has gone outside the boundaries of the window.

PenDown

PenDown activates the pen so that drawing may begin.

PenUp

The opposite of PenDown, PenUp allows movement of the pen without drawing.

SetHeading(Angle)

SetHeading points the pen in the direction specified by Angle. As with Heading, an angle of 0 is straight up, and angles increase clockwise from 0. Four constants have been declared in Turtle:

<u>Constant Name</u>	<u>Value</u>
North	0
East	90
South	180
West	270

SetPosition(x, y)

SetPosition lifts the pen and positions it at point (x, y).

TurnLeft(Angle)

TurnLeft rotates the pen counterclockwise for positive angles and clockwise for negative angles.

TurnRight(Angle)

TurnRight rotates the pen clockwise for positive angles and counterclockwise for negative angles.

TurtleDelay(Time)

TurtleDelay may be used to set up a timing delay between operations; time is in milliseconds.

Wrap

The opposite of NoWrap, Wrap allows the pen's drawing action to wrap from one side of the window to the other when the pen goes outside the boundary of the window.

Location := XCor

The function XCor returns an integer value that specifies the horizontal position of the pen.

Location := YCor

The function YCor returns an integer value that specifies the vertical position of the pen.

For an example of some of the more useful Turtle routines, try the following program:

```
Program TheTurtle;
```

```
Uses
```

```
  MemTypes, QuickDraw, OSIntf, ToolIntf, Turtle;
```

```
Var
```

```
  aChar : char;           {used for "Press any key..."}  
  i      : integer;
```

```
begin
```

```
  SetPosition (-200, 125);  
  SetHeading (East);  
{program TheTurtle}
```

```

(*****
(* First draw a rectangle. *)
(*****
Forwd (150);
TurnRight (90);
Forwd (75);
TurnRight (90);
Forwd (150);
TurnRight (90);
Forwd (75);
(*****
(* Now draw a circle. *)
(*****
SetPosition (100, 90);
for i := 1 to 125 do
begin
  TurnRight (3);
  Forwd (2);
end;

(*****
(* Now draw a triangle *)
(*****
SetPosition (0, 10);
SetHeading (225);
Forwd (100);
SetHeading (East);
Forwd (142);
SetHeading (315);
Forwd (100);
GotoXY (27, 23);
write ('Press any key to continue...');
aChar := ReadChar;
end.

```

The output of this program, which draws a rectangle, circle, and triangle, is shown in Figure 13.1. Although no routines in the Turtle environment are explicitly used to make curved lines, you can draw curves, circles, and so on with creative code like this, which draws the circle:

```

for i := 1 to 125 do
begin
  TurnRight ( 3 );
  Forwd ( 2 );
end;

```

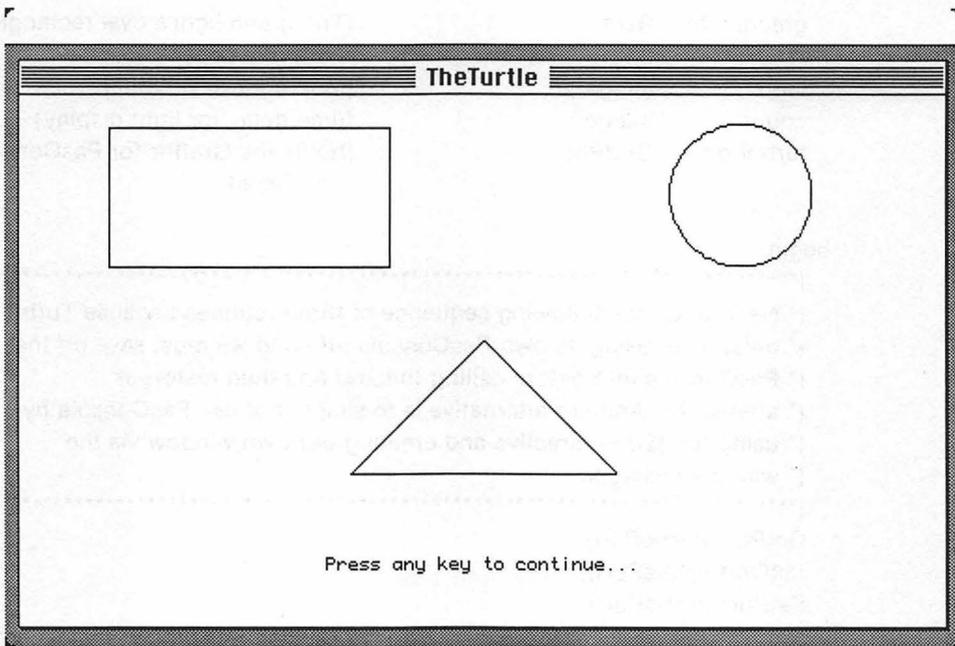


Fig. 13.1.

For more sophisticated drawings you will probably want to use QuickDraw.

Standard Macintosh Graphics

The better to understand the concepts behind QuickDraw graphics in Turbo Pascal, look at the following program:

Program StreetLight;

```
(*****
(* This program draws a streetlight on the standard output screen *)
(* and flashes the three lights to different shades. *)
(*****)
```

Uses

MemTypes, QuickDraw;

Var

```
lightBox : Rect;           {The light box rectangle coordinates}
redLight  : Rect;         {The red light's oval rectangle
                           coordinates}
yellowLight : Rect;      {The yellow light's oval rectangle
                           coordinates}
```

```

greenLight : Rect;           {The green light's oval rectangle
                              coordinates}
loop        : integer;      {loop control variable}
count       : integer;      {time delay for light display}
turboPort   : GrafPtr;      {holds the GrafPtr for PasConsole
                              routines}

```

```
begin
```

```

(*****
(* Need to do the following sequence of three routines because Turbo   *)
(* defaults to using its own PasConsole info and we must save off the  *)
(* PasConsole info before calling InitGraf and then restore it        *)
(* afterwards. Another alternative is to simply not use PasConsole by  *)
(* using the {$U-} directive and creating our own window via the      *)
(* window manager.                                                    *)
(*****
GetPort (turboPort);
InitGraf (@thePort);
SetPort (turboPort);

HideCursor;
SetRect (redLight, 228, 81, 253, 106);  {set up all the rectangles}
SetRect (yellowLight, 228, 112, 253, 137);
SetRect (greenLight, 228, 143, 253, 168);
SetRect (lightBox, 216, 75, 266, 175);
FrameRect (lightBox);
FillRect (lightBox, LtGray);
for loop := 1 to 100 do
begin
  if odd (loop) then
    PenPat (black)           {change the pen pattern to black}
  else
    PenPat (white);
  PaintOval (redLight);     {paint the red light; black}
  for count := 1 to 5000 do
  ;
  if odd (loop) then
    PenPat (gray)          {change the pen pattern to gray}
  else
    PenPat (white);
  PaintOval (yellowLight);  {paint the yellow light; dark gray}
  for count := 1 to 5000 do
  ;
  if odd (loop) then
    PenPat (dkGray)        {change to pen pattern to dark gray}

```

```

else
  PenPat (white);
  PaintOval (greenLight);           {paint the green light; gray}
  for count := 1 to 5000 do
  ;
end;                                 {for loop := 1 to 100}
ShowCursor;
end.

```

As you can see, the **uses** statement informs Turbo that I wish to use QuickDraw. The first item you have not seen before is Rect. In QuickDraw the rectangle is defined as follows:

```

Rect = record
  case integer of
    0 : ( top, left, bottom, right : integer );
    1 : ( topLeft, botRight : point );
  end;

```

Further, a point is defined in QuickDraw thus:

```

VHSelect = (v, h);
Point = record
  case integer of
    0 : (v, h : integer );
    1 : (vh : array[ VHSelect ] of integer );
  end;

```

structure is passed to several different QuickDraw routines and is used to specify a particular rectangle on the screen.

The next unfamiliar type you will see is GrafPtr, which points at a GrafPort record structure. The GrafPort record structure contains a good deal of information controlling how QuickDraw operates. For a listing of the complete GrafPort type, refer to the description of QuickDraw in the Turbo Pascal manual. I must declare the variable TurboPort as a GrafPtr type for the following sequence of statements:

```

GetPort ( turboPort );
InitGraf ( @thePort );
SetPort ( turboPort );

```

As mentioned in the commented code, I need these three procedures because I am using the standard PasConsole output with QuickDraw. Generally, the

PasConsole output window is not used with QuickDraw, but it is indeed possible. Any time QuickDraw routines are used, you must first perform some initialization via InitGraf. ThePort is declared in QuickDraw as a GrafPtr, and its address passes to InitGraf for subsequent use by QuickDraw routines. Because I am using the PasConsole window, I first perform a GetPort, which retrieves the port using the variable TurboPort. Next I invoke InitGraf to perform the necessary initializations for QuickDraw. Finally I call SetPort to restore turboPort as the active port. It is not all that important to understand why these three procedure calls are necessary. Just remember that they must be invoked in that order whenever you use QuickDraw routines with the PasConsole output window. After I discuss the rest of the new routines in this program, I will show how to do the same program without the PasConsole output window.

The next procedure, HideCursor, does just that: it hides the cursor so that it can't be seen on the screen even if you move the mouse around. A call at the end of the program to ShowCursor restores it.

After HideCursor, I set up the coordinates for the light box itself and the three lights inside it via SetRect, whose first parameter is a Rect type, and the next four parameters are the left, top, right, and bottom coordinates respectively. I can set up the rectangle coordinates with simple assignment statements, but SetRect combines four assignment statements in one procedure call, minimizing the source code.

I draw the light box by calling FrameRect with the LightBox as a parameter. FrameRect draws a rectangle with the coordinates of the parameter Rect. FillRect then fills in the light box with light gray, one of several shading constants defined in QuickDraw.

The last section of program StreetLight is a for loop that changes the shading of the lights based on the function odd. For example, every time the loop variable is odd, the pen pattern for the red light is set to black via the PenPat routine. Next, PaintOval is called with redLight as the parameter. The routine PaintOval will draw an oval circumscribed by the parameter Rect, redLight in this case. Since the pen pattern was changed with PenPat, the redLight is drawn with black shading. On even iterations of the for loop the pen pattern is set to white and the redLight is drawn with white shading. The same logic is used to draw the yellow light with either gray or white shading and the greenLight with either dark gray or white shading. After each light has been drawn, a timing delay loop from 1 to 5000 slows the program down so that the color changing does not happen too rapidly. The outer for loop shows one way to perform simple animation on the Macintosh screen.

In order to rewrite the program without the PasConsole output window, I create my own window via the Macintosh window manager. Here is how to rewrite the program with its own output window:

Program StreetLight;

{\$U—}

```
(*****
(* This version of StreetLight doesn't use the          *)
(* standard PasConsole routines ({U—}) and uses        *)
(* QuickDraw to open an output window.                *)
(*****)
```

Uses

MemTypes, QuickDraw, OsIntf, ToolIntf;

Var

```
lightBox      : Rect;           {The light box rectangle coordinates}
redLight      : Rect;           {The red light's oval rectangle
                                coordinates}
yellowLight   : Rect;           {The yellow light's oval rectangle
                                coordinates}
greenLight    : Rect;           {The green light's oval rectangle
                                coordinates}
loop          : integer;        {loop control variable}
count         : integer;        {time delay for light display}
windowRect    : Rect;           {Rect definition for output window}
behind, wPtr  : WindowPtr;      {window screen location and the pointer
                                to it}
```

begin

```
  InitGraf (@thePort);
  InitWindows;
```

```
  HideCursor;
```

```
  SetRect (windowRect, 5, 50, 500, 300); {left, top, right, bottom}
  behind := POINTER (-1);                 {sets up window to be in front of all
                                          others}
```

```
  WPtr := NewWindow (nil, windowRect, 'Our StreetLight Output Window',
    TRUE, NoGrowDocProc, behind, FALSE, 0);
```

```
  SetRect (redLight, 228, 81, 253, 106);
  SetRect (yellowLight, 228, 112, 253, 137);
  SetRect (greenLight, 228, 143, 253, 168);
  SetRect (lightBox, 216, 75, 266, 175);
  FrameRect (lightBox);
  FillRect (lightBox, LtGray);
  for loop := 1 to 100 do
```

```

begin
  if odd (loop) then
    PenPat (black)                {change the pen pattern to black}
  else
    PenPat (white);
    PaintOval (redLight);         {paint the red light; black}
    for count := 1 to 5000 do
    ;
  if odd (loop) then
    PenPat (gray)                 {change the pen pattern to gray}
  else
    PenPat (white);
    PaintOval (yellowLight);     {paint the yellow light; dark gray}
    for count := 1 to 5000 do
    ;
  if odd (loop) then
    PenPat (dkGray)              {change to pen pattern to dark gray}
  else
    PenPat (white);
    PaintOval (greenLight);      {paint the green light; gray}
    for count := 1 to 5000 do
    ;
end;                               {for loop := 1 to 100}
ShowCursor;
DisposeWindow (wPtr);
end.

```

This version of the program executes like the first one except that the title now displayed in the output window is **Our StreetLight Output Window**. I included `OsIntf` and `ToolIntf` in the `uses` statement because the window manager routines are defined in `ToolIntf`. I have a few new variable declarations, including `windowRect`, a `Rect` definition of the output window. I also declared a couple of `WindowPtr` types: `behind` and `wPtr`. `WindowPtr` is nothing more than a `GrafPtr` with its own definition for clarity.

Immediately after our call to `InitGraf` the procedure `InitWindows` is invoked. `InitWindows` must be called before any other window manager routines. The cursor is hidden and a call to `SetRect` defines the boundaries of the window. Next parameters are set up for the call to the routine `NewWindow`. The first parameter, `nil`, instructs `NewWindow` to allocate the storage space for our window. Next `windowRect` defines the boundaries of the window to be drawn. I pass the title to be displayed at the top of the window and set a Boolean flag to be `TRUE` if the window is to be visible and `FALSE` if not. Next the parameter `NoGrowDocProc` instructs `NewWindow` to draw the

window without a grow box, the little box in the bottom right corner used for resizing windows. The parameter behind is next; it says whether or not this window will be displayed behind other windows. I set behind equal to the result of Pointer (-1), so NewWindow will place our window on top of all others. This is no big deal here, since this window will be the only one on the screen, but it is an important parameter for programs that display multiple windows. The next parameter, FALSE, tells NewWindow to close the window with a go-away box in the upper right corner. Finally, a reference constant of 0 is passed to designate this window as number 0. The call to NewWindow draws the window, and if you go on, the rest of the program is the same as the earlier version except that I perform a DisposeWindow on wPtr. This procedure frees up the memory space used by the window.

NewWindow takes loads of parameters and is rather bulky to use for several windows. Later in this chapter I will show you how to use another window-creation routine, GetNewWindow, requires a resource file but fewer parameters.

Fun with the Mouse

The Macintosh mouse can be used in several different ways in Pascal. Here I present a few useful routines that allow you to know the status of the mouse's button and the position of the cursor on the screen.

The following program, TheButton, provides you with the mechanism to determine whether the mouse's button is being pressed or not. Try running this program so that you can see what sort of value the Button routine returns:

```

Program TheButton;
(*****
(* This program shows how the functions Button and      *)
(* KeyPressed work.                                     *)
(*****)

Uses
  MemTypes, QuickDraw, OsIntf, ToolIntf;

Var
  c : char;

begin
  {program TheButton}
  GotoXY (10, 5);
  write ('The value of the Button Function is:');
  GotoXY (10, 10);

```

```

writeln ('(To stop, press a key on the keyboard)');
while NOT (KeyPressed) do
begin
  GotoXY (47, 5);
  if Button then
  begin
    write ('TRUE (Press any key to continue)');
    (*****
    (* Wait until a key is pressed so          *)
    (* that the TRUE message can be          *)
    (* displayed.                              *)
    (*****
    repeat
    until KeyPressed;
    c := ReadChar;
  end
  else
    write ('FALSE ');
end;
end.
                                     {while NOT (KeyPressed) do}
                                     {program TheButton}

```

As you press the mouse button, the value TRUE is displayed and you are instructed to press a key to continue. When you press a key, the value FALSE is displayed. The routine Button returns a Boolean value, which you may use to determine any further action, for instance wait until the user presses the button.

If you enjoy drawing on the Macintosh, you'll appreciate the following mouse-based program, which allows you to create images in the standard output window. When you run this program, press the mouse button while moving it around to draw freehand:

Program Draw;

```

(*****
(* This program allows the user to make drawings on      *)
(* the standard output screen. It also has a few        *)
(* commented statements, which when un-commented       *)
(* add certain drawing characteristics.                 *)
(*****

```

Uses

MemTypes, QuickDraw, OSIntf, ToolIntf;

```

Var
  loop      : integer;           {loop control variable}
  count     : integer;           {time delay for light display}
  horizontal : integer;           {the horizontal position of the cursor}
  vertical  : integer;           {the vertical position of the cursor}
  dummy     : integer;           {temporarily holds horizontal point}
  turboPort : GrafPtr;           {holds GrafPtr for PasConsole routines}
  thePt     : Point;             {position of mouse}
  done      : boolean;           {are we finished drawing yet?}

begin                                {program Draw}
  (*****)
  (* Set up so that we can use the QuickDraw routines      *)
  (* in the standard console window.                        *)
  (*****)
  GetPort (turboPort);
  InitGraf (@thePort);
  SetPort (turboPort);

  done := FALSE;
  while NOT done do
  begin
    repeat
      if KeyPressed then
        done := TRUE;
      until ((Button) OR (done));
      if NOT done then
      begin
        GetMouse (thePt);
        MoveTo (thePt.h, thePt.v);
        while Button do
        begin
          GetMouse (thePt);
          LineTo (thePt.h, thePt.v);
          (* LineTo (thePt.h + 1, thePt.v + 1); *)
          (* LineTo (thePt.h + 5, thePt.v + 5); *)
          (* dummy := thePt.h - 255; *)
          (* thePt.h := 255 - dummy; *)
          (* LineTo (thePt.h, thePt.v); *)
          end;                                {while Button do}
        end;                                {if NOT done then}
      end;                                {while NOT done do}
    end.                                {program Draw}
  
```

This program uses a while loop to allow you to draw freehand on the screen using the mouse until a key is pressed on the keyboard.

GetMouse returns the coordinates in the ThePt parameter of the cursor so that the drawing pen may be moved with the MoveTo routine. Then, while the button is being held down (while Button do), GetMouse determines where to draw a line using the routine LineTo. All three procedures are easily used together, since their parameters have the same meaning.

After you have drawn with this configuration of the program, remove the comment marks (* and *) from the first of the five commented lines and see how this affects drawing. Then delete that line and remove the comment marks from the next statement. These lines accentuate your freehand movements by drawing additional lines 1 and 5 pixels respectively from your current position. Finally, delete the previous line and remove the comments from the last three commented lines. Then see how easy it is to make mirror images with Turbo Pascal. Figure 13.2 shows an example of what you can draw using the last three commented lines:

The many other QuickDraw routines would take an entire book to explain; they are listed in the Turbo Pascal manual. At this point you should be able to draw in Pascal by applying the pens, rectangles, paint, and so on that have been discussed here. But dazzling graphics aren't the only fun thing you can

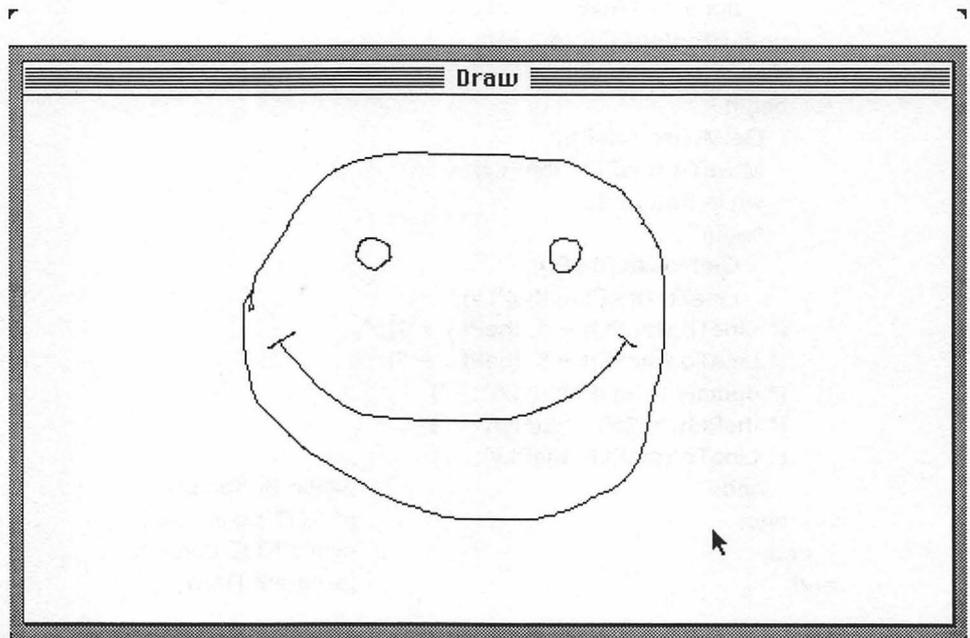


Fig. 13.2.

do on the Macintosh; Turbo Pascal can generate sound, which you may wish to incorporate with your programs. Look at how to make the Mac sing.

Making Music in Turbo Pascal

In order to discuss how to make music on the Macintosh, I must first discuss data types. The following three type declarations, which are found in `OSIntf`, are the only ones you need to produce music on the Mac:

Type

```
Tone = record
```

```
  count    : integer;
```

```
  amplitude : integer;
```

```
  duration  : integer;
```

```
end;
```

```
Tones = array[0..5000] of Tone;
```

```
SWSynthRec = record
```

```
  mode : integer;
```

```
  triplets : Tones;
```

```
end;
```

The type `Tone` declares the basic items that make up a musical tone: count (or frequency), amplitude (or intensity), and duration. The array `Tones` sets up an array of up to 5001 different tone records, and `SWSynthRec`—for Square Wave Synthesizer Record—allows me to set up one record that may contain an entire song. Square wave synthesizer tone generation is just one way to produce sophisticated sounds on the Macintosh. It is one of the easiest ways to play simple songs, so I will work with it exclusively.

In order to play a simple song on the Macintosh, enter the various notes into the `triplets` field of a `SWSynthRec` variable, set its `mode` to the constant `SWMode`, declared in `OSIntf`, and call the procedure `StartSound`, whose syntax is

```
StartSound ( PtrToSynthRec, RecSize, NextProc );
```

`PtrToSynthRec` is the address of the `SWSynthRec` variable, `RecSize` is the block of tones to play, and `NextProc` is a pointer to the procedure to perform when all sounds have been produced. We will pass the result of the `Pointer (-1)` function in the `NextProc` to produce synchronous sounds.

As we mentioned above, the amplitude field of Tone indicates the intensity of the note. The count field refers to the note's frequency, and the chart below shows the frequencies of notes from one octave below middle C to one octave above it.

<u>Note</u>	<u>Count</u>	<u>Note</u>	<u>Count</u>
C	262	C (Middle)	523
C Sharp	277	C Sharp	554
D	294	D	587
D Sharp	311	D Sharp	622
E	330	E	659
F	349	F	698
F Sharp	370	F Sharp	740
G	392	G	784
G Sharp	415	G Sharp	831
A	440	A	880
A Sharp	466	A Sharp	923
B	494	B	988

In order to play an actual song it is almost a necessity to build the song in a file first. The following program builds a song file based on the input for file name, frequencies, and durations:

Program SongBuilder;

Uses

MemTypes, QuickDraw, OSIntf;

```
(*****
(* This program may be used to create "song" files which  *)
(* will then be used in a song-playing program to create  *)
(* Square Wave music via Turbo Pascal.                    *)
(*****)
```

Var

```
nameOfFile : String;           {song file name}
done       : boolean;         {are we finished?}
songFile   : file of Tone;    {file of notes (song)}
eachNote   : Tone;           {var for each note}
moreNotes  : char;           {user's response (Y/N)}
valid      : boolean;        {is this a valid response?}
```

```
begin                                     {program SongBuilder}
  write ('Please enter your song file name... ');
  readln (nameOfFile);
  if (nameOfFile = "") then
    nameOfFile := 'Nameless Song';
  rewrite (songFile, nameOfFile);
  done := FALSE;
  while NOT done do
  begin
    writeln;
    write ('What is the note"s frequency? ');
    readln (eachNote.count);
    eachNote.amplitude := 200;           {set to 200 so don't have to keep
                                          entering!}

    writeln;
    write ('What is the note"s duration? ');
    readln (eachNote.duration);
    write (songFile, eachNote);
    valid := FALSE;
    while NOT valid do
    begin
      writeln;
      write ('Are there any more notes (Y/N)? ');
      moreNotes := ReadChar;
      case moreNotes of
        'Y', 'y' : valid := TRUE;
        'N', 'n' : begin
          valid := TRUE;
          done := TRUE;
        end;

        otherwise
          begin
            writeln;
            writeln ('INVALID RESPONSE...PLEASE TRY AGAIN!');
          end;
      end;
    end;                                     {case moreNotes}
  end;                                     {while NOT valid}
end;                                     {while NOT done}
close (songFile);
end.                                     {program SongBuilder}
```

The program's logic is fairly straightforward, and there are no new concepts in it. It is simply a file that holds all the information for the notes to

the song of your choosing. The intensity or amplitude is set to 200 (each-Note.amplitude := 200;) so that you don't have to keep on entering a number. You may want to remove this and insert code to give each note a unique intensity. Try entering the following notes and durations for a song:

<u>Note</u>	<u>Duration</u>
131	15
311	15
294	15
262	15
247	15
415	15
392	15
349	15
311	15
622	15
587	15
523	15
494	15
587	15
415	15

All the durations are 15, which means that each note is played for exactly the same amount of time. Once you build the song file with this program, you can play it by running the following program and specifying the name of the SongBuilder file.

Program PlaySong;

```
(*****
(* This program may be used to play song files created      *)
(* with the song-building program presented earlier.        *)
(* The songs are played via the Square Wave Synthesizer.   *)
(*****)
```

Uses

MemTypes, QuickDraw, OSIntf;

Var

```
nameOfFile : String;           {song filename}
eachNote   : Tone;            {var for each note}
ourSong    : SWSynthRec;      {array to hold notes}
index      : integer;         {index into OurSong}
```

```

anErr      : OSErr;           {error statuses for FSOpen/Close}
goodName   : boolean;        {has the user entered a valid file?}
refNum     : integer;        {file path reference number}
aChar      : char;           {for "Press any key..."}
recSize    : longint;        {tells file routines how large record is}

```

```

begin
  goodName := FALSE;
  recSize := SizeOf (tone);
  while NOT goodName DO
  begin
    ClearScreen;
    write ('What SongFile do you wish to play? ');
    readln (nameOfFile);
    if nameOfFile = '' then
      nameOfFile := 'Nameless Song';
    anErr := FSOpen (nameOfFile, 0, refNum);
    if anErr = fnfErr then           {File not found error}
    begin
      writeln;
      writeln ('That file does not exist!');
      writeln ('Press any key to try again...');
      aChar := ReadChar;
    end
  else
    goodName := TRUE;
  end;
  index := 0;

  (*****
  (* We will use Square Wave Synthesizer only. *)
  (*****
  ourSong.mode := SWMode;

  while (FSRead (refNum,recSize, @eachNote) <> EOFErr) do
  begin
    ourSong.triplets[index] := eachNote;
    index := index + 1;
  end;
                                     {while (FSRead (refNum...) <> EOFErr)
                                     do}

  anErr := FSClose (refNum);

  (*****
  (* Now play the song via StartSound. *)
  (*****

```



```
Type MENU
,1002 (4)
Example           ;;our Example menu
Bomb Sounds      ;;Alert Box option
Quit             ;;Quit option
```

The file starts off with the comment `* This is a simple resource file`. Resource files show a comment with an asterisk at the beginning of a line; when the program runs, the rest of the line is ignored. You can also specify comments within a resource line by placing a pair of semicolons before the comment.

I next define the output name of the resource file to be `Test.Rsrc`. I build the resource file shown above with an editor such as the Turbo Pascal editor and then compile it, much as I compile a Turbo Pascal program, with `RMaker`. The naming convention for resource text files and `RMaker`-compiled files is that the text files have the extension `.R` and compiled files have the extension `.Rsrc`. So having typed in the test resource file, I save it as `Test.R`. When it is compiled by `RMaker`, as you will see, the output file will be called `Test.Rsrc` because of the second line above.

The remainder of the file `Test.R` contains the actual definitions of the window and menu items. The first line of the window definition is

```
Type WIND
```

which says that the following several lines define a window. In general, an item definition in a resource file must begin with the `type XXXX` statement, where `XXXX` is the name of the item type. `RMaker` supports 12 different resource types, but we will discuss only windows (`Wind`) and menus (`Menu`). For a detailed explanation of the other types, consult the Turbo Pascal manual or the *Inside Macintosh* publications.

The next line in the window definition is a comma followed by the number 1000 and a 4 in parentheses. This line represents the name and identification number for the resource item. The name field is optional, so I omitted it, but I have designated the number 1000 as the identifier for the window item. You can use any value from 128 to 32767 for an identifier number. The number 4 in parentheses on this line means I wish to have this resource item loaded automatically. Several more attributes are discussed in the Turbo manual, but without the (4), the item will not be preloaded. After the (4) comes a comment. Again, the double semicolon tells `RMaker` to ignore the rest of the line.

```
The lines
```

```
Type WIND
```

and

```
,1000 (4)
```

are necessary for all of the different resource types. They are header lines used for all resource types with which you will work. The rest of the window definition contains items unique to WIND. The next line contains the words "Our Window" followed by another comment. In a WIND definition this line specifies the title to be displayed on the window. The next line contains the coordinates for the top, left, bottom, and right of the window. Notice that these numbers are not separated by commas; proper syntax in a resource file is just as important as in a Turbo Pascal file.

The next line specifies two parameters: whether the window will be visible or invisible and whether it has a go-away box in the top left corner. The specific values are Visible and Invisible, GoAway and NoGoAway. The last two lines of the window definition specify the ID and another reference value; we have selected the number 0 for both. So the general format for a WIND-type resource definition is:

```
type WIND
name,identification# (attributes)
title
top left bottom right
VisibleFlag CloseBoxFlag
DefinitionID
ReferenceValue
```

A window in a resource file must have this format. Also take note that this declaration is followed by a blank line, which must follow each resource-item definition.

The last two items in the example resource file define two menus: the Apple menu and my own menu. The format for MENU type items is simple: after the two header lines, state the menu title and the options on that menu. My first menu definition looks like this:

```
Type MENU
,1001 (4)
\14
```

The Type MENU line says this is a menu definition; the next line states that the resource identification number is 1001 and it should be preloaded. The

last line is a little different. The `\14` tells RMaker to use a special character as the title for this menu. The backslash specifies a special character and 14 is the number of the Apple icon at the top of the leftmost menu in Macintosh applications. So I have defined the first Menu type to have for a title the Apple icon. You will see later how the various desk accessory names are placed under that title so they may be executed.

The last definition in the file `Test.R` is the new menu, which contains the identification number 1002, the command to preload, the title `Example`, and the two options `Bombs Sounds` and `Quit`. Now that you know how to create a resource file, take a look at the resource compiler, RMaker.

Using RMaker

RMaker comes standard with the Turbo Pascal package and may be initiated by double-clicking on its icon, just like any other application. Once you have started RMaker, a dialog box asks you what file to compile. Notice that only files with the extension `.R` are displayed in this dialog box. Once you select a file on this screen by double-clicking on it or selecting it and clicking the open button), RMaker starts to compile the file. If any errors are encountered during compilation, they will be displayed on the screen. Otherwise click the quit button when compilation is finished. The Macintosh desktop will show the compiled file with the extension `.Rsc`. Now look at how to use this resource file in a true Macintosh application that uses event handling to interact with the user.

Event-Handling Programming with a Resource File

Up to now I have handled input from the user via several means: `readln` and a few mouse and mouse button status routines. One function declared in `ToolIntf` handles virtually all the input possible from a user; this function is called `GetNextEvent` and is defined like this:

```
function GetNextEvent ( mask : integer;  
    Var theEvent : EventRecord) : boolean;
```

Mask tells the function what type of events to return and which to ignore. TheEvent is a record that contains information about the returned event. Its structure is

```

EventRecord = record
    what      : integer;
    message   : longInt;
    when      : longInt;
    where     : Point;
    modifiers : integer;
end;

```

The information in the EventRecord returned from GetNextEvent may be looked at to determine what should be done within the application to respond. Take a close look at the following program, which performs event handling and uses the resource file I defined earlier:

```

Program EvntHndlr;

```

```

{$R EvntHndlr.RSRC}
{$U—}

```

```

Uses

```

```

    MemTypes, QuickDraw, OsIntf, ToolIntf;

```

```

Const

```

```

    Menu1 = 1001;           {menu ID #1001}
    Menu2 = 1002;           {menu ID #1002}

```

```

Var

```

```

    theEvent   : EventRecord;   {the event record}
    done       : boolean;       {are we finished yet?}
    menuH1     : MenuHandle;    {handle for Menu1}
    menuH2     : MenuHandle;    {handle for Menu2}
    menuResult : longInt;       {result of call to MenuSelect}
    whichMenu,
    WhichItem  : integer;       {High word and low word of MenuResult}
    ourSong    : SWSynthRec;    {array to hold notes}
    i          : integer;       {loop counter}

```

```

Procedure SetUp;

```

```

Var

```

```

    myWindow : WindowPtr;      {pointer to our window}

```

```

begin

```

```

    done := FALSE;
    InitGraf (@thePort);

```

```

InitWindows;
InitMenus;
InitCursor;

myWindow := GetNewWindow (1000, nil, pointer (-1));

menuH1 := GetMenu (menu1);
menuH2 := GetMenu (menu2);
AddResMenu (menuH1, 'DRVR');
InsertMenu (menuH1, 0);
InsertMenu (menuH2, 0);
DrawMenuBar;

(* Set up for notes selection in menu *)
ourSong.mode := SWMode;
for i := 0 to 19 do
begin
  ourSong.triplets[i].count := 262 + i;
  ourSong.triplets[i].amplitude := 200;
  ourSong.triplets[i].duration := 2;
  ourSong.triplets[i + 20].count := 262 + 1;
  ourSong.triplets[i + 20].amplitude := 200;
  ourSong.triplets[i + 20].duration := 2;
end;
end;                                     {procedure SetUp}

Procedure ProcessEvent;

Var
  whichWindow : WindowPtr;               {the window ptr filled by FindWindow}
  whichDA      : Str255;                 {which DA to open if Apple menu
                                        selected}
  dumbInt      : integer;                {blow off value returned from
                                        OpenDeskAcc}

begin
  case theEvent.what of

    MouseDown : case (FindWindow (theEvent.where, whichWindow)) of
      InMenuBar : begin
        menuResult := menuSelect (theEvent.where);
        whichMenu := HiWord (menuResult);
        whichItem := LoWord (menuResult);
        case WhichMenu of
          1 : begin {retrieve and open the DA}

```

```

        GetItem (menuH1, whichItem, whichDA);
        if (OpenDeskAcc(whichDA) > 0) then;
end;

2 : case whichItem of {Example menu}
1 : StartSound (@ourSong,
(2 + (40 * (SizeOf (tone)))),
Pointer (-1));

2 : done := TRUE;
end;
end;
HiliteMenu(0); {dehighlight menu selected}
end;

(* Need to handle DA activity *)
InSysWindow : SystemClick (theEvent, whichWindow);

InDrag : DragWindow (whichWindow, theEvent.where,
screenBits.bounds);

InGoAway : done := TrackGoAway (whichWindow,
theEvent.where);

end;
end;                                     {case theEvent.what of}

end;

begin                                     {program EvntHndlr}
  SetUp;
  repeat
  SystemTask;
  if GetNextEvent (everyEvent, theEvent) then
    ProcessEvent;
  until done;
end.                                     {program EvntHndlr}

```

The \$R compiler directive informs the Turbo compiler that it is to include the file EvntHndlr.Rsrc in the program; I renamed the file Test.R EvntHndlr.R and recompiled it via RMaker for use in the program. The next noteworthy declarations are the constants Menu1 and Menu2, defined as 1001 and 1002 respectively. Note that these numbers match the identification numbers in the resource file for the menu definitions. I define an EventRecord type

called `TheEvent` for subsequent use in event handling and declare a couple of variables called `menuH1` and `menuH2`, which are of `MenuHandle` type. `MenuHandle` is declared in `ToolIntf` as follows:

```
MenuHandle = ^MenuPtr;
MenuPtr = record
    menuID      : integer;
    menuWidth   : integer;
    menuHeight  : integer;
    menuProc    : Handle;
    enableFlags : LongInt;
    menuData    : Str255;
end;
```

The `MenuHandle` type variables are used in a couple of the menu-handling routines described shortly.

The main routine of the program starts off by calling `SetUp` and then performs a `repeat...until` loop to call `SystemTask`, `GetNextEvent`, and `ProcessEvent` until the program is Done. The `repeat...until` loop is the heart of Macintosh event-handling programming. The procedure `SystemTask` in `ToolIntf` updates any desk accessory items, for example resets the clock. Then `GetNextEvent` is called, and if it returns a true value, the routine `ProcessEvent` is called. The first parameter of `GetNextEvent`, `everyEvent`, is a constant defined in `OSIntf` to instruct `GetNextEvent` to return every type of event it encounters. Before I talk about how events are handled in this `ProcessEvent` routine. I should discuss some new setup routines.

Along with the initialization routines I discussed earlier, I invoke the routine `InitMenus` to tell the menu manager to do some work with menu routines. I display the window defined in the resource file `EventHndlr` by calling the routine `GetNewWindow`. Notice that `GetNewWindow` contains far fewer parameters than `NewWindow`. The format for `GetNewWindow` is

```
WPtr := GetNewWindow ( WID, wStorage, behind );
```

where `WPtr` is the pointer to the window, `Wid` is the identification number of the window as specified in the resource file, and `wStorage` points where the window should be stored (`nil` indicates that the window should be stored on the heap, since no local space is declared for it. Behind another `WindowPtr` type defines whether the window should be on top or behind other windows; the value returned by the function pointer (`-1`) instructs the window manager to place this window in front of all others. These last two parameters are identical to the first and third from last parameters in `NewWindow`. The only

parameter introduced in `GetNewWindow` is the identification number. You need not worry about the other parameters in `NewWindow`, since they are defined in the resource file.

I start building the menu bar for the program by calling the function `GetMenu` for both menus. `GetMenu`'s only parameter is the identification number for the menu resource definition as shown in the resource file. A `MenuHandle` is returned from `GetMenu` and is used next in the routine `AddResMenu`. Remember how I defined the Apple menu without any menu options? `AddResMenu` automatically adds options to the menu for the desk accessories defined for the system. I pass the `MenuHandle` for the Apple menu (`menuH1`) and the string `DRVR`, which tells `MenuHandle` to place the options on `menuH1`.

`InsertMenu` inserts each of our menus into the menu bar. `InsertMenu`'s first parameter is the handle of the menu to be added. Its second is an integer specifying before which menu the new menu should be placed; a value of 0 tells `InsertMenu` to place the menu after all others. Finally, the menu bar is drawn by calling the routine `DrawMenuBar`. The rest of the `SetUp` routine is used to define the bomb sounds made when the first option of the new menu is selected.

In the routine `ProcessEvent` I case out on the `what` field of the `Event` record, and I care only about `MouseDown`s. If a `MouseDown` occurs, `FindWindow` is called to determine where the mouse button was when the button was pressed. `FindWindow` takes two parameters, the point where the button was pressed (`theEvent.where`) and a `WindowPtr` type set by `FindWindow` to the window the mouse was in when the button was clicked. The value returned by `FindWindow` is an integer for which several constants have been declared: `InMenuBar`, `InSysWindow`, `InDrag`, and so on. For each of these results different processing must be performed. If the button was clicked in the menu bar (`InMenuBar`), I first call the `MenuSelect` function. `MenuSelect`'s only parameter is the `where` field of the `Event`. It returns a long integer whose high-order word contains the menu selected and whose low-order word contains the option selected within that menu. This is why I call the functions `HiWord` and `LoWord` to fill in the variables `WhichMenu` and `WhichItem` respectively. I case out on `WhichMenu` to determine whether the button was clicked in the Apple menu or the example menu. If the button was clicked on an item in the Apple menu, I must do some work to start up the appropriate desk accessory. First I call `GetItem` for the name of the desk accessory. `GetItem`'s parameters are the `MenuHandle` for the Apple menu, the item selected from `WhichItem`, and the name of the desk accessory (or DA) to start up. The last parameter is filled by `GetItem` and passed to `OpenDeskAcc`. The value returned by `OpenDeskAcc` may be ignored, since any error resulting

from opening it will be handled by `OpenDeskAcc` itself. The calls to `GetItem` and `OpenDeskAcc` start up the available desk accessories without the user knowing anything further about them.

If the second menu is selected (case `whichMenu` of), I know it is the Example menu and should either start the bomb sounds via `StartSound` or stop the program by setting `done` to `TRUE`. At the end of the block of code `InMenuBar` I call the routine `HiliteMenu` with the parameter 0 to dehighlight the highlighted menu.

If the result of the `FindWindow` call is `InSysWindow`, I must call the procedure `SystemClick` to handle activity on any open desk accessories. `SystemClick`'s parameters are the record `theEvent` and `TheWindow` in which the button was clicked.

If `FindWindow` returns the value `InDrag`, call the procedure `DragWindow`. `DragWindow`'s parameters are the window pointer set up by `FindWindow`, the `where` field of the `theEvent` record, and the `bounds` field of the `ScreenBits` record declared in `QuickDraw`.

Finally, if the result of `FindWindow` is `InGoAway`, `TrackGoAway` should be called to determine if the button is released in the go-away box and to close the window. `TrackGoAway`'s parameters are the window pointer returned from `FindWindow` and the `where` field of the `theEvent` record. I automatically set the variable `done` to the result of `TrackGoAway` so that the program will end when the window is closed.

That's all you need to know about event-handling programming to write Macintosh-style applications. I have covered a good deal of material in this chapter, so I suggest that you review all of the sections before continuing with our final chapter, which contains a few useful applications.

Review Summary

1. The `Turtle` provides you with several easy-to-use graphics routines in Turbo Pascal.
2. `QuickDraw` defines several very useful and fast graphics routines that are much more powerful than those in `Turtle`.
3. A pen may be used to draw pictures on the screen in black, dark gray, gray, light gray, and white.
4. Frequency is the numeric value that corresponds to the scale of musical notes; middle C has a frequency of 523.
5. The procedure `StartSound` allows you to produce sound on the Macintosh via a Turbo Pascal program.

6. Resource files let you externally define in your programs several types of items such as windows and menus.
7. RMaker compiles your resource (.R) files and creates .Rsrc files, which may be called up from a Turbo Pascal program.
8. GetNextEvent is the centerpiece of Macintosh-style applications. When used in an event loop, it allows you to handle all user input in a highly structured manner.

Quiz

1. What advantage do Turtle routines have over QuickDraw routines? Vice versa?
2. If I changed the coordinates for the light box in the StreetLight program from 216, 75, 266, 175 to 216, 0, 266, 100, what effect would that have on the graphics display?
3. Given the following code, how many times would "Music" be displayed?

```
while NOT TRUE do
  for i := 1 to 10 do
    writeln ( 'Music' );
```

4. What is GetNextEvent used for?
5. What is the difference between GetNewWindow and NewWindow?

A Few Programs for the Road

The Date-Minder Program
Batting-Average Program
Record Album Database Program
Summing Up
Review Summary
Quiz

In this chapter you will learn:

- A very useful electronic calendar program that can be modified quite easily by incorporating additional routines.
- A program to calculate and maintain baseball batting averages.
- A database program to catalog your album collection.
- The concept of modularity for designing menu-driven programs.

The Date-Minder Program

The first program in this chapter is one I hope you will find helpful all year round. It's an electronic calendar that will tell you what events are coming up in the next 30 days. All you have to do is enter such information as birthdays and anniversaries. Then, when you need to see what's on the horizon for the next month, you select the option "View the next month of events" from the menu. All your calendar information will be stored on disk, so you can add more dates to the file at any time.

Look at the program before I discuss it in detail.

Program DateMinder;

```
(*****)  
(* This program may be used to create an electronic calendar *)  
(* where you can store important dates and view the next *)
```

```
(* month's important dates. *)
(*****)
```

Uses

```
MemTypes, QuickDraw, OSIntf;
```

Const

```
NameOfFile = 'Important Dates';
```

Type

```
MonthDayRec = record                {record for each month}
    month    : String[9];           {Month name}
    numDays  : integer;             {number of days in month}
end;
```

```
EventRec = record                  {record for each event}
    month    : integer;             {event month}
    date     : integer;             {event date}
    occasion : String;              {event name}
end;
```

Var

```
done          : boolean;           {are we finished with the program?}
yearArray     : array[1..12] of
    MonthDayRec;                   {array of months}
eventVarRec   : EventRec;          {reads/writes EventRecs}
anErr         : OSErr;             {error returned from file routines}
refNum        : integer;           {file path reference number}
recSize       : longint;           {tells file routine how large record is}
```

Procedure AddDates;

Var

```
valid         : boolean;           {is user's input valid?}
finished      : boolean;           {is user finished adding dates?}
month         : integer;           {month of the occasion}
date          : integer;           {date of the occasion}
occasion      : String;            {the occasion name}
response      : char;              {user's Y/N response for more additions}
junkRecord    : EventRec;          {used for finding EOF}
```

```
begin                                {procedure AddDates}
(*****)
(* Need to check to see if this file exists first. *)
```

```

(* If it doesn't, we need to create. *)
(*****)

anErr := FSOpen (nameOfFile, 0, refNum);
if an Err = fnfErr then {File not found error}
begin
  anErr := Create (nameOfFile, refNum, 'DATM', 'DATS');
  anErr := FSOpen (nameOfFile, 0, refNum);
end;

(*****)
(* Find the end of the file so that we can append *)
(*****)
anErr := SetFPos (refNum, fsFromLEOF, 0);

finished := FALSE;
while NOT finished do
begin
  ClearScreen;
  valid := FALSE;
  while NOT valid do
  begin
    writeln;
    writeln;
    writeln;
    write ('What month is this occasion (1-12)? ');
    readln (eventVarRec.month);
    if ((eventVarRec.month > 0) AND (eventVarRec.month < 13)) then
      valid := TRUE;
  end; {while NOT valid}
  valid := FALSE;
  while NOT valid do
  begin
    writeln;
    writeln;
    writeln;
    write ('what date is this occasion (1-31)? ');
    readln (eventVarRec.date);
    if ((eventVarRec.date > 0) AND
        (eventVarRec.date <= yearArray[eventVarRec.month].numDays)) then
      valid := TRUE;
  end; {while NOT valid}
  writeln;
  writeln;

```

```

        writeln;
        write ('What is the occasion? ');
        readln (eventVarRec.occasion);
        anErr := FSWrite (refNum, recSize, @eventVarRec);
        writeln;
        writeln;
        writeln;
        write ('do you have any more entries (Y/N)?');
        response := ReadChar;
        if ((response = 'N') OR (response = 'n')) then
            finished := TRUE;
    end;                                     {while NOT finished}
    anErr := FSClose (refNum);
    anErr := FlushVol (nil, 0);
end;                                         {procedure AddDates}

Procedure ViewDates;

Var
    holdDateTime : DateTimeRec;             {holds current date/time}
    nextMonth    : integer;                 {next month (for Jan. after Dec.)}
    aChar        : char;                    {used for "Press any key to continue"}

begin                                       {procedure ViewDates}
    ClearScreen;
    GetTime (holdDateTime);                 {library routine which returns date}
    anErr := FSOpen (nameOfFile, 0, refNum);
    if (anErr = fnfErr) then                 {File not found error}
    begin
        GotoXY (30, 10);
        write ('NON-EXISTENT FILE!!!');
        GotoXY (27, 12);
        write ('Press any key to continue...');
        aChar := ReadChar;
    end
    else
    begin
        writeln;
        writeln;
        write ('*****');
        writeln ('*****');
        if (holdDateTime.month = 12) then
            nextMonth := 1
        else
            nextMonth := holdDateTime.month + 1;
    end
end

```

```

while (FSRead (refNum, recSize, @eventVarRec) <> EOFErr) do
  if (((eventVarRec.month = holdDateTime.month) AND
      (eventVarRec.date >= holdDateTime.day)) OR
      ((eventVarRec.month = nextMonth) AND
      (eventVarRec.date < holdDateTime.day))) then
    writeln ( ' ', eventVarRec.occasion, 'is on',
              yearArray[eventVarRec.month].month, eventVarRec.date : 3);
  write ('*****');
  writeln ('*****');
  anErr := FSClose (refNum);
  writeln;
  write ( ' Press any key to continue...');
  aChar := ReadChar;
end;
end;                                     {procedure ViewDates}

Procedure DisplayMenu;

Var
  selection : char;                       {user's menu selection}
  valid      : boolean;                   {is user's response valid?}

begin                                     {procedure DisplayMenu}
  valid := FALSE;
  while NOT valid do
  begin
    ClearScreen;
    writeln;
    writeln;
    writeln;
    writeln ( ' Which would you like to do:');
    writeln ( ' 1. Add dates to the file');
    writeln ( ' 2. View the next month of events');
    writeln ( ' 3. QUIT');
    writeln;
    writeln;
    write ( ' (Enter 1, 2 or 3) ');
    selection := ReadChar;
    if (selection = '1') OR (selection = '2') OR (selection = '3') then
      valid := TRUE;
  end;                                     {while NOT valid}
  case selection of
    '1' : addDates;

    '2' : ViewDates;

```

```

    '3' : done := TRUE;

end;                                {case Selection}
end;                                {procedure DisplayMenu}

begin                                {program DateMinder}
    recSize := SizeOf (eventRec);
    {set up table for months and number of days}
    yearArray[1].month := 'January';
    yearArray[2].month := 'February';
    yearArray[3].month := 'March';
    yearArray[4].month := 'April';
    yearArray[5].month := 'May';
    yearArray[6].month := 'June';
    yearArray[7].month := 'July';
    yearArray[8].month := 'August';
    yearArray[9].month := 'September';
    yearArray[10].month := 'October';
    yearArray[11].month := 'November';
    yearArray[12].month := 'December';
    yearArray[1].numDays := 31;
    yearArray[2].numDays := 28;
    yearArray[3].numDays := 31;
    yearArray[4].numDays := 30;
    yearArray[5].numDays := 31;
    yearArray[6].numDays := 30;
    yearArray[7].numDays := 31;
    yearArray[8].numDays := 31;
    yearArray[9].numDays := 30;
    yearArray[10].numDays := 31;
    yearArray[11].numDays := 30;
    yearArray[12].numDays := 31;
    done := FALSE;
while NOT done do
    DisplayMenu;
end.                                {program DateMinder}

```

The program has the structure type `EventRec`, which holds the month, date, and name of each occasion. The main routine performs initialization and calls `DisplayMenu` until the program is finished. The routine `DisplayMenu` shows the main menu and depending on the user's selection, calls `AddDates` or `ViewDates` or stops the program.

`AddDates` is the routine that actually writes the event records out to the file. The file where these dates are stored is called `Important Dates`. This

name can be changed through the constant `NameOfFile` at the beginning of the program. At the start of `AddDates` the file is opened and the end of the file is found. Then, while the user is not finished adding records and events two loops are performed to validate the input for the month and date of the occasion. The occasion name is entered and the record is written to the file and the user is asked if he or she has any more entries. If there are more, the while NOT Finished loop is continued; otherwise the file is closed and control is passed back to `DisplayMenu`.

The last function in the date-minder program is called `ViewDates`. First the date is determined via the procedure `GetTime`, which is declared in `OSIntf`. `GetTime` returns a record of `DateTimeRec` whose format is also defined in `OSIntf`. This record structure contains fields for month, day, and year; their contents designate which events stored in the file will occur within the next month. This if statement opens the event file and determines the next month:

```
if ( holdDateTime.month = 12 ) then
  nextMonth := 1
else
  nextMonth := holdDateTime.month + 1;
```

As you can see, if this month is December, I want to make next month January. Otherwise I add 1 to the current month.

Then, the events of the next month are displayed on the screen. A very long if statement is used to determine whether a month fits into this category. It may be helpful to split this statement into two, either of which may be true for the date and event to be displayed:

```
1) if ( ( eventVarRec.month = holdDateTime.month ) AND
      ( eventVarRec.date >= holdDateTime.day ) )
```

or

```
2) ( ( eventVarRec.month = nextMonth ) AND
     ( eventVarRec.date < holdDateTime.day ) )
```

Suppose today is July 10 and the event takes place on July 31; both segments of the case are true, since the event's month (`eventVarRec.month`) is the same as `holdDateTime.month` and the event's date (`eventVarRec.date`) is greater than `eventVarRec.date`.

On the other hand, suppose today is July 10 and the event is on August 5. Both segments are true here also, since the event's month (`eventVar-`


```

arrOfPl      : array[1..20] of PlayerRec; {array of players}
numOfPlyrs  : integer;                  {current number of players}
anErr       : OSErr;                    {error returned from file routines}
refNum      : integer;                  {file path reference number}
aChar       : char;                     {used for "Press any key..."}
recSize     : longint;                  {tells file routines how large record is}

```

Procedure LoadArray;

```

begin                                     {procedure LoadArray}
  numOfPlyrs := 0;
  anErr := FSOpen (team, 0, refNum);
  (*****
  (* If the file doesn't exist, create it          *)
  (*****
  if anErr = fnfErr then                   {File not found error}

  begin
    anErr := Create (team, refNum, 'BAVE', 'AVES');
    anErr := FSOpen (team, 0, refNum);
  end;
  while (FSRead (refNum, recSize,
    @arrOfPl[numOfPlyrs = 1]) <> EOFErr) do
    numOfPlyrs := numOfPlyrs + 1;
    anErr := FSClose (refNum);
end;                                     {procedure LoadArray}

```

Procedure ReDoFile;

```

Var
  i : integer;                             {loop counter}

begin                                     {procedure ReDoFile}
  anErr := FSDelete (team, 0);             {delete the old file first}
  anErr := Create (team, 0, 'BAVE', 'AVES');
  anErr := FSOpen (team, 0, refNum);
  for i := 1 to numOfPlyrs do
    anErr := FSWrite (refNum, recSize, @arrOfPl[i]);
    anErr := FSClose (refNum);
    anErr := FlushVol (nil, 0);
end;                                     {procedure ReDoFile}

```

Procedure EntInfo;

```

Var
  i      : integer;                         {loop counter}

```

```

    response : char;                {user's response}
    data      : real;               {user's input figure}

begin
    LoadArray;
    for i := 1 to numOfPlyrs do
    begin
        ClearScreen;
        write ('Do you have any data for', arrOfPI[i].lastName, '?');
        response := ReadChar;
        if ((response = 'Y') OR (response = 'y')) then
        begin
            writeln;
            write ('How many additional at-bats?');
            readln (data);
            arrOfPI[i].atBats := arrOfPI[i].atBats + data;
            write ('How many additional hits?');
            readln (data);
            arrOfPI[i].hits := arrOfPI[i].hits + data;
            if (arrOfPI[i].atBats <> 0) then
                arrOfPI[i].battingAve := arrOfPI[i].hits / arrOfPI[i].atBats
            else
                arrOfPI[i].battingAve := 0;
            end;
        end;
        ReDoFile;
    end;
    {procedure EntInfo}

Procedure DelPlyr;

Var

    IName : String[15];            {last name of player to be deleted}
    i,j    : integer;              {counters}
    found  : boolean;              {have we found the name yet?}
    aChar  : char;                 {used for "Press any key..."}

begin
    ClearScreen;
    write ('What is the last name of the player to be deleted?');
    readln (IName);
    LoadArray;
    i := 1;
    found := FALSE;
    while ((i <= numOfPlyrs) AND (NOT found)) do

```

```

    if (IName = arrOfPI[i].lastName) then
        found := TRUE
    else
        i := i + 1;

if NOT found then
begin
    writeln;
    writeln ('THAT NAME IS NOT IN THE FILE!');
end
else
begin
    for j := 1 to numOfPlyrs - 1 do
        arrOfPI[j] := arrOfPI[j + 1];
    numOfPlyrs := numOfPlyrs - 1;
    ReDoFile;
    writeln;
    writeln ('THAT PLAYER HAS NOW BEEN DELETED...');
end;
writeln;
write ('PRESS ANY KEY TO CONTINUE...');
aChar := ReadChar;
end;                                     {procedure DelPlyr}

Procedure DoDisp;

Var
    i : integer;                           {loop counter}

begin                                     {procedure DoDisp}
    anErr := FSOpen (team, 0, refNum);
    if anErr = fnfErr then                 {File not found error}
    begin
        writeln;
        writeln ('THERE IS NO BASEBALL FILE ON THIS DISK!');
        writeln;
        write ('Press any key to continue...');
    end
    else
    begin
        ClearScreen;
        write ('NUMBER NAME ');
        writeln('AT-BATS HITS BATTING AVE');
        write ('-----');
        writeln ('-----');
    end
end

```

```

    for i := 1 to numOfPlyrs do
        writeln (' ', arrOfPI[i].number:4, ' ', arrOfPI[i].firstName:10, ' ',
            arrOfPI[i].lastName:15, ' ', arrOfPI[i].atBats:3:0,
            ' ', arrOfPI[i].hits:3:0, ' ',
            arrOfPI[i].battingAve:3:3);

    end;
    writeln;
    write ('Press any key to continue...');
    aChar := ReadChar;
    anErr := FSClose (refNum);
end;                                     {procedure DoDisp}

Procedure sort;

Var
    h,j      : integer;                {loop counters}
    tempPlyr : PlayerRec;              {temporary storage for sort proc}

begin                                       {procedure Sort}
    LoadArray;
    for j := 1 to numOfPlyrs do
        for h := 1 to numOfPlyrs - j do
            if (arrOfPI[h].battingAve < arrOfPI[h + 1].battingAve) then
                begin
                    tempPlyr := arrOfPI[h];
                    arrOfPI[h] := arrOfPI[h + 1];
                    arrOfPI[h + 1] := tempPlyr;
                end;
        end;
    end;                                     {procedure Sort}

Procedure DisAve;

begin                                       {procedure DisAve}
    Sort;
    DoDisp;
end;                                       {procedure DisAve}

Procedure AddPlyr;

begin                                       {procedure AddPlyr}
    Clear Screen;
    LoadArray;
    numOfPlyrs := numOfPlyrs + 1;
    write ('What is the player"’s first name? ');

```

```
    readln (arrOfPI[numOfPlyrs].firstName);
    write ('What is the player"s last name? ');
    readln (arrOfPI[numOfPlyrs].lastName);
    write ('What is his number? ');
    readln(arrOfPI[numOfPlyrs].number);
    write ('How many at-bats does he currently have? ');
    readln (arrOfPI[numOfPlyrs].atBats);
    write ('How many hits does he currently have? ');
    readln (arrOfPI[numOfPlyrs].hits);
    if(arrOfPI[numOfPlyrs].atBats <> 0) then
        arrOfPI[numOfPlyrs].battingAve :=
            arrOfPI[numOfPlyrs].hits / arrOfPI[numOfPlyrs].atBats
    else
        arrOfPI[numOfPlyrs].battingAve := 0;
    ReDoFile;
end;                                     {procedure AddPlyr}

begin                                   {main procedure}
    recSize := SizeOf (playerRec);
    done := FALSE;
    while NOT done do
        begin
            ClearScreen;
            writeln ('Which of the following do you wish to perform:');
            writeln;
            writeln;
            writeln (' 1. Add a player to the file');
            writeln (' 2. Delete a player from the file');
            writeln (' 3. Enter information for a player');
            writeln (' 4. Display the file by batting average');
            writeln (' 5. QUIT');
            writeln;
            writeln;
            write ('Please enter your selection:');
            choice := ReadChar;
            case choice of
                '1' : AddPlyr;

                '2' : DelPlyr;

                '3' : EntInfo;

                '4' : DisAve;

                '5' : done := TRUE;
```

```

        otherwise
        begin
            writeln;
            writeln;
            writeln ('INVALID RESPONSE...TRY AGAIN!');
            writeln;
            write ('Press any key to continue...');
            choice := ReadChar;
        end;
    end;                                     {case choice of}
end;                                         {while NOT done do}
end.                                         {main procedure}

```

This program introduces no new concepts and therefore should be very easy to understand. The menu displays four options: add a player, delete a player, enter information for existing players, and display the file by order of batting average. The program contains a separate routine for each option as well as three other routines used by some or all of the menu-option routines. These additional functions, ReDoFile, Sort, and LoadArray, are responsible for rewriting the file, sorting the file in the memory array, and reading the file into the memory array respectively. Step through this program and try entering a few statistics to verify that the program works as you expect.

Record Album Database Program

The final program I will show you is one that catalogs record albums. Having entered all my albums, I can either sequentially view the file or search the file based upon album title, artist, year of release, or an extra field for comments. Again, there are no concepts introduced in this program, so you should have no problem following the logic.

Program Albums;

Uses

MemTypes, QuickDraw, OSIntf;

```

(*.....*)
(* This program allows the user to create a file which      *)
(* contains records for each of his or her musical albums.  *)
(*.....*)

```

Const

AlbFile = 'MyAlbums'; {the name of the file}

Type

```

AlbumType = record           {the album record type}
  title  : String[25];       {album title}
  artist : String[20];       {album artist}
  year   : String[4];        {album's year of release}
  xtraField : String[20];    {extra field for misc. search}
end;

```

Var

```

anAlbum : AlbumType;        {the album record var}
finished : boolean;         {are we finished?}
choice   : char;            {user's menu selection}
anErr    : OSerr;          {error returned from file routines}
recSize  : longint;        {tells file routines how large record is}
refNum   : integer;        {file path reference number}

```

Procedure AddAlbums;

```

var
  done : boolean;           {Are we finished adding?}

begin                       {procedure AddAlbums}
  done := FALSE;
  anErr := FSOpen (albFile, 0, refNum);
  (*****
  (* If the file doesn't exist, create it.          *)
  (*****
  if anErr = fnfErr then     {File not found error}
  begin
    anErr := Create (albFile, 0, 'ALBS', 'RECS');
    anErr := FSOpen (albFile, 0, RefNum);
  end;
  (*****
  (* Find the end of the file so that we can append *)
  (*****
  anErr := SetFPos (refNum, fsFromLEOF, 0);

  while NOT done do
  begin
    ClearScreen;
    GotoXY (10, 10);
    write ('What is the album's title ($ to end)?');
    readln (AnAlbum.title);
    if (anAlbum.title = '$') then      {enter $ to stop}
      done := TRUE

```

```

else
begin
  GotoXY (10, 11);
  write ('Who is the artist? ');
  readln (anAlbum.artist);
  GotoXY (10, 12);
  write ('What year was the album made? ');
  readln (anAlbum.year);
  GotoXY (10, 13);
  write ('What do you want in the extra field?');
  readln (anAlbum.xtraField);
  anErr := FSWrite (refNum, recSize, @anAlbum);
end;
end;
anErr := FSClose (refNum);
anErr := FlushVol (nil, 0);
end;                                     {procedure AddAlbums}

Procedure DispTheRecord;

begin                                     {procedure DispTheRecord}
  ClearScreen;
  GotoXY (20, 10);
  writeln ('The album title is:           ', anAlbum.title);
  GotoXY (20, 11);
  writeln ('The album artist is:         ', anAlbum.artist);
  GotoXY (20, 12);
  writeln ('The album"s release was in:  ', anAlbum.year);
  GotoXY (20, 13);
  writeln ('The extra field is:          ', anAlbum.xtraField);
  GotoXY (25, 16);
  wrote ('PRESS ANY KEY TO CONTINUE...');
  choice := ReadChar;
end;                                     {procedure DispTheRecord}

Procedure DisplayFile;

begin                                     {procedure DisplayFile}
  anErr := FSOpen (albFile, 0, refNum);
  (*****
  (* If file doesn't exist, display message. *)
  (*****
  if anErr = fnfErr then                   {File not found error}
  begin
    ClearScreen;

```

```

    writeln (' THERE IS NO ALBUM FILE ON THE DISK!');
    writeln;
    write (' Press any key to continue');
    choice := ReadChar;
end
else
begin
    while (FSRead (refNum, recSize, @anAlbum) <> EOFErr) do
        DispTheRecord;
        anErr := FSClose (refNum);
    end;
end;
                                         {procedure DisplayFile}

Procedure GrpSearch;

Var
    foundOne : boolean;           {have we found a match yet?}
    group    : String[20];       {holds name of group we wish to search}

begin
                                         {procedure GrpSearch}
    GotoXY (15, 16);
    write ('For which Group/Artist do you wish to search?');
    readln (group);
    anErr := FSOpen (albFile, 0, refNum);
    (*****
    (* If the file doesn't exist, display message. *)
    (*****
    if anErr = fnfErr then           {File not found error}
    begin
        ClearScreen;
        writeln (' THERE IS NO ALBUM FILE ON THE DISK!');
        writeln;
        write (' Press any key to continue');
        choice := ReadChar;
    end
    else
                                         {search the file}
    begin
        foundOne := FALSE;
        while (FSRead (refNum, recSize, @anAlbum) <> EOFErr) do
            if (anAlbum.artist = group) then
                begin
                    foundOne := TRUE;
                    DispTheRecord;
                end;
            end;
        end;
    end;
end;

```

```

if NOT foundOne then
begin
  writeln;
  writeln;
  writeln ('THAT ARTIST IS NOT ON THE FILE!');
  writeln;
  write ('Press any key to continue...');
  choice := ReadChar;
end;
anErr := FSClose (refNum);
end;
end;                                     {procedure GrpSearch}

Procedure Tag1Search;

Var
  foundOne : Boolean;                       {have we found a match yet?}
  tag1      : String[20];                   {holds tag 1 we wish to search}

begin                                     {procedure GrpSearch}
  GotoXy (15, 16);
  write ('What is the extra field you wish to search?');
  readln (tag1);
  anErr := FSOpen (albFile, 0, refNum);
  (*****
  (* If the file doesn't exist, display message. *)
  (*****
  if anErr = fnfErr then                   {file not found error}
  begin
    ClearScreen;
    writeln (' There is no album file on the disk!');
    writeln;
    write (' Press any key to continue');
    choice := ReadChar;
  end
  else                                     {search the file}
  begin
    foundOne := FALSE;
    while (FSRead (refNum, recSize, @anAlbum) <> EOFErr) do
      if (anAlbum.xtraField = tag1) then
        begin
          foundOne := TRUE;
          DispTheRecord;
        end;
      if NOT foundOne then

```

```

begin
  writeln;
  writeln;
  writeln ('THAT EXTRA FIELD IS NOT ON THE FILE!');
  writeln;
  write (' Press any key to continue...');
  choice := ReadChar;
end;
anErr := FSClose (refNum);
end;
end;                                     {procedure Tag 1 Search}

Procedure TitleSearch;

Var
  foundOne : boolean;                       {have we found a match yet?}
  title     : String[25];                   {holds title we wish to search}

begin                                     {procedure GrpSearch}
  GotoXY (15, 16);
  write ('What title do you wish to search?');
  readln (title);
  anErr := FSOpen (albFile, 0, refNum);
  (*****
  (* If the file doesn't exist, display message.          *)
  (*****
  if anErr = fnfErr then                    {file not found error}
  begin
    ClearScreen;
    writeln (' THERE IS NO ALBUM FILE ON THE DISK!');
    writeln;
    write (' Press any key to continue');
    choice := ReadChar;
  end
  else                                       {search the file}
  begin
    foundOne := FALSE;
    while (FSRead (refNum, recSize, @anAlbum) <> EOFErr) do
      if (anAlbum.title = title) then
        begin
          foundOne := TRUE;
          DispTheRecord;
        end;
      if NOT foundOne then
        begin

```

```

        writeln;
        writeln;
        writeln ('THAT TITLE IS NOT ON THE FILE!');
        writeln;
        write ('Press any key to continue...');
        choice := ReadChar;
    end;
    anErr := FSClose (refNum);
end;
end;                                     {procedure TitleSearch}

Procedure YearSearch;

Var
    foundOne : boolean;                   {have we found a match yet?}
    year      : String[4];                {holds year we wish to search}

begin                                     {procedure GrpSearch}
    GotoXY (15, 16);
    write ('For what year do you wish to search?');
    readln (year);
    anErr := FSOpen (albFile, 0, refNum);
    (*****
    (* If the file doesn't exist, display message. *)
    (*****
    if anErr = fnfErr then                 {File not found error}
    begin
        ClearScreen;
        writeln (' THERE IS NOT ALBUM FILE ON THE DISK!');
        writeln;
        write (' Press any key to continue');
        choice := ReadChar;
    end
    else                                     {search the file}
    begin
        foundOne := FALSE;
        while (FSRead (refNum, recSize, @anAlbum) <> EOFErr) do
            if (anAlbum.year = year) then
                begin
                    foundOne := TRUE;
                    DispTheRecord;
                end;
            if NOT foundOne then
                begin
                    writeln;

```

```
writeln;
writeln ('THAT YEAR IS NOT ON THE FILE!');
writeln;
write ('Press any key to continue...');
choice := ReadChar;
end;
anErr := FSClose (refNum);
end;
end;                                     {procedure YearSearch}

Procedure Search;

Var
  done    : boolean;                       {are we finished searching?}
  selection : char;                       {the user's search selection}

begin                                     {procedure Search}
  done := FALSE;
  while NOT done do
  begin
    ClearScreen;
    GotoXY (20, 8);
    writeln ('Which field would you like to search:');
    GotoXY (20, 9);
    writeln (' 1. Group/Artist Name');
    GotoXY (20, 10);
    writeln (' 2. Album Title');
    GotoXY (20, 11);
    writeln (' 3. Year of Release');
    GotoXY (20, 12);
    writeln (' 4. Extra field');
    GotoXY (20, 13);
    writeln (' 5. QUIT');
    GotoXY (20, 14);
    write ('Please enter your selection: ');
    selection := ReadChar;
    case selection of
      '1' : GrpSearch;

      '2' : TitleSearch;

      '3' : YearSearch;

      '4' : Tag1Search;

      '5' : done := TRUE;
```

```

        otherwise;                {don't do anything; redisplay screen}
    end;                            {case selection of}
end;                                {while NOT done do}
end;                                {procedure Search}

begin                                {main procedure}
    recSize := SizeOf (anAlbum);
    finished := FALSE;
    while NOT finished do
    begin
        ClearScreen;
        GotoXY (20, 8);
        writeln ('Select one of the following options:');
        GotoXY (20, 9);
        writeln (' 1. Add albums to the file');
        GotoXY (20, 10);
        writeln (' 2. Sequentially display the file');
        GotoXY (20, 11);
        writeln (' 3. Search the file');
        GotoXY (20, 12);
        writeln (' 4. QUIT');
        GotoXY (20, 13);
        write ('What is your choice?');
        choice := ReadChar;
        case choice of
            '1' : AddAlbums;

            '2' : DisplayFile;

            '3' : Search;

            '4' : finished := TRUE;

            otherwise;                {dont' do anything; redisplay screen}
        end;                            {case choice of}
    end;                                {while NOT finished do}
end.                                program Albums}

```

This program lends itself to the addition of new options and modules. The search routines are almost identical. When you wish to stop adding albums to the file, just enter a dollar sign and control will return to the main menu. The extra field may be used to specify the condition of the album, the owner of the album, the type of music, and so forth. Be sure to log some albums to see how easy it is. Then search for albums by a particular artist, from a certain year, and so on.

Summing Up

The date-minder program can definitely help you and your family if you often forget important dates. The program is modular; you can easily add more options to the main menu, for instance viewing the next 60 days' events or maybe even the entire file. Because of the modular structure of the program—each task having its own routine—all you have to do is add more options to the menu in `DisplayMenu` and make sure they are accepted as valid in the `if` and `case` statements thereafter. Then add the appropriate routines to the program. The baseball and album programs are both modular and can easily be customized.

I sincerely hope that the information you have learned in this and all the preceding chapters is all you need to write your own applications. Even though you have finished reading this book, keep it handy for use as a quick reference guide when working in Turbo Pascal.

Review Summary

1. Modular structure is used in all the programs in this chapter: each separate task has its own routine, and more tasks can be added by developing additional routines.

Quiz

1. Why couldn't I have added one to the current month to get the value of the next month in the date-minder program?
2. Why do I always check the value of the `atBats` field before calculating the batting average in the `EntInfo` routine?
3. How are invalid entries handled in the main menu of the album database program?

—Appendix A—

The Borland Toolboxes and Turbo Tutor

Turbo Pascal Database Toolbox
Turbo Pascal Access
Turbo Pascal Toolbox Numerical Methods
Turbo Pascal Tutor

Borland International offers three additional products for the Turbo Pascal compiler. The first one, Turbo Pascal Database Toolbox, provides routines that sort files (TurboSort) and maintain a database (Turbo Pascal Access).

Turbo Pascal Database Toolbox

One part of the database toolbox calls a function that will sort an input file based on the specifications you provide. This function has the syntax

```
function TurboSort ( ItemLength : integer;  
    InpPtr, LessPtr, OutPtr : ProcPtr ) : integer;
```

where ItemLength uses SizeOf to describe the length of the items to be sorted. InpPtr, LessPtr, and OutPtr are pointers to routines to be executed for the three phases input, sort, and output respectively. These last three parameters are addresses and may be passed by placing an at sign (@) before the names of the routines. For example, to sort a file of PlayerRec type that receives data via an Enter routine, that uses PlayerSort to sort the players, and that reports results via a Display routine, I call TurboSort like this:

```
SortResult := TurboSort( SizeOf ( PlayerRec ), @Enter,  
    @PlayerSort, @Display );
```

Of course I declare and write the routines Enter, PlayerSort, and Display somewhere else in the program. The Enter routine, which lacks parameters, should specify how data for the players is to be put in—via screen entry or

file reading—and calls another Database Toolbox routine, `SortRelease`, to pass the item to be sorted. `PlayerSort` should be a function that returns a Boolean value of true if the two parameter records are in the correct sorted order. Finally, the `Display` routine, which has no parameters, should perform an output loop that calls `SortReturn`. This Toolbox routine retrieves the next item in the list, writes or displays it, and continues item after item until the Toolbox function `SortEOS` returns a value of TRUE, which indicates the whole list has been parsed.

Database Toolbox contains two units with sorting routines: `Sort` and `LSort`. The routines I described above are for use with `Sort`, which may be used to sort up to 32767 items. To sort more than 32767 items use the routines in `LSort`, which have the same syntax but whose routine names are preceded with an L.

Turbo Pascal Access

The Turbo Pascal Access portion of the Database Toolbox provides prewritten routines to maintain a database. These routines are available in high- and low-level interface. The records in the database file may be ordered by one or more keys into the file. Suppose I want to index a phone book file on both the phone number and the name. I can do it because I have a separate index file that specifies the ordering of the records in the data file by field.

The Turbo Pascal Access routines require a bit of setup via `SetConst`, which comes with the package. Once this is complete, you may write programs to use either low- or high-level calls to Access routines. The indexing method used for the database files is a B+ tree structure. The B+ tree is similar to the binary tree except that more than two siblings are allowed on each level. B+ trees are discussed in detail in Appendix B of the Turbo Pascal Database Toolbox manual.

Through the low-level routines `AddRec`, `DeleteRec`, and `PutRec`, the data file may be manipulated in any way. The index files must be kept in synch via the routines `AddKey` and `DeleteKey`. The data file may be parsed via the routines `FindKey`, `SearchKey`, `NextKey`, and `PrevKey`.

Alternatively, high-level routines make much of the index manipulation invisible to the programmer. The names of these routines generally start with a TA for Turbo Access. The database file may be handled via calls to the routines `TAInsert`, `TADelete`, and `TAUpdate`. In addition, records may be read via `TARead`; the file may be parsed via `TAPrev` or `TANext`; and it may be reset to the beginning via `TAReset`. The Turbo Pascal Database

Toolbox manual details these operations as well as several others I have not mentioned. The purpose of this discussion is to show you what tools are in the Database Toolbox and to determine if it includes the tools to satisfy your programming needs.

Take a quick look at some of the capabilities of the other Toolbox package, Numerical Methods.

Turbo Pascal Toolbox Numerical Methods

The Numerical Toolbox is geared toward engineers, scientists, and students of advanced mathematics. The package includes routines for the following types of problems and solutions:

- Roots to equations in one variable
- Interpolation
- Differentiation
- Integration
- Matrices
- Eigenvalues and eigenvectors
- Initial and boundary value methods
- Least-square approximation
- Fast Fourier transform
- Graphics demonstrations

If these topics are foreign to you, the Numerical Methods Toolbox may not be what you need. However, the graphics demonstration programs at the end of the manual are worth seeing. Least-squares approximation and Fourier transforms are presented in the programs LSQDemo and FFTDemo respectively. Although this package is directed toward scientific and engineering applications, students of calculus and even algebra will find interesting routines that quickly solve mathematical problems.

Borland International offers one more add-on package for the Macintosh: Turbo Pascal Tutor.

Turbo Pascal Tutor

The Turbo Pascal Tutor comes with a diskette that contains all the programs discussed in the manual. Anyone who wishes to learn how to program Turbo Pascal on the Macintosh will find valuable information in this package. It may

be used as either a tutorial for the novice or as a reference guide for the more experienced programmer. With over 600 pages of programming information, it is probably the most detailed publication on Macintosh Turbo Pascal to date. The accompanying disk containing all the source code for programs and resource files is invaluable for quickly working with the sample programs. The Turbo Pascal Tutor is a worthwhile investment for anyone working with Turbo Pascal on the Macintosh.

Appendix B

Reserved Words

These are the reserved words in Macintosh Turbo Pascal:

and	implementation	repeat
array	in	set
begin	inline	scl
case	interface	shr
const	label	string
div	mod	then
do	nil	to
downto	not	type
else	of	unit
external	or	until
file	otherwise	uses
for	packed	var
forward	procedure	while
function	program	with
goto	record	xor
if		

— Appendix C —

Quiz Answers

Chapter 1

1. 1K = 1024. 128K = 131,072, or approximately 128,000. 512K = 524,288 or approximately 512,000. 1 meg = 1,024,000. 4 meg = 4,096,000.
2. An interpreter translates Pascal code on the fly as it is being executed and unlike a compiler, is not generally capable of producing stand-alone double-clickable applications.
3. In the problem-solving phase the programming problem is defined and analyzed, and a general solution of the problem is developed step by step. In the implementation phase the general solution is translated into code that the computer can understand. The general solution itself is called an algorithm.
4. First start the water boiling, then get an egg and drop it into the boiling water. A delay is executed for the length of cooking time necessary before the egg is removed.

Chapter 2

1. No
2. The option Clear deletes the selected text and does not copy it to the Clipboard.
3. Stack Windows places the edit windows on top of each other, whereas Tile Windows gives each window its own area of the screen.
4. Compiling to disk results in a double-clickable application on the disk, whereas compiling to memory simply creates an executable version of the program in RAM.

Chapter 3

1. An identifier is a name given to a constant, type, variable, procedure, function, or program.

2. Comments are an important part of programming. They may be placed anywhere in the program for the purpose of explaining code steps and/or identifiers.
3. At the end of the program.
4. The proper order is const, type var.
5. An integer takes up less memory and must always represent a whole number.
6. The semicolon denotes the end of a statement in Pascal.
7. In order of high precedence to low: *, /, MOD, DIV, +, -.
8. By the use of parentheses.

Chapter 4

1. Subsequent output is always started on the next line followed by a writeln whereas a write does not cause output to start on a new line.
2. 65535.34
3. You can insert blank lines via writeln with no parameters.
4. Readln provides the programmer with the ability to request input from the user.
5. if ((A < 100) AND (A > 0)) then...if ((A = 100) OR (A = 0)) then...

Chapter 5

1. A counter determines how many times a loop is executed and an accumulator gathers a total that will periodically be updated. An example of a counter is i as shown here:

```
i := 0;
if name = 'Smith' then
  i := i + 1;
```

Alternatively, an example of i as an accumulator is

```
for i := 1 to 10 do
  :
  :
  :
```

2. It does not matter how or where you use a counter inside loop as long as you do not modify it for the purpose of early termination.
3. 1 := 1; Total := 0;

```
while (1 <= 5) do
  begin
    readln(UserNumber);
    Total := Total + UserNumber;
  end;
writeln('The total is: ', Total);
```

4. Begin-end pairs within a for statement denote a compound statement within the loop.
5. write('How many iterations? ');

```
readln(1);
for Loop := 1 to I do
  ...
```

6. A repeat...until loop will always execute once, since the loop test is not performed until the end, whereas a while loop may never be executed.
7. Nested loops may repeatedly perform a block of code within another repeatedly performed block of code.
8. Goto statements should be avoided in order to maintain a structured programming approach where hard-to-follow branches are not used.

Chapter 6

1. A function by definition returns a value separate from any parameters passed to it, whereas a procedure may return values only as passed by reference.
2. Global variables may be accessed and changed by any routine (unless redefined as a local procedure), whereas local variables are limited to change only by the routine in which they are declared.
3. Parameters passed as var may be modified by the called routine, whereas those passed with no var specification may not be changed.
4. The \$1 compiler directive may be used to check I/O results via the function IOResult to determine if unexpected input was received, for example a real when an integer is required.
5. A unit is a program block that allows better program structure, modularity, and separate compilation.
6. The interface section is visible by other areas, for example the calling program. The implementation section is private to the unit and not visible by calling routines.
7. UnitMover moves units from one file to another. It lets you make the Turbo program small enough to run on a 512K Macintosh.

Chapter 7

1. A. 100
B. 5
C. 12
D. FALSE
E. 1
F. 2
2. Pascal is fun
3. Wednesday

Chapter 8

1. Although these figures started out as fixed values, they must be permitted to change throughout the program in order to specify the new range of possible numbers after each guess. For example, if your number is 75, MinNum changes to 51 after the program's first guess so that the new range of possible numbers is 51 through 100.
2. If the conditions are met in the first if statement, the second if statement is not executed, so the code can be simplified as follows:

```
if NOT ( inDecimal >=0 ) OR NOT ( inDecimal <= 65535 ) then
begin
  writeln;
  writeln ( 'INVALID ENTRY...PLEASE TRY AGAIN!' );
  writeln;
end;
```

3. I could have reversed the order of the outHex elements in the WRITELN statement in DecToHex to read

```
writeln ( 'is', outHex[ 6 ], outHex[ 5 ], outHex[ 4 ],
  outHex[ 3 ], outHex[ 2 ], outHex[ 1 ], '...' )
```

4. Yes, 0 through 9 are written the same way in hexadecimal and base 10 notation.

Chapter 9

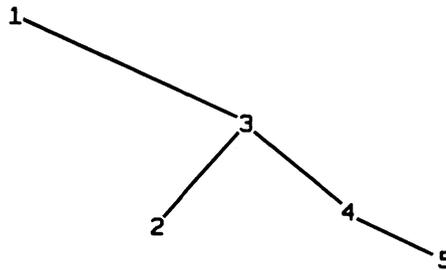
1. Simply put, a file is a collection of records and a record is a group of fields.
2. Because the purpose of a record is not to centralize data structures but

rather to group related items like the name, and position in program BaseBall.

3. **If** was used to remove any possible garbage left over in memory from any other variables that may have used that memory address. It is not necessary to understand why this garbage may exist; however, it is important always to initialize sets just as you would initialize any other variable.

Chapter 10

1. Because it is very difficult to follow the logic of a recursive routine and because they can very easily cause the program to crash due to lack of memory.
2. With the linked list there is no worry about reshuffling the entire list when an item is added; the only item affected is the one that now must point to the new item.
- 3.



4. A stack is a LIFO structure, in which the last item placed on the stack is the first to be processed. By contrast, a queue is a FIFO structure, in which the first item in the queue is the first to be processed.

Chapter 11

1. The entire collection is the file, each album is a record, and each song is a field for that particular record.
2. The statement ran sequentially through the file until the end in order to receive new entries.
3. Variant records can save memory space.

Chapter 12

1. It forces the programmer to think through the program logic more thoroughly without having a debugger to fall back on for help.

2. C IS FUN
3. A: 2 B: 0 C: 15 D: 0

Chapter 13

1. Turtle routines are easy to use, but QuickDraw routines are much more powerful.
2. It would shift the light box up 75 pixels, but the lights would still be in their original position.
3. None, since the value of (while NOT TRUE do) will never be true.
4. GetNextEvent is used to get all input from a user via one routine. It is the heart of all Macintosh-style event-handling programs.
5. GetNewWindow uses fewer parameters than NewWindow because it operates with a window definition in a resource file.

Chapter 14

1. The same problem would arise: the Succ of 12 (Succ(12)) would be 13, which is meaningless. So no matter how I set up that part of the code, I would have to take the special case of January being after December into account.
2. It is checked so that I don't divide by 0.
3. The screen is redisplayed.

—Appendix D—

Bibliography

Borland International, *Turbo Pascal Macintosh* (package manual), Borland International, Inc, 1986.

Borland International, *Turbo Pascal Tutor* (package manual), Borland International, Inc, 1987.

Wikert, J. and Davis, S., *Learning Macintosh Pascal*, Scott, Foresman and Company, 1986.

Index

A

Absolute value function, 167-168

Accumulator, 118

Add

- modifying values of, 144, 145
- variable parameters and, 141-143

AddDates, 306

AddRecord, 244, 326

Address, 139

Add routine, 138

Advanced concepts

- handle data types and, 221-225
- introduction to, 218-234
- pointer and, 221-225, 225-230
- queues and, 230-233
- recursion and, 218-221
- stacks and, 230-233
- trees and, 230-233

Advanced data structures, 216

Algorithm, 7, 58

Alphabetical phone-book file program,
245-249

Alphanumeric information, 70

American National Standard Institute, 5

American Standard Code, 15-16, 261

Amplitude, 288

AND, 110

ANSI standard Pascal, 5

Apostrophe, 61, 64, 70

Appending a file, 239

Apple II Pascal, 5

Apple Menu, 25-26

Application programs, 181

Applications and advanced concepts. *See*

Advanced concepts

data structures in. *See* Data structures

debugging and file analysis. *See*

Debugging and file analysis

graphics, sound, and resources. *See*

Graphics, sound, and resources

programs in. *See* Programs

records and files in. *See* Records and
files

ArcTan function, 170-171

Arithmetic, 79-82

Arrays

application of, 213-216

data types and, 83

elements of, 207, 225

files versus records in, 236-237

parallel, 208-209

simple, 206-208

ASCII, 15-16, 261

- ASCII hexadecimal conversion values, 261–262
- Assembly language
 - binary numbers and, 14
 - MacsBug and, 266
- Assignment operator, 74
 - values of expression and, 77
- Assignment statement
 - invalid, 75
 - use of, 81, 82
 - valid, 74
 - variables and, 74
- Asterisks, 63
- Auto Indent, 33, 86
- Auto Save, 42

- B**
- Back, Turtle graphics and, 271
- Backward looping with Downto, 125
- Base 10 integers, 67
- BASIC, 4
- Batting-average program, 308–314
- Begin-end pair
 - examples of, 99
 - Pascal programs and, 47
 - procedure and, 138
 - program body and, 64–65
- Beginner's All-purpose Symbolic Instruction Code, 4
- Bibliography, 336
- Binary numbers, 14, 67–68
- BinaryPointer, 233
- Binary search, 185
- BitAnd function, 260
- 16-bit integer, 67
- Bits
 - flags and, 15
 - term of, 14
- Blank lines, printing of, 97, 98
- Blank spaces, 46
- Blind pointer, 224
- Body
 - of function, 148
 - of procedure, 136
 - of repeat statement, 126
- Boolean data, 72
- Boolean End of File function, 238
- Boolean expression, 185
 - function and, 150, 151
- Boolean operators, 110
- Boolean type, 206
- Boolean values, 66, 72
- Borland toolboxes, 325–327
 - Turbo Pascal access in, 326–327
 - Turbo Pascal database toolbox in, 325–326
 - Turbo Pascal toolbox numerical methods in, 327
- Boundary value methods, 327
- Brackets, curly, 63
- B+ tree structure, 326
- Buffer, 237
- BuffSize, 237
- Button routine program, 281–282
- Byte
 - ASCII and, 16
 - integers and, 67
- 2-byte integer, 67
- 1-byte-length indicator, 71
- 4-byte value, 221

- C**
- CalcFact, 220
- CalcNDisp, 200
- Calendar program, 301–308
- Call procedure, 138–139, 149
- Carriage return, 97
- Case-sensitivity with Find What, 34
- Case statement, 111, 252, 395
- Central library, 166
- Change, 35, 36
- Characters, 66, 70–72
 - identifier and, 60, 61
- Character string, printing of, 90, 91
- Checks in If statement, 111
- Check Syntax, 41
- Chr function, 175
- Clear, 32
 - Turtle graphics and, 271
- ClearScreen, 185
- Clipboard, 32
- Clock Folder file, 18

- Close, 237
 - corresponding, 238
 - file menu and, 28
 - Closing a file, 238
 - COBOL, 4, 5
 - Coefficient scientific notation, 69
 - ColorType, 204
 - Command key, 53
 - Comment
 - block, 63–64
 - declaration and, 61, 63
 - embedded, 63
 - line, 61, 63
 - marks, 284
 - resource file and, 291
 - specifying, 291
 - Common Business Oriented Language, 4, 5
 - Comp. *See* Computational data type
 - Compat Unit file, 19
 - Compilation, 48
 - Compile menu, 39–44
 - functions of, 39, 40
 - selecting Run from, 65, 66
 - Compiler
 - versus interpreters, 13–14
 - MacsBug and, 266
 - Turbo Pascal, 13
 - Compiler directive, 63
 - flag-type, 152
 - \$L, 155
 - \$O, 155–156
 - parameters and, 151–157
 - \$R, 156
 - event handling and, 296
 - \$U, 157
 - 2's complement notations, signed integers
 - and, 68
 - Complete Pascal programs, 181
 - Compound expressions, 110
 - with while loop, 117
 - Compound If statement, 108
 - Comp real numbers, 69
 - Computerese, 3
 - Computer language, 4
 - Computational data type, 203
 - Concat function, 171–172
 - Concepts, advanced. *See* Advanced concepts
 - Condition
 - false, 106
 - true, 106
 - Conditional, 103, 104
 - decision symbol and, 10
 - Conditional If statement, 105
 - Conditional structures, 130
 - Connector symbol, 11
 - Const. *See* Constant
 - Constant, 62–64, 75–76
 - declaring a, 75
 - Turtle and, 272
 - Constant section, 75
 - Control variable, 130
 - loops and, 126
 - Conversion programs, 185
 - Conversion values for hexadecimal ASCII,
 - 261–262
 - Copy function, 172
 - Copy option, 32
 - Corresponding close, 238
 - Cosine function, 169
 - Counter-control variable, 119–120
 - Counter loop, 116–117
 - Count field, 286
 - C program language, 5
 - Create, 238
 - CubInchToLiter, 185
 - Curly brackets with comments, 63
 - CurrentTask, 230
 - Cut, 32
- D**
- Database toolbox, 325–326
 - Data space, 252
 - Data structures
 - advanced, 202–217
 - numeric types and, 202–203
 - parallel arrays and, 208–209
 - records and files and, 210–211
 - sets and, 211–213
 - simple arrays and, 206–208
 - types and, 204–206
 - using arrays, records, and sets in an application and, 213–216

- Data type, 82–85
 - array, 83
 - declaring, 66–72, 73
 - defining, 73
 - enumerated, 83
 - handle, 221–225
 - identifier and, 73
 - ordinal, 83, 85
 - real number, 203
 - records, 83
 - scalar, 85
 - user-defined, 83–85
 - Data values, 72
 - assigning with Read and Readln, 97–103
 - Date-minder program, 301–308
 - DateTimeRec, 307
 - Debugger, 255
 - compiler directives and, 152
 - TMON, 256
 - Debugging and file analysis, 48, 255–269
 - encipher/decipher program and, 263–266
 - MacsBug and, 266–268
 - TMON and, 268
 - Decimal point, 68
 - real numbers and, 68
 - Decimal-to-hexadecimal conversion
 - program, 191–199
 - explanation of, 197–199
 - Decipher option, 266
 - Decipher program, 263–266
 - Decisions, 103–106
 - Decision symbol, 10–12
 - exit points of, 12
 - Decision test, 103
 - Declarations, 62–64, 66, 75
 - examples of, 99
 - DecToHex, 197
 - Default directory types, 43
 - Delete function, 172–173
 - DeleteRec, 326
 - Desk Accessories for Apple Menu, 25, 26
 - Diamond-shaped decision symbol, 10
 - Differentiation, 327
 - Directive include-file, 154
 - Disassembler, 14
 - Disk activity, 256
 - Display routine, 326
 - DIV. *See* Integer division
 - Dots symbol, 262–263
 - Double-clickable applications, 13
 - Double integers, 203
 - Double-precision data types, 69
 - Double real variable, 203
 - Downto, 125
 - Drawing freehand, 284
 - Dump program, 256–260
 - Duration, 288
 - \$D with MacsBug, 267
 - \$D+ with MacsBug, 267
 - Dynamic storage allocation, 230
- ## E
- Echo printing, 256
 - Editing Turbo Pascal program, 48–50
 - Edit menu, 32–33
 - Editor, 13
 - Edit Transfer, 29
 - Edit window, 37
 - after saving, 51, 52
 - Eigenvalues and eigenvectors, 327
 - Eigenvectors, eigenvalues and, 327
 - 8088 or 8086 assembly language, 14
 - Electrical switches, 14
 - Electronic calendar program, 301–308
 - Elements, 207
 - Else portion of If statement, 104
 - Embedded comment, 63
 - Encipher/decipher program, 263–268
 - Ending value with loops, 126
 - Enter routine, 324–325
 - Enumerated data type, 83, 205–206
 - EOF. *See* Boolean End of File function
 - Equations in one variable, 327
 - Error message, 6
 - Read and Readln and, 100
 - Errors, 13
 - E scientific notation, 69
 - Event-handling programming with
 - resource file, 293–299
 - EventRecord, 294
 - EventRec type, 306
 - Executable lines, 47

- Exit points, 12
- Exp function, 170
- Exponent, 69
- Exponential notation, 68
- Expression, 77-78
 - arithmetic, 77
 - valid, 77
 - variables, mathematical operators, and constants in, 77
- Extended integers, 203
- Extended real numbers, 202-203
 - data types of, 69
- External storage, 17
- Extra field with record album database program, 322

- F**

- Factorial program, 218-220
- False, decision making and, 103-104
- Fast Fourier transform, 327
- Fields, records and files, 235-236
- Field-width parameter, 94, 95
- FIFO. *See* First-in-first-out strategy
- FigureHex, 197
- File-building program, 256
- File-dump program, 256-260
- File-handling library procedures, 237-240
- File-handling routines. *See* Boolean End of File function
- File menu, 27-31
- File name, 243
- Files, 210-211, 235-254
 - appending, 239
 - defining, 290
 - file-handling library procedures and, 237-240
 - phone-book program and, 240-245
 - records and, 211
 - sorting and merging, 245-251
 - variant records and, 251-253
- Files versus arrays of records, 236-237
- Find, 34
- Finder
 - Pascal disk and, 20
 - RAM disk and, 20
- Find Error, 41, 42
- Find Next, 35
 - dialog box for, 35, 36
- Find option, 34, 35
- Find What, 34
- First-in-first-out strategy, 231
- First line of program, 60-61
- Flag, 14-15, 152
- Flexibility, 124, 125
- FlipFlop, 198
- Floating-point numbers, 69
- Flowchart, 7-8
- Flowchart symbols, 8-12
- Flowlines, 10-11
- Flow of a program, 103
- FlushVol, 240
- FName, 238
- FNum, 238
- Font/DA Mover
 - application, 39
 - utilities and sample programs file of, 18
- Font menu, 39
- Font Windows, 39, 40
- For loop, 212
- Format menu, 37-39
- Formatting
 - output, 94-97
 - program, 86
- Formula Translator, 4
- For statement, 113, 119, 119-125
- FORTTRAN, 4
- Forwd, Turtle graphics and, 271
- FSClose, 240
- FSOpen, 238
- FSWrite, 240
 - syntax for, 239
- Function
 - parameters and, 148-151
 - type of, 148
- Fundamentals, 3-23
 - flowchart symbols and, 8-12
 - machine language and, 14
 - Macintosh storage and, 14-17
 - Pascal and, 4-5
 - programming and, 3-4
 - programming technique and, 6-8
 - Syntax and, 6
 - Turbo Pascal and, 12-14, 18-20, 20-22

G**Get Info**

- compile menu and, 42
- dialog box for, 42

GetMouse, 284**GetNewWindow**, 297**GetNextEvent**, 293**GetPort**, 278**Global variables**, 185

- versus local variables, 140-141

Go-away box window, 292**Goto statement**, 130-132

- avoiding the use of, 131-132

GrafPort record structure, 277**Graphics**, 270-285

- mouse and, 281-285
- standard Macintosh, 275-281
- Turtle, 271-275

Graphics demonstrations, 327**Grow box**, 281**Guess-a-number program**, 186-191

- explanation of, 190-191

H**Halt**, 263**Handle data types**, 221-225**Hard copy**

- printing, 53
- of program, 53

Header lines, 292**Heading**

- function and, 148
- procedure and, 136
- program, 46, 60
- Turtle graphics and, 271

Hexadecimal ASCII conversion values, 261-262**Hexadecimal notation**, 191-192**Hex-character dump program**, 256-260**Hex values**, 192-193**HideCursor**, 278**Hi function**, 177**High-level interface**, 326**High-order bit**, 68**High-order nibble**, 260**HiWord function**, 177**Home**, Turtle graphics and, 272**Home Cursor**, 35**I****\$.I.** See **Include file directory****Icon on disk**, 52**Identifier**, 60, 61-62, 72-73

- invalid, 62
- legal numeric variable, 74
- naming, 62
- valid, 62

If statement, 103

- common errors with, 106-108
- compound, 108
- conditional, 105
- nesting the, 108-110

If...then...else compound statement, 103**Images**, 282**Imagewriter**, 20**Implementation of Turbo Pascal unit**, 158**Implementation phase**, 7-8**InchToMeter**, 185**Include file directory**, 43, 154**Include files**, 27-28**InDecimal**, 197**Indentation in program formatting**, 46, 86**Index variable**, 119**Infinite loop**, 114, 116**Information**, 103**InitGraf**, 278**Initial and boundary value methods**, 327**Initial value methods**, 327**InitWindows**, 280**Inner loop**, 130**Input file**, 325**Input/output routines**, 159**Input/output symbol**, 8-10**Insert function**, 173**Inserting numbers with binary tree**, 233**InsertMenu**, 298**Integer division**, 78**Integer division operator**, 78**Integers**, 66-68

- double, 203
- extended, 203

- long, 202-203
 - maximum and minimum values for, 66-67
 - range of, 202
 - valid, 66
- Integration, 327
- Intel-based 8088/8086 microprocessors, 14
- Intensity, 288
- Interactive statements, 98
- Interface
 - high-level, 326
 - low-level, 326
 - of Turbo Pascal unit, 157-158
- Internal storage, 17
- Interpolation, 327
- Interpreter
 - versus compilers, 13-14
 - Turbo Pascal and, 13
- Int library function, 169
- Invalid assignment statement, 75
- Invalid identifier, 62
- Invisible window, 292
- I/O. *See* Input/output symbol
- IOResult, 153

- K**
- K. *See* Kilobyte
- KeyPressed, 159-160
- Kilobyte, 16-17

- L**
- \$L. *See* Link Object file directory
- Label declaration, 131
- Label with Goto statement, 131
- Last-in-first-out traffic, 231
- Least-square approximation, 327
- Legal numeric variable identifier, 74
- Length of a string variable, 171
- Library, 166
- Library features, 165-178
- Library procedures, 237-240
- Library routines, 166, 167-177
 - mathematical routines and, 167-171
 - string routines and, 171-174
- LIFO. *See* Last-in-first-out traffic
- Linked list, 225
- Linker, 13
- Link Object file directory, 43
 - compiler directive and, 155
- Lister Folder file, 18
- Ln function, 170
- LoadArray, 314
- Local variable
 - declaring, 140
 - global versus, procedures and, 140-141
- Lo function, 177
- LongInt. *See* Long integer
- Long integer, 202-203
- Loop
 - counting, 116-117
 - infinite, stopping, 114, 116
 - inner, 130
 - nested, 129-130
 - outer, 130
 - repeat...until, 129
 - summary of, 126
 - while, 114, 129
- Loop control variable, 117
- Loop counter, 116-117
- Looping structures, 113-133
 - backward with Downto and, 125
 - counting the, 116-117
 - Goto statement and, 130-132
 - nested, 129-130
 - repeat statement and, 126-128
 - repeat...until, 129
 - for statement and, 119-125
 - summing, 118-119
 - while, 129
 - while statement and, 114-116
- Low-level interface, 326
- Low-order nibble, 260
- LoWord function, 177
- LSort, 326

- M**
- Machine code, 102
- Machine-language programming, 14
- Mac II Interfaces disk file, 18
- Macintalk Folder file, 19

- Macintosh
 - communicating with Turbo Pascal, 12-14
 - memory, 14, 16-17
 - mouse for, 281
 - storage, 14-17
 - window manager program, 278-289
 - MacPaint, 53
 - MacsBug, 255, 266-268
 - commands, 267
 - invoking, 266-267
 - Mac screen, 94
 - Main memory, 8
 - Main module, 135, 136
 - Main program, 185
 - declarations of, 216
 - logic of, 185
 - Mantissa scientific notation, 69
 - Mathematical calculation with flowcharts, 10
 - Mathematical order of operation, 78-79
 - Mathematical routines, 167-171
 - Mathematical variables, 81, 82
 - Matrices, 327
 - MaxGuess, 190
 - Megabyte, 17
 - Memory, RAM-ROM, 16-17
 - Memory location, 221
 - Memory requirements of data types, 221
 - Menu, 25-44
 - Apple, 25-26
 - Compile, 39-44
 - Edit, 32-33
 - File, 27-31
 - Font, 39
 - Format, 37-39
 - Search, 34-37
 - Transfer, 44
 - Menu1, 296-297
 - Menu2, 296-297
 - Menu bar, 298
 - MenuHandle type variables, 297
 - Merging records, 245-251
 - Message, printing of, 90, 91
 - Metric conversion program, 181-186
 - explanation of, 185-186
 - Microprocessor, 14
 - Misc Folder file, 19
 - Mistake correction, 49
 - MOD. *See* Modulo division
 - Modular programming, 135
 - Modules of program, 135
 - main, 135, 136
 - Modulo division, 78
 - Motorola-based 68000 microprocessors, 14
 - Mouse, 281-285
 - Mouse-based program, 282
 - Mouse button, 281
 - Mult
 - procedure and, 138
 - variable parameters and, 141-143
 - Multiple-item Readln statement, 101
 - Multi-variable declaration, 147
 - Music in Turbo Pascal, 285-290
 - MyDA Folder file, 18
 - MyDemo Folder file, 18
- N
- Natural log function, 170
 - Nested loop, 129-130, 185
 - Nesting, 108
 - New
 - file menu and, 27
 - memory and, 222
 - NewString, 198
 - NewWindow, 280-281
 - Nibble
 - high-order, 260
 - low-order, 260
 - NoGrowDocProc, 280-281
 - NOT, 110
 - Note frequencies, 286
 - NoWrap, Turtle graphics and, 272
 - Null string, 72
 - length indicator, 72
 - Numerical methods, 327
 - Numerical toolbox, 327
 - Numeric types, 202-203
- O
- \$O. *See* Output file, directory of
 - Odd function, 168
 - Off-page connector symbol, 11
 - Open, file menu and, 27

- Operating system, ROM and, 17
 - Operator order of precedence, 78-79
 - Optional parameter `BufferSize`, 237
 - Options, 33
 - Compile menu and, 42
 - dialog box for, 33, 43
 - OR, 110
 - Order of precedence of operators, 78-79
 - Ord function, 174-175
 - Ordinal data type, 83, 85
 - OunceToGram, 185
 - Outer loop, 130
 - Out-of-range subscripting, 207
 - Output
 - first line of, 96
 - formatting, 94-97
 - Output file
 - directory of, 43
 - enciphered file and, 266
 - Output unit file, 160
 - Overlay structure, 252
- P**
- Page setup, 29
 - Parallel arrays, 208-209
 - Parameters, 141-157
 - compiler directives and, 151-157
 - field-width, 94, 95
 - functions and, 148-151
 - passing, 143
 - passing information with, 141
 - type declarations for, 144
 - value, 145, 145-148
 - variable, 141-145
 - `Writeln` and, 89
 - Parentheses, 64
 - comment and, 63
 - order of operations and, 79
 - Pascal. *See also* Turbo Pascal
 - Apple II, 5
 - definition of, 4-5
 - University of California at San Diego
 - version, 5
 - PasConsole, 185
 - QuickDraw and, 278
 - unit and, 159-160
 - .pas extension, 44
 - PasPrinter unit, 94
 - Passing parameters, 143
 - Past, 32
 - PenDown, Turtle graphics and, 272
 - Pen movement, 271-275
 - Pen pattern, 278
 - PenUp, Turtle graphics and, 272
 - Phase implementation, 7-8
 - Phase problem-solving, 7-8
 - PhoneArray, 249
 - Phone-book file, 245-249
 - displaying contents of, 249-251
 - Phone-book program, using a file in,
 - 240-245
 - PhoneType, 243
 - PlayerSort, 326
 - Plus sign, 212
 - Pointer, 221-230
 - assigning value to, 223
 - blind, 224
 - var, 224
 - Pointers program, 228
 - Pointer type
 - declaration, 228
 - up arrow and, 222
 - Position function, 173-174
 - Positive values, Turtle graphics and, 271
 - Pred function, 175-176
 - Print, 29
 - file menu and, 30
 - printing program with, 53
 - Printer
 - preparing, 53
 - sending output to, 93-94
 - Printer identifier, 94
 - Printing
 - blank lines, 97, 98
 - echo, 256
 - Turbo Pascal program, 52-54
 - Problem-solving phase, 7-8
 - Procedure, 134-141
 - call, 138-139
 - execution of, 135-136
 - global versus local variables and, 140-141
 - passing values to, 139
 - placement of, 137
 - structure of, 136
 - use of, 137-139
 - variables with, 139-140

- Procedure definition statement, 141
 - Procedure dispose, 223-224
 - Processing strategies, 230
 - Process symbol, 8
 - Program block, 61
 - examples of, 99
 - main, 185
 - modules and, 65
 - Program Disk, 18, 20
 - Program flow
 - if statements and, 106
 - procedure and, 136
 - of while statement, 114
 - Program heading, 46, 60
 - Programming
 - definition of, 3-4, 6-8
 - modular, 135
 - Programming languages, 4-5
 - machine-language, 14
 - Programs, 181-201, 301-323
 - application, 181
 - BaseBall, 215
 - batting-average, 308-314
 - body of, 64-65
 - controlling flow of, 103
 - date-minder, 301-308
 - decimal-to-hexadecimal conversion, 191-199
 - editing, 48-50
 - elements of, 60-66
 - encipher/decipher, 263-266
 - execution speed of, 13
 - file-building, 256
 - file-dump, 256-260
 - first, 44-47
 - formatting of, 86
 - guess-a-number, 186-191
 - hex-character dump, 256-260
 - metric conversion, 181-186
 - phone-book, 240-245
 - printing, 52-54
 - record album database, 314-322
 - review of, 65-66
 - running, 47-48
 - saving, 50-52
 - tape-counter, 199-201
 - Program statement, 46, 47
 - compiling programs to disk by, 155-156
 - examples of, 99
 - Punctuation, 205
 - importance of, 81
 - PutRec, 326
- ## Q
- Queues, 230-233
 - QuickDraw graphics, 275-277
 - ROM and, 17
 - Quit, 31
 - Quiz answers, 330-335
 - Quotes
 - doubling single, 70-71
 - single, 70
- ## R
- \$R*. See Resource file, directory of
 - RAM, 16-17, 19
 - Random-access memory, 16-17
 - disk package, 19
 - Range with guess-a-number program, 185
 - \$R* compiler directive, 156
 - event handling and, 296
 - Read, 237
 - assigning data values with, 97-103
 - general syntax of, 98
 - input command of, 8
 - ReadChar, 159-160
 - Read.file program disk file, 18
 - Reading a file, 238
 - Read-line statement, 47
 - Readln
 - assigning data values with, 97-103
 - general syntax of, 98
 - statement, 101
 - Read Me program disk file, 18
 - Read-only memory, 16-17
 - Real data, 69
 - Real numbers, 66, 68-69
 - converting, 203
 - data types and, 203
 - extended, 202-203
 - formatting of, 96
 - fractional parts in, 68
 - index, 208

- Real variable
 - double, 203
 - standard, 203
- Record album database program, 314-322
- Records, 210-211
 - data type of, 83
 - declaring, 210-211, 213
 - files versus arrays of, 236-237
 - size of, 253
 - sorting and merging, 245-251
 - use of, in application, 213-216
 - variant, 251-253
- Record structure, 253
- Recursion, 218-221
- ReDoFile, 314
- Relational operators, 104
- Repeat statement, 126-128
- Repeat...until loop, 129
 - event handling programming and, 297
 - tape counter program and, 200
 - while versus, 129
- Repeat...until statement, 113, 127
 - function and, 150-151
- Reserved words, 329
- Reset, 237
- Resource file, 156-157, 290
 - directory of, 43
 - event-handling programming with, 293-299
 - simple, 291
 - syntax for, 291-292
- Resources
 - event-handling programming and, 293-299
 - graphics, sound, and, 270-300
 - RMaker and, 293
- RESUME option, 100
- Return, 101
- Rewrite, 237
- RMaker
 - compiling, 292-293
 - using, 293
 - utilities and sample programs file of, 18
- ROM, 16-17
- Roots to equations in one variable, 327
- Rounding function, 168-169
- Routines, library, 167-177
- Run, 47
 - compile menu and, 39
 - Run program, 47, 48, 49
 - Run-time error, 41
- S
- Sample, 89
- SANE. *See* Standard Apple Numeric Environment Library
- Save
 - dialog box for, 51
 - File menu and, 28
- Save As
 - File menu and, 28
 - saving a program and, 51
- Save Defaults, 31
 - File menu and, 44
- Save option, 50-51
- Saving Turbo Pascal program, 50-52
- Scalar data type, 85
- Scientific notation, 68
- Scope of variable, 140
- Scope rules, 140
- Screen, 22, 24, 25, 94, 256
- Screen dump, 53
- SearchFile, 244
- Search menu, 34-37
 - functions of, 34
- Semicolons, 76-77
 - errors with If statements and, 107-108
 - procedure and, 136
 - program heading and, 60
 - uses of, 46
 - writeln and, 64
- Set, 211-213
 - declaring a, 212
 - using in an application, 213-216
- SetConst, 326
- SetHeading, Turtle graphics and, 272
- SetPosition, Turtle graphics and, 272
- SetRect, 278, 280
- Shift Left, 33
- Shift Right, 33
- ShowCursor, 278
- Shr operator, 260
- Simple arrays, 206-208
- Simple resource file, 291
- Simple Turbo Pascal arithmetic, 79-82
- Sine function, 169-170

- Single write statement, 92, 93
- 68000 assembly language, 14
- SizeOf function, 176
- Song
 - notes and duration for, 288
 - playing a simple, 285
 - saving on disks, 290
- SongBuilder file, 288
- Song file program, 286-287
- Sort
 - batting-average program and, 314
 - database toolbox and, 326
- Sorting and merging records, 245-251
- SortRelease, 326
- Sound, 270-300
 - making music in Turbo Pascal and, 285-290
 - utilities and sample programs file of, 19
- Spaces, 46
 - program heading and, 60
 - Writeln and Writeln statements and, 95
- Spelling error correction, 49
- Square root function, 167
- Square wave synthesizer tone generation, 285
- Squaring function, 167
- Stacks, 230-233
 - array and, 231
 - parameters and, 147-148
- Stack Windows, 37, 38
- Standard Apple Numeric Environment Library, 69
- Standard Macintosh graphics, 275-281
- Standard real variable, 203
- Starting value with loops, 126
- StartSound, 285
- Startup window, 33
- Statement, 88-112
 - Boolean operators and, 110
 - case, 103, 111
 - conditionals and, 103
 - data values and, 97-103
 - decisions and, 103-106
 - For, 113, 119-125
 - Goto, 130-132
 - If, 103
 - common errors with, 106-108
 - nesting the, 108-110
 - If statements and, 106-108
 - If...then...else compound, 103
 - procedure definition, 141
 - program, 46, 47
 - Read and, 97-103
 - read-line, 47
 - Readln and, 97-103
 - repeat, 126-128
 - repeat...until, 113
 - function and, 150-151
 - Turbo Pascal, 88-112
 - while, 113, 114-116
 - write, 89-97
 - write-line, 47
 - writeln, 89-97
- Statement indentation, 46
- StepPointer, 229
- Stepwise refinement, 135
- Storage, 14-17
 - external, 17
 - internal, 17
- Storage medium, 8
- String, 66, 70-72
 - maximum size of, 71
 - null, 72
 - \$R compiler directive and, 156
 - value of, 222
- String output, 97
- String routines, 171-174
- String type declaration, 71-72
- Structure and syntax, 58-88
 - constants and, 75-76
 - data types and, 66-72, 82-85
 - expression and, 77-78
 - mathematical order of operation and, 78-79
 - program formatting and, 86
 - semicolon and, 76-77
 - simple Turbo Pascal arithmetic and, 79-82
 - Turbo Pascal program and, 60-66
 - variables and, 72-73, 74-75
- Structured design, 6-7
 - modular programming and, 135
 - procedures and, 135
- Structured programming, 135
 - unit and, 93
- Subprogram of modular programming, 7, 135
- Subranges, 85

- Subtraction sign, 212
- Succ function, 176
- Summing loops, 118–119
- Swap function, 177
- SwapWord function, 177
- SWSynthRec. *See* Square wave synthesizer tone generation
- Symbol, 6
 - ;ca, 240
 - decision, 10–12
 - flowchart, 8–12
 - input/output, 8–10
 - three dots, 262–263
- Symbol table, 42
- Syntax
 - definition of, 6
 - errors in, 41
 - structure and. *See* Structure and syntax
- System
 - flowchart and, 8
 - RAM disk and, 20
- System error, 103
- System Error Messages, 41
- T**
- TADelete, 326
- TAInsert, 326
- TANext, 326
- Tape counter program, 199–201
 - explanation of, 200–201
- TAPrev, 326
- TAReset, 326
- TAUpdate, 326
- TempNumber, 197
- Terminal symbol, 8
- Termination condition of repeat statement, 126
- Text files, 260
- TheButton, 281
- TheEvent, 293–294
- Tile Windows, 38, 39
 - Format menu and, 38
- TMON, 268
 - debugger, 256
 - window-based, 268
- To Disk, 41
 - double-clickable application in, 41
 - saving program and, 50
- To Memory, 41
- Tone
 - amplitude field of, 286
 - arrays, 285
 - type, 285
- Toolbox numerical methods, 327
- Top-down design, 135
 - problem solving by, 7
 - procedure and, 138
- Total record size, 253
- TotMinutes, 200
- TotSeconds, 201
- Transfer menu, 44, 45
 - dialog box for, 29, 30
 - file menu and, 29
- Transfer option, 31
 - dialog box for, 31
- Trees, 230–233
- True condition, 106
- True or false
 - condition, 114
 - decision making and, 103–104
- Truncation function, 168
- Turbo compiler with semicolons, 77
- Turbo Pascal
 - arithmetic, 79–82
 - communicating with Macintosh, 12–14
 - compiler, 13
 - complete program in, 181
 - creating binary tree and, 233
 - database toolbox, 325–326
 - disk icon, 20, 21
 - editing of program for, 48–50
 - first program for, 44–47
 - getting acquainted with, 24–55
 - getting ready to use, 18–20
 - history of, 5
 - introduction to, 5
 - library features in, 165–178
 - looping structures in. *See* Looping structures
 - making music in, 285–290
 - menu discussions of, 25–44
 - parameters in. *See* Parameters
 - printing program for, 52–54
 - procedures in. *See* Procedure programs in. *See* Programs
 - running the program for, 47–48
 - saving the program for, 50–52

- Turbo Pascal (*cont.*)
 - screen, 25
 - starting, 20–22
 - structure and syntax in. *See* Structure and syntax
 - toolbox numerical methods, 327
 - tutor, 327–328
 - unit, 157–158
 - units in. *See* Unit
 - Turbo Pascal Access, 325, 326–327
 - Turbo Pascal Desktop, 22
 - TurboPort, 277
 - TurboSort, 325
 - Turbo X.X program disk file, 18
 - TurnLeft, Turtle graphics and, 272
 - TurnRight, Turtle graphics and, 273
 - TurtleDelay, Turtle graphics and, 273
 - Turtle Folder file, 18
 - Turtle graphics, 271–275
 - Turtle routines, 273–274
 - Turtle unit, 271–275
 - Tutor, Turbo Pascal, 327–328
 - Type declaration, 83, 285
 - for parameters, 144
 - Type of function, 148
 - Types, 204–206
- U**
- \$U. *See* Unit file directory
 - \$U compiler directive, 157
 - UCSD Pascal, 5
 - Underscore, 62
 - Undo, 32
 - Unit, 157–162
 - compiling, 158
 - modular programming and, 11
 - PasConsole information and, 159–160
 - structure of, 158
 - Turbo Pascal and, 93, 157–158
 - UnitMover and, 160–162
 - uses clause and, 158–159
 - Unit file directory, 43
 - UnitMover, 160–162
 - dialog box for, 160, 161
 - invoking, 160, 161
 - utilities and sample programs file of, 18
 - UnitMover window, 160, 162
 - University of California at San Diego
 - Pascal version, 5
 - UnPack program disk file, 18
 - User-defined data types, 83–85
 - Uses clause, 158–159
 - Uses statement, 277
 - Utilities and Sample Programs Disk, 18
- V**
- Valid assignment statement, 74
 - Valid expression, 77
 - Valid identifier, 62
 - Valid integers, 66
 - Valid response, 100
 - Valid variable declaration statement, 73
 - Value parameters, declaring, 145
 - Values
 - ending, 126
 - midpoints of, guess-a-number program and, 185
 - parameters for, 145–148
 - procedure and, 139
 - starting, 126
 - Var. *See* Variables
 - Variable declaration statement, 73
 - Variable identifier, 74
 - Variable name, 99
 - Variable number, 202
 - Variable parameters, 141–145
 - Variables
 - assigning, 74–75
 - declaring, 72–73, 140, 210
 - double real, 203
 - global, 185
 - global versus local, 140–141
 - index, 119
 - MenuHandle type, 297
 - problems with, 256
 - reading, 100–102
 - readln and, 100
 - standard real, 203
 - Variant block, 253
 - Variant records, 251–253
 - Var pointer, 224
 - Vertical position on Mac screen, 94

ViewDates, 307
View the Next Month of Events Option,
304
Visible window, 292

W

While loop, 114, 129, 185
drawing freehand with, 284
While statement, 113, 114–116
While versus repeat...until loops, 129
Window
defining a, 291
File menu and, 27
go-away box, 292
invisible, 292
resizing, 281
Search menu and, 35, 37
visible, 292
Window-based TMON, 268
Window manager program, 278–289
WindowPtr types, 280
WindowRect, 280
Words only with Find What, 34
Wrap, Turtle graphics and, 273

Write, 237
output symbol of, 9
semicolon and, 90
statements and, 89–97
execution of, 90
single, 92, 93
Write-line statement, 47
Writeln, 64
display, 96–97
statements and, 89–97
using, 91–93
Writing a file, 238

X

XCor, Turtle graphics and, 273

Y

YCor, Turtle graphics and, 273–275

Z

Zoom window, 38

The Official Book on Turbo Pascal for the Macintosh

"When you have completed reading COMPLETE MACINTOSH TURBO PASCAL, you will have enough knowledge of our powerful compiler to create sophisticated programs which utilize the Macintosh environment. Borland is pleased to endorse COMPLETE MACINTOSH TURBO PASCAL as the 'official' book on Turbo Pascal for the Macintosh."

From the Foreword by Philippe Kahn

An all-in-one reference guide and tutorial, **Complete Macintosh Turbo Pascal** provides everything you'll need to begin programming in Pascal with Borland's compiler.

Complete Macintosh Turbo Pascal lets you build your programming skills from the ground up. The book starts with a thorough explanation of programming and the Pascal language and progresses through advanced topics. Experienced programmers will welcome this book's complete coverage of:

- Debugging
- Assembly Language
- File dumps
- Resource files
- Windows
- Event handling
- Mouse programming
- Graphics
- Using other Borland products with Turbo Pascal

Complete Macintosh Turbo Pascal provides examples with each topic and offers several complete programs, including a date minder and a record album database program.

Whether you're a novice or an experienced programmer, **Complete Macintosh Turbo Pascal** is your key to enjoying the full power of Borland's sophisticated compiler with your Macintosh.

Joseph Kelly is a systems analyst and the author of several well-received computer books.

ISBN 0-673-38456-X

