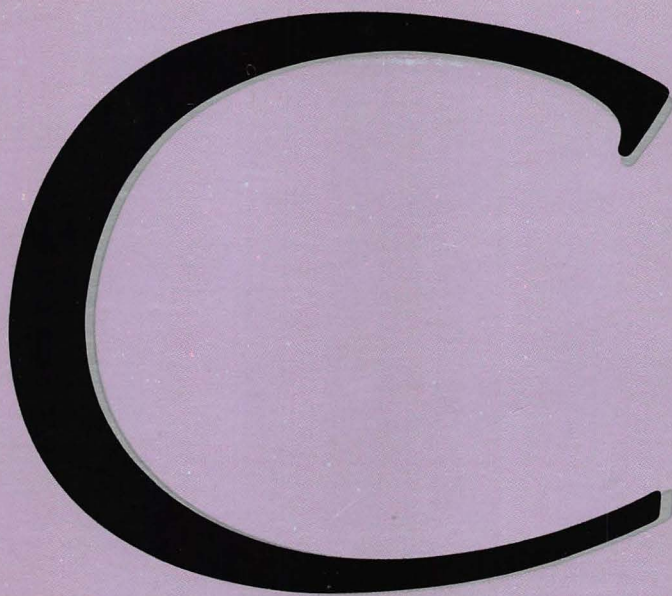


MACINTOSH



PRIMER PLUS

THE WAITE GROUP/STEPHEN W. PRATA



Macintosh™ C Primer Plus

Bantam Computer Books

Ask your bookseller for the books you have missed

THE AMIGADOS MANUAL
by Commodore-Amiga, Inc.

THE APPLE IIc BOOK
by Bill O'Brien

THE COMMODORE 64 SURVIVAL MANUAL
by Winn L. Rosch

COMMODORE 128 PROGRAMMER'S REFERENCE GUIDE
by Commodore Business Machines, Inc.

EXECUTIVE GUIDE TO THE EPSON GENEVA
by Marshall P. Smith

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR APPLE II
by Tim Hartnell

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR COMMODORE 64
by Tim Hartnell

EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR IBM PC
by Tim Hartnell

EXPLORING THE UNIX ENVIRONMENT
by The Waite Group/Irene Pasternack

FRAMEWORK FROM THE GROUND UP
by The Waite Group/Cynthia Spoor and Robert Warren

HOW TO GET THE MOST OUT OF COMPUSEVE, 2d EDITION
by Charles Bowen and David Peyton

HOW TO GET THE MOST OUT OF THE SOURCE
by Charles Bowen and David Peyton

THE MACINTOSH
by Bill O'Brien

THE NEW jr. A GUIDE TO IBM'S PCjr
by Winn L. Rosch

ORCHESTRATING SYMPHONY
by The Waite Group/Dan Shafer with Mary Johnson

PC-DOS/MS-DOS
User's Guide to the Most Popular Operating System for Personal Computers
by Alan M. Boyd

POWER PAINTING: COMPUTER GRAPHICS ON THE MACINTOSH
by Verne Bauman and Ronald Kidd/illustrated by Gasper Vaccaro

SMARTER TELECOMMUNICATIONS
Hands on Guide to On-Line Computer Services
by Charles Bowen and Stewart Schneider

SWING WITH JAZZ
Lotus Jazz on the Macintosh
by Datatech Publications/Michael McCarty

TEACH YOUR BABY TO USE A COMPUTER
Birth Through Preschool
by Victoria Williams, Ph.D. and Fredrick Williams, Ph.D.

UNDERSTANDING EXPERT SYSTEMS
by The Waite Group/Mike Van Horn

USER'S GUIDE TO THE AT&T PC 6300 PERSONAL COMPUTER
by David B. Peatroy, Ricardo A. Anzaldúa, H. A. Wohlwend, and Datatech Publications

Stephen Prata
The Waite Group

MacintoshTM **C Primer Plus**



BANTAM BOOKS
TORONTO • NEW YORK • LONDON • SYDNEY • AUCKLAND

MACINTOSH C PRIMER PLUS

A Bantam Book/March 1986

Apple, LaserWriter, MacDraw, and MacPaint are registered trademarks of Apple Computer Inc.

Macintosh is a trademark licensed to Apple Computer Inc.

Hippo C is a trademark of Hippopotamus Software, Inc.

Microsoft Word is a registered trademark of Microsoft Corporation

UNIX is a trademark of AT&T Bell Laboratories

Throughout the book, the trade names and trademarks of some companies and products have been used and no such uses are intended to convey endorsements of or other affiliations with the book. Moreover, the reader should not assume Apple or Hippopotamus Software have in any way authorized or endorsed this book, which is the result of the authors' own research and analysis.

All rights reserved.

Copyright © 1986 by The Waite Group, Inc.

This book may not be reproduced in whole or in part, by mimeograph or any other means, without permission.

For information address: Bantam Books, Inc.

ISBN 0-553-34197-9

Published simultaneously in the United States and Canada

Bantam Books are published by Bantam Books, Inc. Its trademark, consisting of the words "Bantam Books" and the portrayal of a rooster, is registered in the U. S. Patent and Trademark Office and in other countries. Marca Registrada. Bantam Books, 666 Fifth Avenue, New York, NY 10103.

PRINTED IN THE UNITED STATES OF AMERICA

BH 0 9 8 7 6 5 4 3 2 1

Dedication

To June 8.

Contents

Acknowledgments

Preface

1	C as a Macintosh Programming Language	1
	C and You	2
	This Book Is For...	3
	Macintosh C Versus Generic C	3
	How Computer Languages Work	6
	Which C Compiler?	9
	Starting Off in Hippo C	10
	What's Ahead	14
2	C Basics	17
	C and Functions: The Modular Approach	17
	Variables and Declarations	23
	Operators	27
	Expressions and Statements	33
	More on Functions: Arguments and Return Values	35
	Summary	42
3	I/O Functions and Types	43
	Character I/O	44
	The Character Type	52
	The C Preprocessor – A First Look	59
	Other C Types	62
	Formatted Output: printf()	67
	Formatted Input: scanf()	74
	Toolbox Examples	79
	Summary	84
4	Control Statements	85
	The while Loop	86
	The do...while Loop	88
	The for Statement	89
	Other C Assignment Operators	92
	Nested Loops	93
	Two if Statements	96
	Conditional Expressions	99
	Other Jumps	109

	More Operators	116	
	Summary	118	
5	Functions		119
	Review	119	
	Arguments	120	
	Return Values	125	
	Return Values and Functions Types	126	
	Type Conversions and Type Casts	131	
	Using Addresses and Pointers	134	
	Scope: Local and Global Variables	144	
	Recursion	151	
	Macros	155	
	Summary	160	
6	Arrays and Structures		163
	The Array	164	
	Structures	178	
	A Quickdraw Structure	181	
	Functions, Structures, and Pointers	188	
	Friendly Advice	191	
	Summary	192	
7	Compound Data Structures		195
	Arrays of Structures	195	
	Complex Declarations and typedef	210	
	Function Pointers	215	
	Summary	217	
8	Character Strings		221
	Character Strings	221	
	String I/O	225	
	String Functions	235	
	Writing C String Functions	240	
	Macintosh Pascal Strings	249	
	Quickdraw Functions	251	
	Programming With Pascal-Formatted Strings	256	
	Summary	257	
9	C and the Macintosh Toolbox		259
	Pascal Procedures and Functions	259	
	Pascal Types	264	
	Back to Quickdraw	271	
	Patterns	281	
	The Cursor	288	

One More Example	291	
Summary	295	
10 A Mac Miscellany		297
The Macintosh Software System	297	
Macintosh Memory Management	298	
Regions	309	
Events	319	
Files	329	
A Soundmouse Experiment	338	
Summary	341	
Conclusion	342	
A Keywords in C		345
B Operators		347
C Binary, Octal, and Hexadecimal Numbers		355
D Bit Fiddling		361
E Macintosh ASCII Table		369
Index		377

Acknowledgments

I would like to thank Jerry Volpe of the Waite Group for his editorial guidance, Mitchell Waite of the Waite Group for his support, and Rick Oliver of Hippo Software for answering questions.

Preface

C is undoubtedly the most popular language for serious software development on microcomputers today. C's syntax is direct, compact, and easy to learn, and is well suited to an efficient, modular style of program development. Because of its syntax, and because it is a compiled language, C translates into programs that run quickly and use a minimum of memory. C's modularity also makes it easy to adapt a particular program to run on many different computers. On the Macintosh, C provides a direct and efficient way to access the Mac's built-in Toolbox and Quickdraw routines, which are essential for writing "Mac-like" programs that make use of the Mac's outstanding graphics and features such as windows, menus, and the mouse.

This book is written for two groups of readers. If you have never programmed in C before, this book provides a complete introduction to the language, using simple examples and a step-by-step approach. You'll find it helpful to have at least a nodding familiarity with some other computer language such as BASIC or Pascal — it is possible to learn C as a first language, but C has so many advanced features it's helpful if you've gotten your feet wet with a less complete language. If you are already experienced in C, this book will teach you how to use the language effectively in the Macintosh environment. The Mac is so different from other computers that using C to access the Mac's advanced features forms a major part of the book; C programmers will find it a whole new world.

The program examples in this book are based on the Hippo C Level 1 compiler, which is a product of Hippopotamus Software, Inc. of Los Gatos, CA. As we discuss more fully in Chapter 1, we chose Hippo C because we found it to be an excellent first choice for anyone learning to program in C. It is easy to use, provides a full implementation of C, has a superior method of dealing with error messages, provides direct access to the Toolbox routines, and uses an easy MacWrite-style editor. However, we should emphasize that this book will work with any of the popular C compilers on the Mac. Where there are differences between Hippo and standard C, such as the use of floating point, we point them out, and explain how the standard version works.

This book covers the complete C language, with many program examples that draw extensively upon the Macintosh's own built-in Toolbox routines. Certain topics that are essential to using the Toolbox, such as functions, pointers, and structures, are emphasized more heavily than in an introductory C book. By the time you finish this book you will have learned the C language, and you will also have seen enough of the Toolbox to be comfortable using it, and to be able to figure out how to use those routines not covered here. Thus, if your goal is to learn C, this book will meet that goal. If your goal is to develop large-scale Macintosh programs, this book will give you the background you need and point you in the right direction.

1

C as a Macintosh Programming Language

In this chapter you will learn about:

- Libraries
 - The Macintosh toolbox
 - Compiled languages
 - Using Hippo C
-

The Macintosh is a fascinating and innovative device. Its system of *mouse-driven menus*, *windows*, and *integrated graphics* has redefined how a computer should work. Once you have experienced the ease of using a Mac, it is difficult to go back to the staid keyboard-input approach of most other computers. The natural response of many of us after using the Mac is to want to write programs for it. But if *using* the Mac is simpler than using most computers, *programming* for it is more difficult, at least if you want your programs to have the Macintosh look to them. Fortunately, the Mac memory banks contain a fabulous selection of built-in software specifically designed to place the Mac's unique features at the programmer's service.

One of the best ways to program for the Macintosh is to use the *C language*. C certainly is the choice of the majority of the independent software developers working on Macintosh programs. If you know a little about the Macintosh history, you may find this a bit surprising, for an extended form of Pascal was the original development language for the Macintosh. Also, the built-in software routines are written to be accessed as Pascal procedures and functions. But Pascal originally was developed as a teaching language, while C was developed as a working programmers' language. To make Pascal into a suitable development tool, Apple had to extend and modify the language so that it could accomplish things that C does naturally. Thus, Macintosh Pascal is not the same as standard Pascal, but C on the Macintosh is no different, on the whole, from C on any other computer. And C programs can use the built-in Macintosh software.

Pascal and C Roots

Pascal and C, two of the most successful computer languages, are each the result of the work of one man (but not the same man!) rather than of a committee. This gives them a sharper focus than what a diffuse consensus usually produces.

Niklaus Wirth, the Swiss computer scientist, developed Pascal to teach sound programming practices. It emphasizes an organized, disciplined, modular programming style. It uses the structured control statements favored by computer science, and offers great flexibility in the representation of data. Because it is meant to teach good programming technique, Pascal is intolerant of deviations from its standards.

Dennis Ritchie, a system software specialist at Bell Labs, developed C as part of the UNIX operating system development program. Thus, C is a working programmer's language. It offers many of the same features as Pascal (both are ultimately descend from the European computer language ALGOL), but it is a little more tolerant. It gives the programmer more control than Pascal, but it also requires that the programmer exercise more responsibility.

C and You

What makes C a good programming language? It has a modular design that makes it easy to break down a complex program into easily manageable parts. It has a rich selection of operators that allow you to express yourself succinctly. Its structured statements guide you into structured programming, a technique that increases program readability and reliability. It lets you store a large program package over several files without any special effort. It offers you a fine control over program details. And it produces compact, efficient programs that run swiftly.

We've established that the Mac is a desirable computer and that C is a desirable language. This book aims to show you how to program in C on the Macintosh; we hope that makes this a desirable book. The discussion of C begins in Chapter 2, but first we have some preliminary matters to deal with. One of the most important is outlining what we expect of you.

This Book Is For ...

As you must have guessed, this book is for someone who wishes to learn C on the Macintosh. But this general description includes many

possibilities. You might be expert in C, but a Macnovice. Or you could be a Macintosh programmer in some other language who wants to see how to do it in C. Or you might be unpacking your first Macintosh while wondering what a programming language is. Here's what we assume about you:

1. You have used a Macintosh. You know how to move the mouse cursor around, how to select menu items and windows, how to change a window size, how to make minimal use of an editor, such as Macwrite.
2. You are not familiar with Macintosh's built-in software routines.
3. You may have done some programming but don't know C.
4. You have access to a Macintosh and to a C compiler for the Macintosh. Our examples use the Hippo C, Level 1 compiler. You may have to make some adjustments if you use another compiler.

The main emphasis of the book is presenting the C language, and the secondary emphasis is exploring the Macintosh's built-in software routines. We will cover all the essentials of C, but only cover part of the Macintosh software. Since it would take at least two books this size to present the whole Macintosh package, this limited coverage is necessary. The parts we do cover, however, are interesting and lay the groundwork for understanding the whole.

Let's look more closely now at what makes learning C on a Macintosh different from learning C in another environment.

Macintosh C Versus Generic C

Language designers face two conflicting demands. First, a language should be portable. That is, it should let you write programs that can run on a variety of machines. Second, a language should let you use the special capabilities of a particular computer. A Mac program, for instance, should be able to use the mouse.

C has an admirable record for portability. It has done so by keeping the core language limited. The number of "keywords," words reserved to

have special meaning to the language, is much smaller for C than for Pascal or modern BASIC. Many things that are built-in statements for other languages are handled in C by "functions." A function is just a separate programming module that can be incorporated easily into a program. C implementations come with libraries of functions to handle many basic tasks, including taking input from the keyboard and printing output on the screen. When C is moved to a new system, the language is left intact, and the library is rewritten as necessary in order to function on the new system.

The function system used by C makes it easy to incorporate the special capabilities of the Macintosh. All that is necessary is to make the Macintosh software routines available as C functions.

In brief, then, the C language you will learn in this book is the same C you would learn anywhere else. There are no new keywords, no need for nonstandard extensions. However, with the inclusion of its software routines as functions, the Macintosh offers an incredibly rich library from which C programs can draw. Programs that use only the standard library should be quite portable. Programs that use the special Macintosh software routines are not portable. Let's look further at the library.

The Function Library

First, there is a standard library of C functions that can be found in all but the smallest implementations of C. At present, the library is standard in the sense that it represents a common consensus among the many C implementations; eventually we may have an official committee-legislated standard. Not all implementations offer exactly the same library, but the agreement is good. The **Hippo C**, Level 1 implementation, for example, offers about 50 library functions that would be found in other implementations.

What do these functions do? They handle input and output in a variety of manners: a character at a time, a line at a time, formatted numbers, and so on. They manipulate "character strings," which are sequences of characters treated as a unit. They open and close files and handle file input and output. They manage computer memory, print error messages, and determine the nature of individual characters. You'll see many of these functions as you go through the book. These are portable functions; you should be able to use them (most of them, at least) on any C system.

Next, we have the Macintosh built-in software routines. There are approximately 500 of them. The majority of them belong to what is called the "Macintosh Toolbox." The remaining ones belong to the Operating

system. We'll loosely refer to all the routines as the Toolbox, for both groups are equally accessible as C functions.

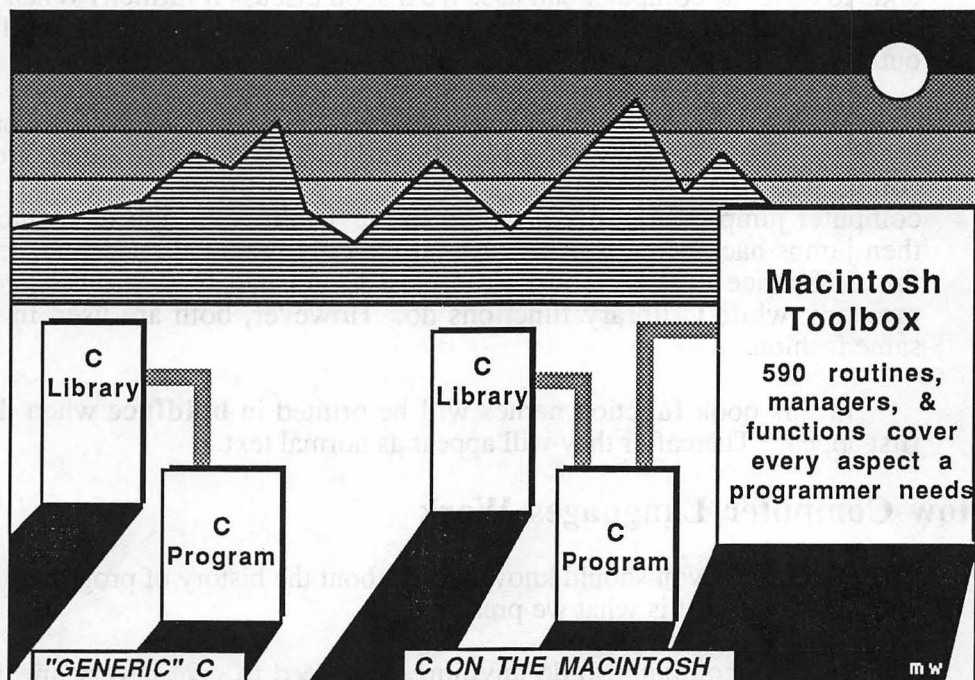


Figure 1.1 Macintosh has superior resources

These functions are intended to provide the programming tools you need to produce programs in the Macintosh style. They are subdivided according to use into several packages, typically called *managers*. For example, the Window Manager is a group of routines used to control windows, and the Menu Manager relates to menus. This book concentrates on Quickdraw (sometimes called the Screen Manager) for examples. Quickdraw handles output to the screen, both text and graphics, making it an interesting and important package to deal with. Also, it offers illustrations of many concepts basic to Macintosh programming.

In addition to these two large sources of functions, Macintosh C implementations will offer a few special purpose functions. For instance, C and Pascal use different formats for storing character strings, so C implementations generally provide functions to convert one format to the other.

There is an important technical difference between the standard C library and the Toolbox. The C library comes with the C compiler and is stored on a disk. (The compiler is the program that converts your written C code to code the computer can use. We'll soon discuss it further.) When the compiler puts a program together, it searches through the library, pulling out the particular code you need and inserting it into your program.

The Toolbox, however, is built into the Macintosh as ROM (read only memory). Because they are always available, Toolbox routines aren't copied into your program. Instead, when a Toolbox routine is called, the computer jumps from your program to the Toolbox section of memory, then jumps back when finished. About the only practical consequence of this difference is that Toolbox routines do not add to the size of your program, while C library functions do. However, both are used in the same fashion.

In this book function names will be printed in **boldface** when they first appear. Thereafter they will appear as normal text.

How Computer Languages Work

To appreciate C, you should know a little about the history of programming languages, and that is what we present next.

To get a computer to do anything, you need to give it very specific instructions. These instructions must first be stored in the *computer memory*. The basic unit of computer memory is called a "bit," and it can store either a 1 or a 0. That's not much, but computers have oodles of bits, so they can store lots of 1s and 0s. Anyway, this implies that the instructions we give to a computer have to consist of some sort of code formed from 0s and 1s. For example, you might tell a computer to "00010010 01010010." This may sound like a tiresome, grungy, error-prone ("Did I say 01010010? I meant 01001010!"), time-consuming process. It is, but it was what the first programmers had to do. It's called "machine-language" programming, and, as you might expect, different machines have different languages.

The next step up was to replace machine-language code with *mnemonic* representations, using terms like CLR, MOVE, and JMP to stand for particular machine codes. Instructions in this new form were called "assembly language." A special program, called the "assembler" was devised to convert assembly language to machine language. Now

programmers would write a program in assembly language and submit it to the assembler to get the machine-language equivalent. The resulting machine code was called "object code." A full program might consist of several blocks of object code that would be "linked" together by another program, called the "linker" to produce the final machine-language program.

This is still a lot of work, and assembly language is just as machine-dependent as machine language. Still, even today, many programmers use assembly language because it provides the most compact and efficient programs.

The next great step towards making programming less machine-dependent and easier to use was the development of the "compiled" language. Here FORTRAN led the way. The idea is to develop a language that is problem-oriented instead of machine-oriented. FORTRAN allows programmers to set up programs more along the lines of algebraic equations and formulas than as specific computer instructions. Then a special program called the "compiler" translates the program to assembly language.

The compiled language approach has two tremendous advantages. First, it makes programming easier, since the language relates more directly to problems being solved. Secondly, a compiled language can be used on many different varieties of computer. All that is necessary is to write a separate compiler for each type of computer. Examples of compiled languages are FORTRAN, COBOL, Pascal (except Macintosh Pascal), and C.

One other strategy has been developed, that of the "interpreter." This is the approach used by BASIC and Logo, for example. Like the compiler, the interpreter is a program. However, it does not translate language statements into assembly language. Instead, it acts upon them directly. For example, if you say `PRINT "HOWDY"` in BASIC, the interpreter sees the word `PRINT` and then prints the word "Howdy" itself. Compared to the compiled approach, the interpreter is much more direct. To print "Howdy" in a compiled language, you would write a simple program, compile, assemble, link, and run the program. This might take a couple of minutes on a MAC. However, once they are put together, compiled programs run much faster than interpreted ones. In short, interpreters make program development simpler because you get immediate feedback, but compilers produce a superior final product. C uses the compiler approach.

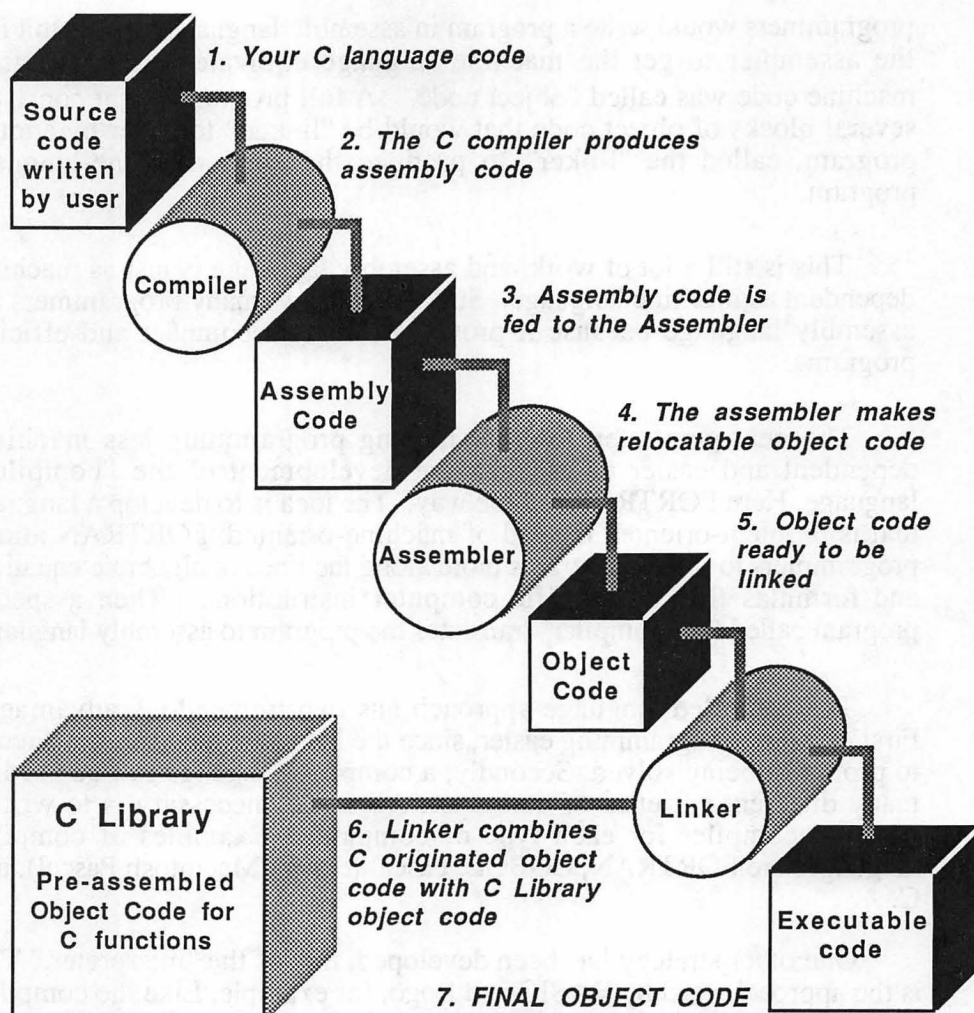


Figure 1.2 Using a compiled language

Which C Compiler?

Currently several companies make C compilers for the Macintosh. They all implement pretty much the same standard C language, but there are differences in the processes you must go through to set up and run a

program, as well as differences in some language aspects that C leaves open. In order to develop consistent examples, we had to choose one compiler to work with. After looking at several possibilities, the Waite Group staff chose Hippo C, Level 1. That doesn't mean you have to use the same compiler, but you may have to make some adjustments if you don't. Let's look first at the reasons for our choice, then at some of the differences between compilers.

Most of the C compilers for the Macintosh are intended as development systems for application programmers. That is, the projected user is an experienced software programmer with some prior knowledge of the Macintosh. Putting together even a simple program often involves going through several steps. Hippo C, Level 1, on the other hand, is much more suitable as a learning vehicle for C. Many steps are done automatically, so that it only takes a push of the mouse button to prepare an executable program. Also, it is one of the least expensive compilers.

Level 1 does have some limitations. For example, it doesn't access about 100 Toolbox routines. Since that still leaves about 400 to play with, that is no real problem for someone learning C. Second, it doesn't offer floating-point numbers, that is, numbers like 3.14159 or 2.5. The Toolbox mainly uses integer numbers, so the lack of floating-point is not a problem for the examples we use. (Hippo C also offers Level 2, which is a complete application-level development system accessing the whole Toolbox and allowing floating-point numbers.)

What differences can you expect if you are using another compiler? Probably the main difference will be the steps you have to go through to run a program. We'll run through the steps for Hippo C, but you will have to check the manual for other compilers. A second difference may be in the names used for the Toolbox routines. The Pascal descriptions used in the Apple manuals use names like **DrawChar**. Hippo C and many other C compilers transliterate these to pure lowercase: **drawchar**. Others retain the mixed uppercase and lowercase of the original. Third, different compilers may provide slightly different implementations of the C library functions. Fourth, the library contents are not exactly the same; one compiler may offer functions missing from another. Fifth, C leaves some choices, such as how many bits of memory will be used to store a number, open to the implementer; and not all have made the same choices.

We will point out where compiler differences might matter as they come up in the text.

Starting Off in Hippo C

Let's quickly run through the steps you would go through to prepare and run a program using Hippo C, Level 1. First, you would turn the Mac on and load the Hippo disk. You should obtain the typical Macintosh contents window showing what is on the disk. Select the Hippo C icon (the hippo face labeled "Hippo C" — you can't miss it). Eventually, you get a window called "Untitled." This is a blank area in which you can write your program. Just type it in from the keyboard. You can move the cursor with the mouse, and you can use the Edit menu to make editing changes. The Hippo manual describes the editor more fully, but if you've used Macwrite, you can use the Hippo editor. When finished, you can use the File menu to select SAVE. This gives you an opportunity to choose a name for the file, and it stores the file on disk. The name you choose must end with a ".c" so that the compiler can recognize it as an official C program. C distinguishes between upper and lower case, so make sure you use a lowercase c. At this point, the screen may look like Figure 1.3.

Next, use the mouse to open the Programs menu. Move the mouse down to the program title (**greet.c**, in this case) and click it. This puts a check by **greet.c**, identifying it as the file you wish to compile. Figure 1.4 shows this stage.

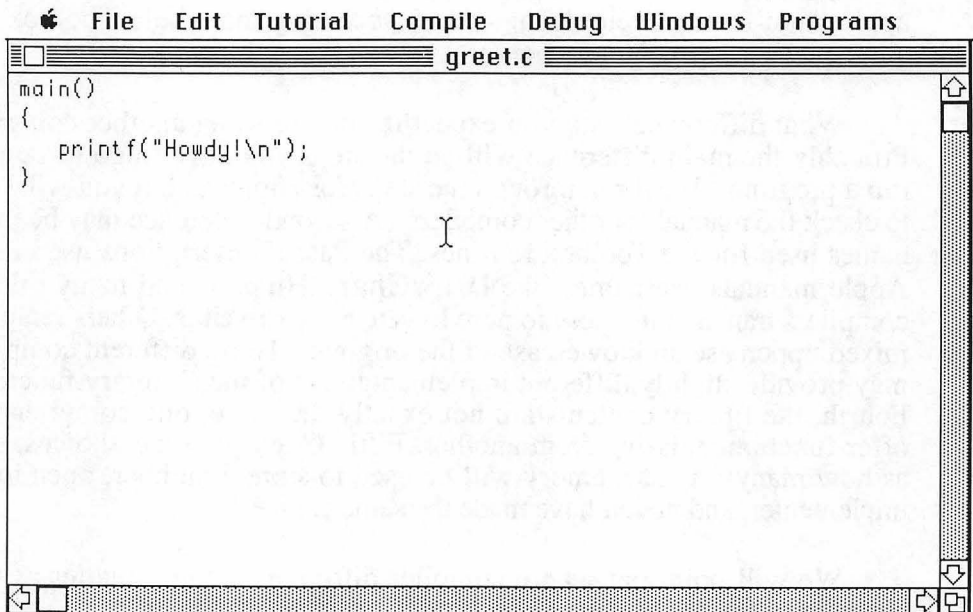


Figure 1.3 The greet.c program

There are two points to remember about selecting a file. First, the effect is cumulative. If you later select another file, the original file is still selected also, *unless* you click it to remove the check mark. (This feature makes it possible to compile a multi-file program.) Second, you are selecting the saved file, not the one showing on the screen. That is, if you change the screen without saving it, that has no effect on the compilation process. When you correct or modify a program, always save it before compiling it. If you get a message saying that **a.out** is up to date, you probably forgot to save your modifications.

Now it is time to compile. Actually, the computer must go through several steps. First, the **greet.c** file is compiled into an assembly code file called **greet.s**. This file is assembled into a machine language, or object, file called **greet.o**, and the **greet.s** file is removed. Then the **greet.o** file is linked with any library routines required, and the resulting executable program is placed in a file called **a.out**. If we started with just one file, as in this case, the object file is erased.

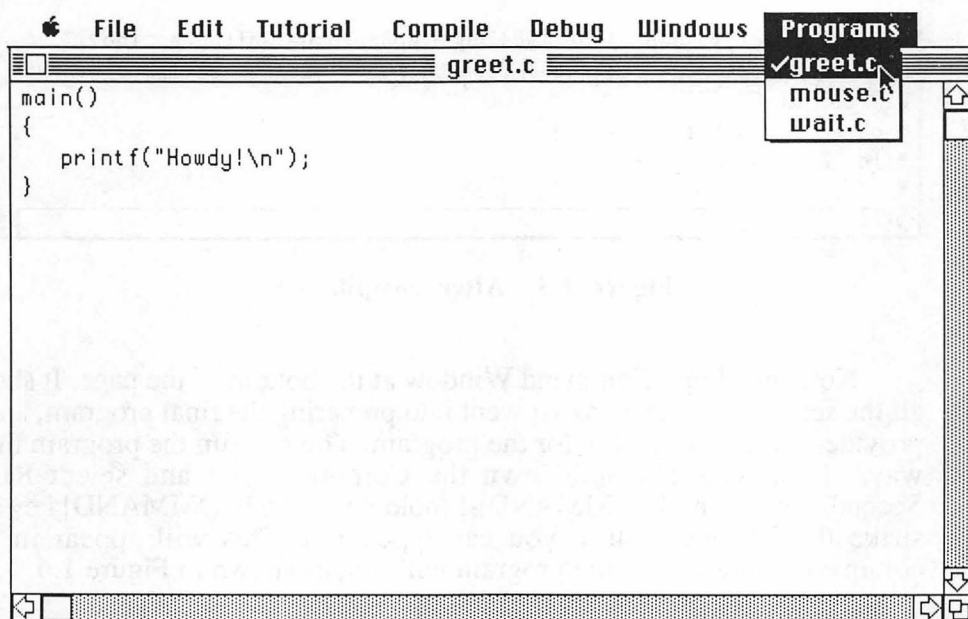


Figure 1.4 Selecting **greet.c** for Compilation

With Hippo C, we accomplish this entire sequence by pulling down the Compile menu and selecting the COMPILE choice. The process will take a minute or so. Every now and then, you will see one of the intermediate commands appear near the bottom of the screen. When it finishes, the screen will look like Figure 1.5.

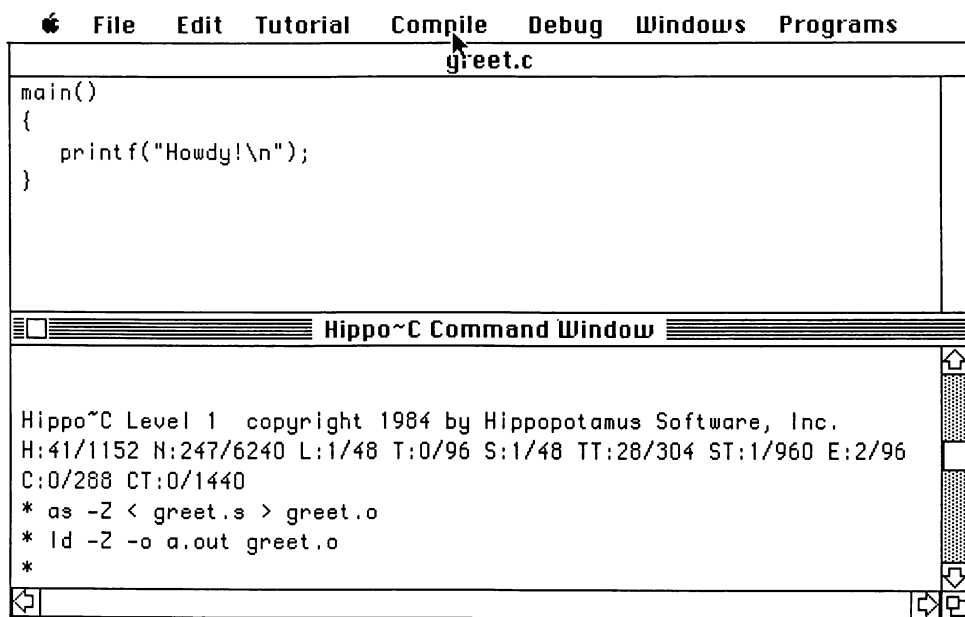


Figure 1.5 After compilation

Note the Hippo Command Window at the bottom of the page. It shows all the separate commands that went into preparing the final program, and it provides the area of action for the program. You can run the program three ways. First, you can pull down the Compile menu and select RUN. Second, you can hit [COMMAND]-i (hold down the [COMMAND] key and strike the [i] key). Third, you can type a.out. This will appear in the command window, and the program will run, as shown in Figure 1.6.

Instead of using the Compile menu to put the program together, you can use the Window menu to select the Hippo Command Window and then

type out the commands shown in Figure 1.5. This approach allows you to select other compiler options discussed in the Hippo C manual.

As you can see, it is pretty simple to prepare and run a program using Hippo C. We'll refer to this mode as the "Hippo C environment." This environment conforms to the traditional Macintosh pattern with its use of windows, menus, and mouse.

Hippo C offers another environment for program development. It's called HOS (for Hippo Operating System), and is accessed from the File menu by selecting QUIT TO HOS. (There is also a QUIT TO FINDER choice for returning to the regular Macintosh environment.)

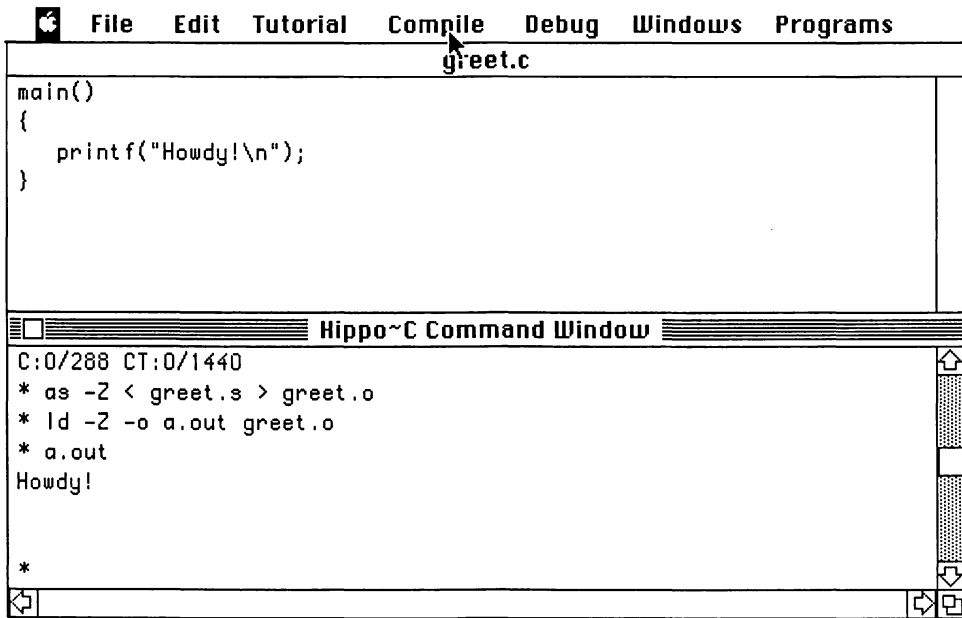


Figure 1.6 Running the program

HOS provides the old-fashioned keyboard-screen environment. You can't edit programs in it, but you can compile and run them, and you can

use several HOS commands described in the manual. These imitate certain UNIX commands, such as `cat`, which shows and combines files, and `mv`, which changes the name of a file. These commands can also be run from the Hippo Command Window in the Hippo C environment.

We won't use HOS too often, for the Hippo environment is much more convenient. However, the Hippo environment also takes up much more of the Macintosh's memory, so a program that produces an out-of-memory system failure in the Hippo C environment may run in HOS.

Again, if you are using another compiler, you will have to use its manual to guide you.

What's Ahead

The following chapters present the C programming language. Initially, we will present both portable examples and examples that draw upon the Macintosh Toolbox. As we progress, the emphasis will shift from learning C to seeing how to program the Macintosh in C. Thus the last two chapters are concerned mainly with exploring the Toolbox and gaining a basic understanding of how to use it.

To use the Toolbox, you need to know about functions, pointers, and structures. Therefore we introduce these topics earlier than is usual. Pointers and structures are considered advanced topics, but the rich rewards they provide when used with the Toolbox make it foolish to defer them to the closing chapters. And there is nothing like regular contact to make an "advanced" topic seem everyday.

So let's get started!

A Note About Style

Throughout this book, you are going to encounter samples of C program code. Because C is a free-form language, and has few rules about how to lay out a program, we have tried to emphasize the advantages of using an easily readable program structure. As you will find, a good structure makes maintenance and understanding of a program much easier. Unfortunately, our book pages could not present as many characters per program line as you can display directly on the Macintosh's screen. Consequently, in bringing some of the program examples into book form, certain lines of program code were longer than we could easily fit on a page, so they had to be wrapped over two lines. Where a break in a line of code might cause a syntax error, we have marked the line with an arrow which curves down to the next line like this:

```
printf ("In this last example there were %d  
characters.\n", count);
```

In some cases, comment lines have wrapped around as well. As you will see later, comments are set off from code with slashes and asterisks (`/*, */`). While putting the comments on two lines as you try out the programming examples won't affect how the program runs, you will find that the structure of a program is generally more apparent if you keep your comments lined up, and on a single line. Here is an example of a wrapped comment line:

```
int ch;    /* This is the preferred declaration  
           for EOF check */
```

Finally, you should note that we sometimes use special block separators like these:

From Mac's Toolbox: New Routines

to indicate special points of interest or additional background information.

2

C Basics

In this chapter you will learn about:

- **The role of functions**
 - **Using standard, Macintosh, and user-designed functions**
 - **C variables: declarations and types**
 - **Expressions and statements**
 - **C operators**
 - **Function arguments and return values**
-

The best way for most people to learn a language (computer or spoken) is to use it; but to use it, one needs to know at least a little about several different aspects of the language. For example, to speak English, it is much more useful to know a few nouns, verbs, prepositions, and adjectives than it is to have mastered two thousand verbs and nothing else. In this chapter, we'll introduce several features of C. Many will require deeper study later, but you will see enough to let you write simple, yet instructive, programs.

C and Functions: The Modular Approach

One of the most influential and useful methods to emerge from computer science is top-down programming. Top-down programming is the rather sensible approach of breaking a large programming job into smaller, more manageable tasks. Each of these tasks, if necessary, can be further subdivided until one large programming problem is broken down into many simpler, easily programmable modules. One module, perhaps, may deal with getting input from the user, another may sort the data, another may process the data, and another may present the results in an attractive manner. If one part of a program doesn't work satisfactorily, then the relevant module can be modified, leaving the rest of the program unchanged. This greatly aids the development and maintenance of healthy programs.

Some languages, such as C and Pascal, are particularly well suited to the top-down approach because they allow and encourage the development of separate programming modules. In C these modules are termed *functions*. A typical C program would consist of several functions along with some programming to tie them together. These functions may be created by the programmer, or they may come from a library of functions provided with the language. A large number of functions have become accepted as part of the standard C package. Beyond that, functions are the key to harnessing the special powers of the Macintosh, for Macintosh C systems provide hundreds of special functions for that purpose. Therefore it is a good idea to get familiar with the ideas of functions as quickly as possible.

The main() Function

A C program consists of one or more functions, and every C program contains a function called **main()**. In C, function names are followed by parentheses. The parentheses may be empty, as in this example, or they may contain information to be used by the function. The **main()** function is written by the programmer. When a program uses more than one function, **main()** is the first function activated. It typically provides the overall organization for a program, calling up other functions as they are needed. Here is a simple example; note that we use *lowercase*. C distinguishes between *UPPERCASE* and *lowercase*, so that a function called **Main()** or **MAIN()** is not the same as **main()**.

```
main()
{
    printf("Hey! Meet my main function!\n");
}
```

Braces ({ and }), sometimes called "curly brackets," are used to mark the beginning and end of a function definition, so this **main()** function consists of just one *statement*. In this case, the statement is a standard C output function, **printf()**. The semicolon at the end of the line identifies it as a complete C statement. We say that **main()** *calls* or *invokes* the **printf()** function.

The **printf()** (the "f" stands for "formatted") function then prints on the screen the message contained between the double quotes. **printf()** is called a "library function," meaning it is part of the C implementation. This is an example in which the function parentheses contain information used by the function. The **\n** at the end is not printed literally; instead, it causes the printer or screen cursor to move to the next line. For this reason, **\n** is called

function. The `\n` at the end is not printed literally; instead, it causes the printer or screen cursor to move to the next line. For this reason, `\n` is called the "newline" character. Just as an `H` character causes an `H` to be printed, a newline character causes a new line to be started. C has special symbols for several such "nonprinting" characters. These are characters that have no printed representation but that cause a screen cursor or printer to do such things as start a new line, move back a space, and so on. We will summarize them in Chapter 3.

As you can see, we didn't get very far without having to use a library function. C uses functions to handle input and output, so any program that uses input or output will use functions.

Toolbox Functions

Let's write a simple program that uses some of the specialized Macintosh functions. Here's one that deals with the Macintosh *cursor*, the pattern that indicates the mouse position. The `showcursor()` function makes it visible, and the `hidecursor()` function makes it disappear. (Recall that some implementations refer to these as `ShowCursor()` and `HideCursor()`.) As we introduce new Toolbox functions, we'll give a short summary like the following one. In it, we'll provide the routine name as it appears in Apple Macintosh literature.

From Mac's Toolbox: New Routines

HideCursor	This routine hides the cursor
ShowCursor	This routine shows the cursor

Now here is the program:

```
main()
{
    showcursor(); /* makes the cursor appear */
    hidecursor(); /* makes the cursor disappear */
    showcursor(); /* these are C comments */
}
```

Note that each statement ends with a semicolon. Here, the parentheses are all empty, for these functions, unlike `printf()`, need no further information to do their jobs.

We've used program "comments" here for the first time. In C, `/*` indicates the start of comment and `*/` marks the end of a comment. Comments are aids to the programmer and are ignored by the compiler. In C, a comment can be on the same line as code or on its own line or spread over several lines.

If you run this program, the hiding and showing happen so rapidly that they are hard for the eye to catch. We can slow the process down by using the `getchar()` function. This function, which is part of the standard C library, gets a character from keyboard input. When `getchar()` is encountered in a program, the program will halt until you hit a keyboard key. Once you hit a key, the `getchar()` function fetches it, and the program continues. Normally, the program will use the fetched character, and we will show how to do that later. For the present, however, we just use `getchar()` as a convenient way to halt a program until we are ready to move on. Here, then is the modified version:

```
main()
{
    showcursor();
    getchar(); /* program halts until a key is struck */
    hidecursor();
    getchar();
    showcursor();
}
```

Try this version. The cursor will show, and the program will wait until you strike a character key. The `getchar()` function will fetch the character, and `hidecursor()` will make the cursor disappear. Type another character, and the cursor reappears. Then the program ends.

Look at the program code again. All we have in it are functions! The `main()` function we defined, the `getchar()` function, came from the standard C library, and the `showcursor()` and `hidecursor()` functions came from the special Macintosh library of functions. With Hippo C we merely had to use the function name. Other C compilers may use other names (`HideCursor()` instead of `hidecursor()`, for example) and require additional steps to locate the functions, but they would still use the same basic program. Also, many

C implementations require that you include the file `stdio.h` before `getchar()` can be used; we'll discuss this file in Chapter 3.

One aspect of the `hidecursor()` function that didn't show up in this example is that there are different degrees of being hidden. If you use `hidecursor()` twice, it takes *two calls* to `showcursor()` to make the cursor visible, and so on.

User-Generated Functions

Suppose we want a program to use a function of our own design? The simplest way is to include the code for the new function in the same file as `main()`. Here is an example of what we mean:

```
main()
{
    printf("I am about to invoke the fabulous gronk
          function!\n");
    gronk(); /* a user-defined function */
    gronk();
    printf("Perhaps fabulous overstates the case.\n");
}

/* Here we define the gronk function */
gronk()
{
    printf("GRONK! GRONK! ");
}
```

Running this program produces the following output:

```
I am about to invoke the fabulous gronk function!
GRONK! GRONK! GRONK! GRONK! Perhaps fabulous overstates
the case.
```

Note that omitting a newline character in `gronk()` results in several `printf()` outputs being printed on the same line.

Looking back at the program, we see that `gronk()` is defined in the same manner as `main()`. The function name is followed by a parentheses pair. Braces mark the beginning and end of the body of the function. One

difference is that the compiler expects to find a `main()` function, but that it doesn't know about the `gronk()` function until it runs into it. Secondly, as we stated earlier, program execution starts with `main()`. We could, for instance write the program this way:

```
/* Here we define the gronk function */

gronk()
{
    printf("GRONK! GRONK! ");
}

/* the order in which functions are defined doesn't
   matter */

main()
{
    printf("I am about to invoke the fabulous gronk
           function!\n");
    gronk();
    gronk();
    printf("Perhaps fabulous overstates the case.\n");
}
```

The program still would start with `main()`, using `gronk()` only when `main()` calls it.

Try to put together a simple program of two or more functions for yourself to make sure you see how to do it.

There is much more to functions than we've seen so far, but functions are not the whole C story. Let's move on to other fascinating facets of the language.

Variables and Declarations

Programs often work with numbers or letters that get stored, altered, and otherwise manipulated. (Such is the fate of computer data!) This is usually done through the use of *variables*. A variable is simply a quantity whose value we can change (or vary). In C, as with most computer languages, a variable is represented by a memory storage location that is assigned a particular name or "identifier" by a program. If we give a variable the name `x`, then the statement

```
x = 22;
```

causes the numerical value of 22 to be stored in the corresponding memory location.

C requires that you "declare" variables. This means that you provide a list of variables that the program uses. The compiler can then scan the list and set up the appropriate storage locations in memory. C variables are "typed". This means we need to state what kind or type each variable is: will it store a number, a letter, or something else? This information is included in the variable declaration. Here is an example illustrating the declaration and use of variables:

```
main()
{
    int x; /* declaring an integer variable */
    char ch; /* declaring a character variable */

    printf("I will now reveal the sum of 2 and 2: ");
    x = 2 + 2;
    printf("%d\n", x); /* form for printing an integer */
    printf("Here is how I grade my performance: ");
    ch = 'A';
    printf("%c\n", ch); /* form for printing a character */
}
```

Notice again how each program statement is terminated with a semicolon. Running the program produces this smug output:

```
I will now reveal the sum of 2 and 2: 4
Here is how I grade my performance: A
```

Now let's look at some of the statements in more detail.

```
int x; /* declaring an integer variable */
char ch; /* declaring a character variable */
```

Each of these two lines is a declaration. The words **int** and **char** are C "keywords". A keyword is a word reserved for use as part of the language. You shouldn't try to use one for another purpose, such as a name for a variable or a function. These particular two keywords represent two of C's variable types. The **int** keyword is used to identify variables that hold integer (whole number) values, values such as 3, 5, -5, and 127. The **char** keyword indicates variables that hold character values. A character can be a letter, a digit, some other typographical character, such as **&**, **#**, or **]**, or a nonprinting character, such as the space character or the newline character. Appendix A contains a complete list of C keywords.

A simple C declaration consists of a type identification followed by a list of one or more variable names, using commas to separate the variable names from each other. For example, if we needed two integer variables, we could use this declaration:

```
int x,y;
```

Or we could declare each variable separately:

```
int x;
int y;
```

In either case, we inform the compiler that we want it to set up two memory storage locations, each suitable for storing an integer.

C allows you reasonable latitude in choosing names for a variable. Please see the box on this topic.

C has several other basic types as well as a variety of derived types. A fundamental type not supported by several implementations (including Hippo C, Level 1) is type **float**. Since we won't use float examples, we should say a little about this type for those of you that have implementations that can use it. It is used to store numbers with decimal points. For example, 3.14, 1.333, and 7.0 are floating-point numbers. The decimal point makes 7.0 a floating-point number even though it has an integral value. Floating-point numbers use a storage scheme that is different from that used for integer types; thus a 7 is stored differently from a 7.0. Type declarations let the computer store and interpret floating-point and integer numbers properly.

Names in C

Standard C allows you to use up to 8 characters meaningfully in a name for a variable or function. You can use longer names, but the compiler will only look at the first 8 letters. Thus, the names **longfellow** and **longfellar** would be considered the same, since they have the same initial 8 characters. Many implementations of C, however, allow a greater number of significant characters. Hippo C, for example, allows 16.

The characters at your disposal for name creation are the lowercase letters, the uppercase letters, the digits, and the underscore character (**_**). The only other restriction is that the first character of an identifier cannot be a digit. Here are some valid and invalid examples:

VALID	INVALID
length	2big
my_age	money\$
cow3	cow 3
BigTime	pac-man

Try to choose meaningful names for your functions and variables; that will make your programs easier to read and understand.

Our example program uses a couple of new options for the **printf()** function. Consider the following line, for example:

```
printf("%d\n", x); /* form for printing an integer */
```


The parentheses contain two distinct entries. The second is *x*, the name of the variable whose value we wish to print. The first is `"%d\n"`; this is a "format specifier". It tells in what form the variable is to be printed. This format specifier may look peculiar at first, but it is quite straightforward once you get used to it. The `%d` is code for "print an integer", and the `\n` is, again, the newline character, meaning to start a new line after the number is printed. Similarly, in the statement

```
printf("%c\n", ch); /* form for printing a character */
```

the `%c` is code for "print a character". In the next chapter we will run through these format codes in detail.

Finally, note the following statement:

```
ch = 'A';
```

The point to note here is that the character *A* is enclosed in single quotes. In C, single quotes are used to identify a letter as being a character. If we had omitted them, the compiler would have thought that *A* was the name of a variable — in this case, of one that we forgot to declare.

In standard C, only a *single character* can be assigned to a **char** variable at a time. That is, we can have statements like

```
ch = 'E';
```

or

```
ch = '!';
```

but not the following:

```
ch = 'donut'; /* illegal donut */
```

Series of letters are handled by "character strings", a topic that we will return to more than once. (Some implementations allow a character constant of two or even four characters, but we will ignore those variants.)

Let's continue our quick scan of C and turn to the topic of operators.

Operators

Operators "do things to stuff." More technically, operators operate on operands. For example, consider the statement

```
ch = 'A';
```

Here there is one operator, the *assignment operator*, represented by the = symbol. It has two operands, the variable `ch` and the character `A`. The action or operation performed by the assignment operator is that the value to its right is assigned to the variable on its left.

C has an unusually long list of operators, and we'll look at a few of the more basic ones now.

The Assignment Operator: =

We've just discussed that one. Keep in mind that it works from right to left. That is, the operand on the left must be the name of the variable that receives the value. Thus

```
my_bonus = 2000;
```

is correct (but, unfortunately, a bit of a daydream). On the other hand,

```
2000 = my_bonus;
```

is invalid in syntax as well as in fact.

One tricky feature of the C assignment operator is that it can be used sequentially. For example, if you wish to set the variables **dick**, **jerry**, **jimmie**, and **ronnie** all to 0, you can do this:

```
dick = jerry = jimmie = ronnie = 0;
```

Again, the action is from right to left; first **ronnie** is assigned 0, then **jimmie** is assigned **ronnie**'s value, and so on.

The assignment operator is a "binary" operator; that means it is an operator that takes *two* operands.

The Addition Operator: +

The addition operator is also a binary operator, and it generates the sum of its two operands. Here are two sample examples:

```
x = 2 + 2;      /* add two constants */
y = x + 5;      /* add a variable and a constant */
```

The Subtraction Operator: -

Use the subtraction operator to find the difference between two quantities:

```
x = 23000 - 8000;
y = x - 23;
```

The second operand is subtracted from the first. Since this operator has two operands, it, too, is a binary operator.

The Sign Operator: -

The same symbol that is used for subtraction is also used to indicate or change the algebraic sign of a quantity:

```
x = -23;  
y = -x;
```

Although the sign operator uses the same symbol as does the subtraction operator, the two operators are considered distinct from one another. One difference is that the sign operator is a "unary" operator, meaning that it operates upon a single operand or value.

The Multiplication Operator: *

Use the * symbol to indicate multiplication:

```
income = hours * payrate;  
inches = 12 * feet;
```

This binary operator causes the operand to its left to be multiplied by the operand to its right.

The Division Operator: /

This binary operator causes the operand to its left to be divided by the operand to its right. For example, the statement

```
her_share = 120 / 10;
```

results in the value 12 being assigned to **her_share**.

When integer values are used, the results are truncated, that is, rounded down to the nearest whole number.

INTEGER DIVISION

TRUNCATED RESULT

7/4	1
8/4	2
9/4	2
11/4	2

In C, the same operator is used for both integer and floating-point division. The result depends on the operand types. If both operands are integers, as above, then integer division is performed. If one or both operands are floating-point, then division is floating point. This means, for instance, that $7.0/4$ (floating-point 7 divided by integer 4) is evaluated as 1.75. That is, the division is carried out to as many decimal places as the system allows; the answer is not truncated to the nearest integer.

The Modulus Operator: %

This is another binary operator. Both operands should be integers. The result is the remainder obtained when the first operand is divided by the second:

MODULUS OPERATION

RESULT

7 % 4	3
8 % 4	0
9 % 4	1
11 % 4	3

The modulus operator will not work with floating-point numbers.

Operator Precedence

Often expressions involve more than one operator, and the answer that one gets may depend on the order in which the operations are performed. For example, consider this statement:

```
answer = 6 * 2 + 8;
```

If we first multiply 6 by 2, we get 12; adding the 8 then gives an answer of 20. But if we first add 2 and 8, then multiply by 6, the answer is 60.

Computers are not tolerant of ambiguity, so C has a series of "precedence" rules to determine what gets done first. For familiar situations like the example above, C follows the usual conventions of algebra, namely, multiplication and division are performed before addition and subtraction. Thus 20, and not 60, is the correct answer for the preceding example. Table 2.1 ranks some operators by precedence.

OPERATORS	ASSOCIATIVITY
()	left to right
- (unary)	left to right
* / %	left to right
+ - (subtraction)	left to right
=	right to left

Table 2.1 Operators In Order Of Decreasing Precedence

We'll talk about associativity later. Meanwhile, note that we have added parentheses to our list of operators. We have already used parentheses in conjunction with functions, but *those parentheses* are not operators. The parentheses in Table 2.1, however, are "precedence" operators, and they modify the ordinary precedence of operations. Suppose, for instance, that in our example we wanted the addition to take place before multiplication. Then we could say this:

```
answer = 6 * (2 + 8);
```

Again, C usage is like that of ordinary algebra.

How does a C compiler know whether you are using parentheses for grouping purposes or as part of a function statement? It tells by context. If you write

```
beans = jack ( 2 + 5); /* parentheses indicate a function */
```

the compiler assumes jack is a function name because there is no operator between the name and the first parenthesis. So, unlike algebra, you *must* use the multiplication sign if you want multiplication:

```
beans = jack * ( 2 + 5); /* parentheses for grouping */
```

What if two operators have the same precedence? In some cases it makes no difference:

```
peas = 20 + 10 - 8;
```

You get the same answer whether you add first or subtract first. But in other cases it does make a difference. Consider this statement:

```
figs = 12 / 4 * 3;
```

How do we evaluate it?

```
12 / 4 * 3 -> 3 * 3 -> 9 ?    (division first)
```

or

```
12 / 4 * 3 -> 12 / 12 -> 1 ? (multiplication first)
```

This is where the associativity information in Table 2.1 is used. Division and multiplication associate from left to right, so in this two-operator statement, the left-most operation is performed first. In this case, division is left-most, so the correct answer is 9. If you want the multiplication to take place first, use parentheses to enforce your desires:

```
figs = 12 / (4 * 3);
```

We will see many more C operators as we go along. Appendix B contains a complete listing of C operators and their precedences. Meanwhile, let's take on some terminology.

Expressions and Statements

Expressions are combinations of values, operators, and other expressions. The simplest expression is just a single value, and you can build from there. Here are some expressions:

```
13
figs
figs + 13
(figs + 13) * (peas - 3)
pears = 3 + 2
```

In C every expression has a value. For an expression using the assignment operator, the value of the expression is the same as the value assigned to the variable. Thus, the value of the last expression in the list is 5.

Because every expression has a value, it can be assigned to a variable. Thus, instead of writing

```
x = 3 + 5;
y = 2 * x;
```

we could write this:

```
y = 2 * (x = 3 + 5);
```

Here 3 and 5 are combined to 8, and 8 is assigned to x. Then the entire expression `x = 3 + 5` has the value 8, so 8 is multiplied by 2, and 16 is assigned to y. This may not be the clearest usage, but it is legal. If you delight in producing obscure-looking code, keep this feature in mind; however, we won't use it again.

A *statement* is the primary building block of a function. It represents a complete instruction to the computer, an action for the computer to perform. So far we have seen three kinds of statements.

The first kind of statement we have seen is the function call, like this:

```
hidecursor();
```

Here the computer is instructed to activate a particular function.

The second kind of statement we have seen is the declaration, such as:

```
int legs;
```

It instructs the computer to set aside and label a memory location to hold a variable of the given type.

The third form is the assignment statement, like the following:

```
legs = 2 * noses;
```

Here the computer action is to find a value and assign it to a variable.

We will encounter other forms of statements later.

As you surely have noticed by now, C statements are terminated with a semicolon. (Certain forms may end with a brace; we'll point them out when they show up.) The usage is different from that of Pascal. In Pascal, the semicolon separates one statement from the next and may be omitted after the last statement. In C, the semicolon is *part* of the statement and cannot be omitted.

We have been using one statement per line, but that is not really a C requirement. You may put several statements on one line, or one statement on several lines, for the compiler uses semicolons to tell where one statement ends and another begins. Spaces and newlines are ignored except as separators. Thus, the following is legal (but ugly) C:

```

main(
){ int legs, noses;  noses  = 5; legs =
  2
  *
  noses;
  printf("%d\n",
         legs); }

```

About the only troublesome practice would be to break a line between an opening and closing double quote in a `printf()` statement. We'll return to that matter in the next chapter.

This flexible formatting is called "free formatting". You should regard it as an opportunity to format your programs for maximum clarity, not as license to see what you can get away with.

Now let's go back to functions.

More on Functions: Arguments and Return Values

Breaking a program into separate functions is a great technique for promoting modular, readable, serviceable programs. But it does create new problems. For example, often one function needs information from another function; we need communication *between* functions. Sometimes the calling function needs to pass information to the called function, and sometimes the called function needs to pass information back. In C, we can use *arguments* and *return* values to meet these needs.

Arguments

When we call a function, we can pass information to it by placing the information within the function parentheses. An item so passed is called an argument. For example, in the function call

```
printf("La di da!\n");
```

the argument is "La di da!\n". The `printf()` function prints what it is told, but the program has to tell it exactly what is to be printed.

Some functions require more than one argument. In this case, we can use an *argument list*, which is a series of arguments separated by commas. For example, the function call

```
printf("%d", legs);
```

has an argument list of two arguments. The first is "%d" and the second is legs. Incidentally, printf() is a bit unusual in that the number of arguments it takes is not fixed at one value. Typically, a function will take only one argument, or only two, but not both.

For a more typical usage of arguments, let's look at a couple of examples from the Macintosh library of functions.

A Graphics Example

We'll look at an example using two functions from the Macintosh Quickdraw package. This package contains those functions used to control the Macintosh screen, so it is a good package to know. The Macpaint program, for example, is based on the Quickdraw package.

Using Quickdraw is one situation in which compiler differences appear. According to *Inside Macintosh*, the official Apple description of Toolbox functions, the `initgraf()` function must be called before using Quickdraw. In Hippo C, Level 1, this is done automatically, so you can go ahead and use the Quickdraw functions without further preparation. In fact, you'll get in trouble if you do try to call `initgraf()`. Other compilers may require you to call `initgraf()` explicitly or, may make calling it a compiler-option choice. If you are using a different compiler, try checking its documentation to see what to do. Or look for an example using Quickdraw. Does it use `initgraf()`? If so, check how its argument is declared and what `#include` files are used. If you can't find an answer, try running the programs as shown in this book. If that doesn't work, write or call your implementer for further instructions.

The use of `initgraf()` is the most important compiler-dependent matter you will face in this book. Clear up that problem, if it is one, and the rest of the book should be clear sailing.

The two Quickdraw functions we'll look at are `moveto()`, which moves the pen to a specified location, and `lineto()`, which draws a line from the current pen position to an indicated position.

From Mac's Toolbox: New Routines

LineTo
MoveTo

Draws a line to the indicated point
Moves pen to point without drawing

Here is the sample program:

```
main()
{
    int horiz, vert;

    horiz = 50;
    vert = 50;
    moveto(horiz,vert);    /* move to start of box */
    lineto( 5 * horiz, vert);    /* draw top side */
    lineto( 5 * horiz, 3 * vert); /* right side */
    lineto( horiz, 3 * vert);    /* bottom */
    lineto ( 50, 50 );          /* left side */
}
```

First, understand that on a Macintosh screen position is measured from the upper left-hand corner of the window. The full screen measures 512 units horizontally by 342 units vertically. Figure 2.1 shows the output of this program. You may have to use the mouse to move and enlarge the Hippo Command Window first, so that the box will fit.

The first function call is this:

```
moveto(horiz,vert);
```

The first argument (**horiz**) to **moveto()** indicates the horizontal coordinate for locating the pen, and the second argument (**vert**) indicates the vertical position. In this case, both have the numerical value of 50, so the pen is placed 50 units down and 50 units to the right of the upper left-hand corner of the window.

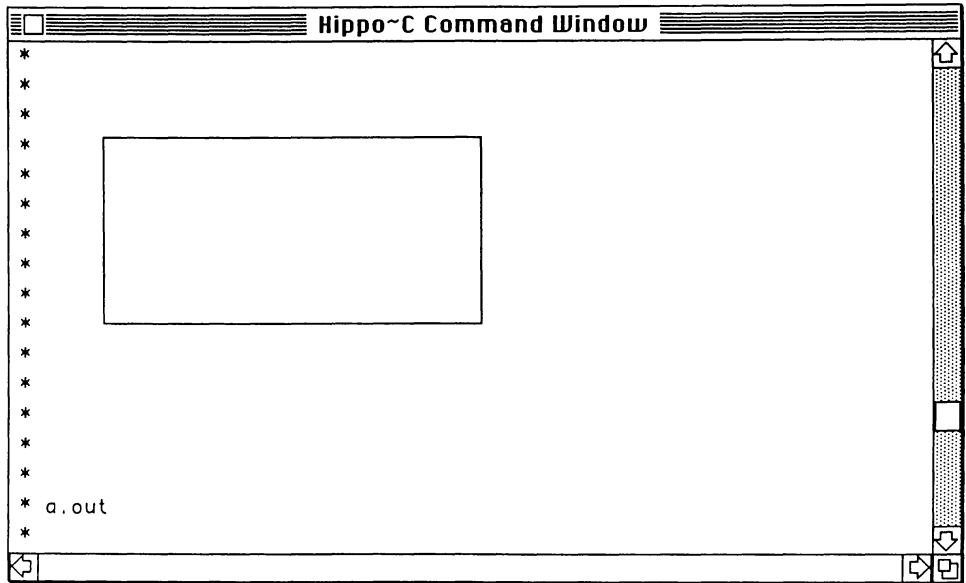


Figure 2.1 Drawing a Box

The second function call is this:

```
lineto( 5 * horiz,vert);
```

This time the pen draws a line as it is moved from its original position (50,50) to its new position (250,50). In this case, the expression **5 * horiz** is evaluated to 250, and then that value is transmitted as an argument.

The remaining function calls complete the job of drawing a rectangle. In each case a line is drawn from the current position to the new position described by the function arguments.

A very important point to note is that the arguments to a C function can be variables, as in the first call, constants, as in the last call, or any other expression. In each case the *value* of the variable, constant, or other expression is evaluated and passed on to the function. This is called "passing by value". For those familiar with Pascal, C arguments correspond to Pascal value parameters rather than to Pascal variable parameters.

What if we want to write a function that uses arguments? We'll save that topic for a later chapter. For the present, we will content ourselves with *using* functions with arguments. However, to satisfy at least some of your curiosity, we can note a couple of points now. First, when defining a function that uses arguments, we need to include declarations for new variables that receive the passed values. Second, functions expect passed arguments to correspond in number and in type to arguments described in the function definition. For example, both `moveto()` and `lineto()` expect two integer arguments. Passing the wrong number of arguments or arguments of the wrong type can lead to program failure.

The Function Return Value

Now let's see how functions can communicate values back to the calling program. A C function has the potential to "return" a single value to a program. For example, recall that we mentioned the `getchar()` function, which gets the next keyboard input character. To find out what the character is, we can use a statement like this:

```
ch = getchar();
```

We would say that **ch** is assigned the value returned by the `getchar()` function.

Here is a primitive example:

```
main()
{
    char ch;

    printf("Type a character\n");
    ch = getchar();          /* returned value is assigned
to ch */                   ↩
    printf("\nThat was a %c\n", ch);
}
```

A run looks like this:

```
Type a character
Y
That was a Y
```

In this case, `getchar()` performed an action (finding the next input character) and returned a value (the value of that character). We used a fancier form of `printf()`, which we will explain in detail in Chapter 3. Basically, the `%c` acts as a *place holder* showing where and in what form `ch` will be printed.

The Hippo C version of `getchar()` has its "echo" input. That means the characters appear on the screen as you strike the keys. For instance, in the last example, the `Y` appears on the screen. Some implementations use a non-echoing `getchar()`; the character entered from the keyboard is processed by the program, but not shown on the screen. In Hippo C the echo feature can be turned off by using the function call `echo_off()`; the call `echo_on()` will restore echoing.

Setting up a function to return a value is simple enough; we just use the keyword **return** in the function definition. Here is a short example illustrating the technique:

```
main()
{
    int value;

    value = always2();
    printf("%d\n", value);
    printf("%d\n", always2());
}

/* simple function that always returns 2 */
always2()
{
    printf("Processing...\n");
    return 2;
    printf("This is a feeble function.\n");
}
```

The output looks like this:

```
Processing...
2
Processing...
2
```

There are a few points to note. First, when a function reaches a return statement, it returns whatever value follows the keyword return. This can be a constant, a variable, or any other form of expression. The expression is evaluated, and the value is returned to the calling program.

Second, when a function executes a return statement, it quits, and control returns to the calling program. Thus, the final printf() statement in always2() is never executed.

Third, upon return to the calling program, the return value can be used the same way as any other value. Thus, in this program, one time we assign the return value to a variable, and one time we use it as an argument to printf(). We also could have used it as part of an expression, as in

```
y = 3 * (4 + always2() );
```

You can think of the function call as being replaced by the corresponding return value.

The Function as a Black Box

Once you learn a function's argument list, its action, and its return value, you know all that you need to know about that function. Sometimes this is referred to as the "black box" view of a function. The term is borrowed from electronics, where a black box would be characterized by its electronic input and output characteristics, with the interior of the box hidden away. Similarly, with a function, we know what goes in, we know what comes out, but we can be entirely ignorant of the internal programming.

The black box view is more than an analogy, it is a design goal. The idea is that if a function's only interactions with another function are through its input (argument list) and output (return value), then the function won't produce some unexpected side effect on other parts of the program.

Summary

C is a modular language. C programmers break down a complex problem into simpler tasks and then write separate modules called functions to handle these tasks. There is a large library of C functions available on most C implementations, and this library is currently being standardized by a standards committee. Macintosh C, however, goes far beyond standard C, offering hundreds of functions to access the Macintosh environment and facilities. Thus, using functions is the key to using C on a Macintosh.

A C program consists of one or more functions. One function must be called **main()**, and this is the first function executed in any program. It can then call, or invoke, other functions.

The body of each function consists of a series of statements. Declaration statements announce the names and types of the variables to be used in a function. Common types are integer (**int**) and character (**char**). Assignment statements assign values to variables. Function call statements activate functions.

Information is communicated to a function through an argument list, which is a list of values to be used by the function. In turn, a function can use a **return** statement to provide a value for the calling program.

To be able to use a function properly, you should know three things about it: the argument list it expects, the action(s) it performs, and the value, if any, it returns.

3

I/O Functions and Types

In this chapter you will learn about:

- Input and Output (I/O)
 - Character I/O
 - Formatted I/O
 - Fundamental C types
 - The `#include` and `#define` preprocessor directives
 - Using a simple loop
 - Using functions
-

A computer is well-suited for making rapid, repetitive calculations, but its innate ability to communicate with humans is much less spectacular. Often the longest, most involved part of a program turns out to be the handling of the computer-user interface. Sometimes, for example, you may wish to instruct a program or provide it information as it is running. In C this involves using some sort of "input" function, that is, a function that gathers information, or input, while the program runs. In turn, a program can communicate back to you by using "output" functions. Input and output functions are referred to collectively as I/O functions, and they are one of the main topics of this chapter.

Traditional I/O functions do such things as read input from the keyboard and print characters on the screen. Much of the special flavor of the Macintosh, however, stems from its innovative I/O forms, such as the mouse, icons, and windows. We must type before we can mouse, however, so we will concentrate on traditional forms in this chapter.

Standard C has a wide variety of I/O functions. Here we will look at four of the most heavily used examples: `getchar()`, `putchar()`, `printf()`, and `scanf()`. We will also look at some related I/O functions from the Macintosh Toolbox. En route, we will take a first look at the C `if...else` and `while` statements.

The second major topic for this chapter is the C scheme of data types. I/O functions are either tied to particular data types or else need to specify the type of data being used. Hence the topic of data types ties in to the topic of I/O.

We will be looking at specific functions, but much of what we do will be good practice for functions in general. As we go along, note how function arguments and return values are used.

Character I/O

Many C programs are built around processing input a character at a time. C has two functions explicitly designed for character processing: `getchar()` and `putchar()`.

Character Input: `getchar()`

You met `getchar()` in Chapter 2, but let's review it now. We can describe a function by three things: its argument list, the action it performs, and its return value. For `getchar()`, these are rather simple. It has no arguments, it obtains the next input character, and it returns the value of that character.

There are a couple of points that need elaboration, however. One is the question of where the input comes from. C programs take input from something called the "standard input." On most systems, including the Macintosh, the standard input is the keyboard by default. ("By default" means that this particular choice is made automatically unless you override it.) However, C implementations typically allow you to change the standard input from the keyboard to, say, a file, at the time a program is run, causing the program to take input from the file instead of the keyboard. Macintosh applications usually do not make use of this "redirection" feature, so we won't discuss it further.

The second point to note is *when* keyboard input is read. Many systems use a "buffered" system in which the keystrokes are stored in an intermediate memory area called a "buffer." The contents of the buffer get sent to the program when the buffer gets full or when the user strikes the [RETURN] key. (The [RETURN] key itself generates the newline character, which serves as a sign to empty the buffer.) Hippo C on the Macintosh, however, uses an unbuffered system. As soon as a key is

struck, the character is transmitted to the program; it is not necessary to strike the [RETURN] key for that purpose.

Here is a short aptitude test using `getchar()`. This program introduces the `if...else` statement. We'll explain it briefly after the program, and more fully in Chapter 4.

```
main()
{
    char ch;

    printf("Enter the letter 's'\n");
    ch = getchar();
    if ( ch == 's' )
        printf("\nYou are hired!\n");
        /* do if ch is 's' */
    else
        /* do if ch is not 's' */
        printf("\nUnfortunately, we have many fine
applicants...\n");
}
```

Here is a sample run, one made by a talented applicant:

```
Enter the letter 's'
s
You are hired!
```

The `if...else` statement makes a test and performs one action if the test is true and an alternative action if it is false. Here the test is to see if the input character (the variable `ch`) is the same as the desired character (the letter `s`). Note that C uses a double equals sign (`==`) for comparing two items for equality. The indentation used with the `if` statement, although not obligatory, is good programming style.

You should also note the use of newlines. The `\n` at the end of the first `printf()` statement moves the cursor down to the next line, so that when the testee types an `s`, it appears on the next line and not at the end of the first line. The `\n` at the beginning of the other `printf()` statements causes those lines to be printed on a line following the user's answer. Otherwise the printing would begin immediately to the right of the response. This initial newline would not be needed on a buffered system, for there you have to

strike the [RETURN] key after your answer, and that key starts a newline anyway.

Character Output: putchar()

The `putchar()` function takes one argument, a character value. Its action is to print the character on the screen. More exactly, it sends the character to the "standard output." For the Macintosh, the standard output is the screen. Most C systems offer a redirection system that lets you replace the screen with a file at run time for catching the output, but we will ignore that feature. In most implementations (but not Hippo C) `putchar()` also has a return value. In that case, the return value is just the printed character, or, if there is an I/O error, the return value is a -1. Often, the return value is not used, but a careful program would use it to check for I/O errors. Here is an example in which the program repeats an input character.

```
main()
{
    char ch;

    ch = getchar();
    putchar(ch);
}
```

A sample run could look like this:

HH

The user typed the first H, and the program produced the second one. With a C implementation using buffered input, the program run would look like this:

```
H[RETURN]
H
```

In this case, you need to strike the [RETURN] key to transmit the character, and that advances the screen to the next line for the output.

Actually, the variable `ch` is expendable in this program. The following version works just as well:

```
main()
{
    putchar( getchar () );
}
```

In this version, the return value for `getchar()` is used directly as the argument for `putchar()`. This illustrates again the concept of passing arguments by value. The actual argument in this case is not the *function* `getchar()` itself; it is the *return value* of the function.

One more point to note is that output for Hippo C *is* buffered. That means the output is directed to an intermediate memory location called a buffer. The contents of the buffer, in turn, are sent to the screen when the buffer is "flushed." This flushing occurs whenever 1) the buffer fills (80 characters for Hippo C), 2) the program ends, 3) a newline is printed, or 4) a special flushing function is called. This function is called, fittingly enough, `fflush()` and is mentioned in Chapter 10.

A Brief Whirl Through the while Loop

It is boring to work with a program that just reads or writes a single character. The `getchar()` and `putchar()` functions are useful because you can write programs that use them repeatedly to process lots of characters. One method is to use the while loop. We'll study loops more fully in the next chapter, but we'll look at the basics now.

A loop is a section of program that can be cycled through repeatedly. One way to create a loop in C is to use the while structure. Here is an example of a loop:

```
while ( ch == 's') /* keyword "while" marks start of
                    loop */
    putchar(ch);
```

As with the `if` statement, there is a test enclosed in parentheses followed by a statement. In the while, the following statement is performed if the test is true. Then the test is made again; if it is still true, the statement

is executed again. Then the test is made again, and so on. The program cycles or "loops" through the while statement over and over again until the test fails.

If, in our example, `ch` did equal `'s'`, then this fragment would repeat indefinitely, printing `s` after `s`, for nothing in this fragment changes the value of `ch`. Loops that don't quit are called infinite loops; usually they are not considered desirable. Normally, however, there is some action within the loop that has the potential to change the result of the test. Consider, for instance, this program fragment:

```
ch = getchar();          /* get a character */
while ( ch == s)         /* check if character is
                           an 's' */
{
    putchar(ch);          /* print character */
    ch = getchar();       /* get a new character */
}
```

Here the brackets indicate a "compound" statement. They indicate the extent of the loop. Without the brackets, only the first statement following the while test would be part of the loop. With the brackets, everything between the brackets is included in the loop. The brackets play much the same role as **begin** and **end** in Pascal.

And what does this fragment do? The user enters a character. If it is an `s`, the program echoes it and prints the next character. If *that* character is an `s`, it prints it and reads the next character. This goes on until a non-`s` is entered, then the fragment is finished. The output could look like this:

```
ssshsss
```

Each of three input `s`'s was repeated, and the loop stopped when an `h` was typed.

Note that the printing of the three `s`'s was delayed until the `h` was typed. This is an example of the output buffering we mentioned earlier. Here the output (`sss`) was saved until the program terminated.

The key point in this example is that the while loop lets a program process characters until some sign to quit pops up. The fact that a new value

for `ch` is obtained each cycle of the loop provides an opportunity to change the result of the loop test.

Here is a program that uses a loop to count the number of characters entered on a line:

```
main()
{
    char ch;
    int count;

    count = 0;           /* initialize count to 0 */
    ch = getchar();      /* read first character */
    while ( ch != '\n')  /* continue until end of line */
    {
        count = count + 1; /* increase count by one */
        ch = getchar();    /* get next character */
    }                    /* end of loop */
    printf ("There were %d characters.\n", count);
                        /* report results */
}
```

Here is a sample run:

```
I'll have some ultra chocolate.
There were 31 characters.
```

Note that spaces and punctuation count as characters.

Recall that hitting the [RETURN] key generates a newline character, symbolically represented by `\n` in C. This program, after initializing the count variable to 0, fetches a character. If it is not the newline character, the program adds one to the count and gets the next character. (In C, the symbolism `!=` means "not equal to.") This continues to the end of the input line, indicated by a newline character, then the program reports the count.

The indentation of the while loop is not required, but it is highly recommended in order to make the program more readable. The indentation shows at a glance the extent of the loop section of the program.

The `printf()` statement is slightly more involved than the ones we have seen so far, and we will explore the matter later this chapter.

Although this program is valid C, it is not stylish C. A more experienced programmer might rewrite the program this way:

```
main()
{
    char ch;
    int count = 0;

    while ( (ch = getchar() ) != '\n')
        count++;
    printf ("There were %d characters.\n", count);
}
```

This may look weird, but all we have done is combine a few steps together and bring in a new operator. First, we have the following modified declaration statement:

```
int count = 0;
```

This declares the int-type variable count and also sets it to zero. Thus, when the computer sets aside space for the count variable, it also places a value in it. We say that the variable count was "initialized."

The next modification is more complex:

```
while ( (ch = getchar() ) != '\n')
```

The outer pair of parentheses mark the extent of the while loop test condition. Thus the test condition itself is this:

```
(ch = getchar() ) != '\n'
```

This looks odd, but keep in mind that parentheses can be used to indicate the precedence of operations. Hence the first thing that gets done is what is within the inner group of parentheses:

```
ch = getchar()
```

That is, a character is fetched and assigned to the variable `ch`. Now, `(ch = getchar())` as a whole is an assignment expression and has the same value as `ch`. This means that the next level of parentheses boils down to checking to see if `ch` is a newline:

```
ch != '\n'
```

In brief, the expression

```
(ch = getchar() ) != '\n'
```

is short for

```
ch = getchar();  
ch != '\n'
```

Because the short form is a *single* expression, it can be used as a test for the while loop, whereas the two-expression version cannot. You should become comfortable with this sort of expression (combined assignment and comparison), for it is a very common idiom in C.

Finally, we have this as-yet mysterious statement:

```
count++;
```

The `++` is the C "increment" operator, and it increases the value of its operand (here `count`) by one. So it is a shorter form of

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

The increment operator version is, however, easier to type, and it is usually implemented more efficiently on the computer. We'll come back to increment and decrement operators in Chapter 4.

Here is a slight variation of the example. Instead of counting each character on a line, it reprints each character:

```
main()
{
    char ch;

    while( (ch = getchar() ) != '\n')
        putchar(ch);
}
```

Again each character in a line is read and printed until the newline character produced by the [RETURN] key shows up. Here is a sample run:

```
Don't repeat this![RETURN]
Don't repeat this!
```

Again, the output of `putchar()` was collected in a buffer, then sent to the screen when the program ended. If we had typed more than 80 characters before hitting the return key, the buffer would have printed out the first 80 as soon as that block had been typed, since the buffer would be filled at that point.

The Character Type

Because we have been dealing with character I/O, now is a good time to look further at type `char`. In particular, we will investigate how characters are stored on a computer and look at various ways to represent characters in C.

Character Storage

Everything in computer memory, be it text, numerical data, or program code, is stored in what we can think of as an electronic pattern of 0s and 1s. The smallest unit of computer memory is called a "bit," and it can hold a single 0 or 1. Using a single bit is much like counting with one finger. Fortunately, computers have many fingers. The next largest unit of computer memory is called a "byte," and it consists of 8 bits. Thus it can hold any combination of eight 0s and 1s; there are 256 possible

combinations. The next largest unit of memory is called a "word," and it is the natural storage unit for a given machine. The size of the word depends on the hardware. For example, the original Apple had an 8-bit word (one byte), which is why it is termed an 8-bit machine. The IBM PC uses a 16-bit word, and the Macintosh has 32 bits at its disposal.

How does this relate to storing characters? Clearly, since a computer stores everything as a pattern of 0s and 1s, it needs to use a code of 0s and 1s to represent characters. The most widespread code, and the one used on the Macintosh, is the ASCII code shown in Appendix E. In this code, for instance, 01000001 represents the character 'A', 00101011 represents '+', and so on. Altogether, the code represents 128 different characters, including several nonprinting characters used to control printers and the like. This implies that a single byte of memory (which can hold 256 different combinations) is large enough to store any character code. C utilizes this fact, so that when we declare a variable to be type `char`, just one byte of memory is allotted for its storage. (Some languages simply use the word size, but this would be wasteful on a Macintosh, for the Macintosh word is large enough to hold 4 characters.)

Let's take a closer look at the code. The representation 01000001 is an example of a "binary number," a number written using just 1s and 0s. Appendix C discusses binary numbers further, but what we should note here is that a binary number can also be expressed as an ordinary decimal number. The 01000001 binary number, for example, is equivalent to the decimal number 65. Thus, we can also speak of the character 'A' as being represented in ASCII code by the number 65, understanding that in actual storage the 65 is in binary form. Appendix E shows not only the binary code, but also the decimal code, octal code (base 8), and hexadecimal code for the ASCII character set. If you are not familiar with these other number forms, you may wish to read the section of Appendix C that describes them.

The ASCII code encompasses the numerical values 0 through 127. The Macintosh modifies it in two ways. First, it extends the code by using the numbers 128 through 217 to represent some additional characters, such as accented vowels and Greek letters. (The list of extra characters may be expanded even more.) Second, several of the "control characters" (codes 0 through 31) are printed using the "missing character" symbol (a vertical, rounded rectangle in the regular font) when sent to the screen. These control characters (so called because they were generated using the [CONTROL] key found on many keyboards) normally are used to do things like ring bells, cause formfeeds to a printer, and the like. The Imagewriter, however, does acknowledge several of these characters.

Character Constants

As we have seen, we declare a character variable with a statement like this:

```
char ch;
```

This sets aside one byte of storage for the variable `ch`. This variable may acquire a value from the keyboard via an input function, or we may assign it a value within a program:

```
ch = 'D';
```

In this example, `'D'` is a "character constant," just as, say, `7` is an integer constant. The usual way to represent a character constant in C is to enclose the character in single quotes, as we did above. The quotes inform the compiler that the `D` is a character constant and not, say, a variable *named* `D`.

Special Character Constants. Some characters are not easily typed or else have unwanted side effects, so C has special mechanisms for representing special characters. One is to use the ASCII code more or less directly. For example, the character whose ASCII code is 14 is the formfeed character. It causes the printer to advance one page. The Macintosh terminal doesn't respond except for printing the missing character symbol. If we had a type `char` variable named `down`, we can assign it this character with this statement:

```
down = '\014';
```

The single quotes alert the compiler that we have a character constant. The backslash (`\`), which slants opposite the ordinary slash (`/`), indicates that we are using a representation of the character rather than the character itself. Finally, the `14` is the ASCII code, *in octal*, for the character.

This form can be used for any ASCII character, not just the unprintable ones. For example, the ASCII code for `'A'` is 65 decimal, or 101 in octal. Thus, we can represent the letter `'A'` with the notation `'\101'`, if we like.

Certain special characters have, in addition to the octal representations, their own individual representations, sometimes termed "escape sequences." Here is a list of these representations, along with the ASCII representation:

<code>\n</code>	newline	<code>\012</code>
<code>\t</code>	tab	<code>\011</code>
<code>\b</code>	backspace	<code>\010</code>
<code>\r</code>	carriage return	<code>\015</code>
<code>\f</code>	form feed	<code>\014</code>
<code>\\</code>	backslash	<code>\134</code>
<code>\'</code>	single quote	<code>\054</code>
<code>\"</code>	double quote	<code>\042</code>

The tab character makes the cursor or printer move to the next tab position. In the Hippo C environment, tab marks are set every 4 character widths; the first tab moves you to column 5, the second to column 9, and so on.

The Mac ignores the backspace and formfeed characters when sent to the screen, but the printer does recognize them.

The carriage return defers from the newline in that the carriage return moves the cursor to the beginning of the same line, while the newline moves the cursor to the beginning of the next line.

These escape sequences can be used in the same fashion as regular characters. For example, if you wish to set a character variable to be the newline character, you could say

```
ch = '\n';
```

or

```
ch = '\012';
```

The first form is more easily remembered and recognized. It is also more portable, since it would still be true on machines that do not use the ASCII code.

A special notation is needed for the single quote because enclosing a regular single quote in single quotes (') would look confusing to the compiler, while the notation \" poses no problem. Similarly, the special notation for the double quote will be needed for character strings, which use ordinary double quotes to mark their beginnings and ends.

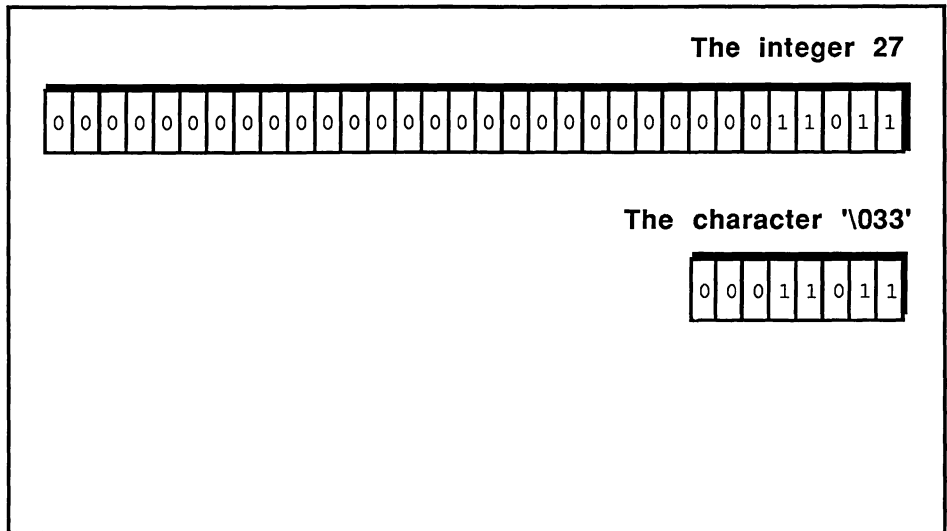


Figure 3.1 27 vs '\033'

Characters and Integers

The Imagewriter uses the [ESCAPE] character to initiate many special instructions, such as those that set the type size, underlining, and boldface. The Macintosh keyboard lacks this key, but C programs can use the ASCII code representation '\033' for the character. The corresponding decimal code is 27; can we then do something like this?

```
char esc;  
esc = 27;
```

Yes, we can! So why bother with the more cumbersome form '\033'? The answer has to do with storage. The '\033' form is a *character* constant, meaning that one byte of memory is used to store it. The 27, however, is an *integer* constant, and in Hippo C on the Macintosh, an integer is stored in

a word, or four bytes. Thus the 27 uses up four times as much storage as '\033', with three of the bytes being all 0s. Figure 3.1 shows this. Also, because `c` is a `char` type and 27 is an integer type, the assignment process involves "type conversion," in which one type is converted to the other. This is another topic we will return to later.

End-of-File

We've used the idiom

```
while ( (ch = getchar() ) != '\n' )
```

to read input to the end of a line, using the fact that the newline character marks the end of line. Many programs, however, are intended to read a file of text input. Is there a special character in C that marks the end of the file?

The answer is, sort of. C uses the symbolism EOF to indicate End-Of-File, and we can use it just as we used '\n'. There is an important conceptual difference we will discuss soon, but first, here is an example that counts the number of characters in a file:

```
#include "stdio.h" /* information used by our program */
main()
{
    char ch;          /* bad C -- see text */
    int ct = 0;

    while ( (ch = getchar() ) != EOF )
        ct++;

    printf("\nNumber of characters = %d\n", ct);
}
```

Before looking at some of the details, let's see what this program does. It reads input a character at a time, incrementing the count, until EOF shows up. Then the program reports the count and ends. And when does EOF show up? If the program is reading a file (which we haven't yet shown how to do), EOF shows up, naturally, at the end of the file, after all the characters have been read. If the program is reading input from the keyboard, we have to use a special keystroke to indicate we are finished supplying input. With Hippo C on the Macintosh, the terminating

keystroke is [OPTION]-d, that is, holding the [OPTION] key down while striking the [d] key. Thus, a run could look like this:

```
I love
to
input.[OPTION]-d
Number of characters = 16
```

Note that the two newline characters (after **love** and **to**) are included in the count, as are the spaces and punctuation.

Now, let's talk EOF. EOF is *not* a special character used to mark the end of a file. Instead, it is a *signal* returned by `getchar()` when it reaches the end of the file. How does `getchar()` know? That depends on the system. Some systems do have a special character or combination of characters to mark the end of a file. Other systems maintain a file record recording the size of the file; when the number of characters read reaches the size of the file, that's the end.

Regardless of the means by which it detects the end of file, `getchar()` then returns the value EOF. With Hippo C and most other C implementations, EOF has the numerical value -1. Why -1? Because this value does not correspond to any ASCII character. Thus no character in the input can accidentally trigger the end-of-file condition.

The value of -1 does pose a problem in many C implementations with the assignment `ch = getchar()`, where `ch` is type `char`. The problem is that often the `char` type is set up to hold the values 0 to 255, so that -1 is an impossible value for `ch`. The solution is to declare `ch` to be type `int`. This entails more storage space, but makes -1 a permissible value. C makes the necessary type conversions to allow an `int ch` to work as well as a `char ch`.

The Hippo C version works with `char ch` because the Hippo C `char` type is set up to hold the values -128 to 127; thus -1 is a possible value for `ch`. However, it is better to use `int` anyway in programs that check for EOF, in case you ever move your program to another system.

One new feature in this program is the opening line:

```
#include "stdio.h"
```

This is an example of a C "preprocessor" directive. Having used it, we'd better discuss it.

The C Preprocessor — A First Look

The C preprocessor is a program that processes your C program before actual compilation takes place. It is invoked automatically when you compile a program. The intent of the preprocessor is to make life simpler for you. We'll look at its two most important capabilities, the **#define** directive and the **#include** directive.

The #define Directive

Instructions to the preprocessor are identified by an initial **#** symbol. Also, a directive should begin at the beginning of a line. The **#define** directive, as you might expect, lets you set up definitions. Often it is used to set up symbolic definitions for constants. Here are some example directives:

```
#define NO    0
#define STOP 'q'
#define MESSAGE "\nVery good! Do it again!\n"
```

In each case, the first symbolism following the **#define** becomes a symbolic representation of the rest of the line. Here we have defined **NO** to represent the integer 0, **STOP** to represent the character 'q', and **MESSAGE** to represent a character string. Using capital letters for the names is not required, but it is a common convention. It lets you spot definitions at a glance when they are used in a program. You can use your definitions when writing a program; and when you compile the program, the preprocessor replaces your definitions with the corresponding values. Here is an example:

```
#define STOP 'q'
#define MESSAGE "\nVery good! Do it again!\n"
main()
{
    char response;
```

```

    printf("Enter a character other than %d:\n", STOP);
    while ( (response = getchar() ) != STOP )
        printf(MESSAGE);
    printf("\nWasn't that great fun!\n");
}

```

Every time STOP appears in the program, the preprocessor replaces it with an 'q'.

Here is a sample run:

```

Enter a character other than q:
w
Very good! Do it again!
h
Very good! Do it again!
y
Very good! Do it again!
q
Wasn't that great fun!

```

What's the point to the #define directive? There are several. First, using a defined constant can make the meaning of a particular constant clearer. In our example, using the name STOP indicates the role played by the letter 'q'. Such definitions help document a program, particularly if we include a comment. We could, for instance do something like this:

```

#define DAY_TO_SEC  86400  /* seconds in a day */

```

Second, using a series of define directives lets us gather in one place the constants used by a program. This, too, helps document the program.

Third, a defined constant makes it easier to modify the program. If, for instance, we wished to change the loop terminator from 'q' to 's', we would just change the define directive, leaving the body of the program unchanged. This is particularly useful if the constant is used in several different places in a program.

Fourth, a defined constant makes a program more portable. EOF is an example of a defined constant. On most systems it has a value of -1, but on

some systems it has a different value. We merely use EOF in our programs, and let the definition take care of the differences between systems.

But where is EOF defined? Its definition is part of a special file named `stdio.h`. The name stands for "standard input/output header." A header is something that goes at the top of something else, and that is what is accomplished by the `#include "stdio.h"` directive. Let's look at it next.

The #include Directive

The `#include` directive instructs the compiler to include the named file at the beginning of the program. Thus, the line

```
#include "stdio.h"
```

produces the same effect as physically typing out the entire contents of the `stdio.h` file at that location. You can use `<stdio.h>` instead of `"stdio.h"` if you prefer. This particular file is part of the standard C package. The actual contents vary from implementation to implementation, but in general they concern input/output matters. In particular, the file generally includes the definition for EOF. Thus, if you look in the Hippo C version of `stdio.h`, you will find this line:

```
#define EOF (-1)
```

You will also find several other define directives concerning I/O and file operations. Many C implementations also include definitions of `getchar()` and `putchar()` in `stdio.h`; these versions of C require that you include the `stdio.h` file in any program using those functions. We will include `stdio.h` only when Hippo C requires its use.

The include directive is not confined to system files. You can create your own files of definitions and use them in the same fashion. Just place the filename in double quotes or in angle brackets. It is not necessary that the filename end in `.h`, but that is the usual practice. It tells you the nature of the file.

The preprocessor has other abilities, and we will unveil them as they are needed.

Other C Types

We have discussed character I/O, but other types of I/O are possible. We may wish to read or write numerical data or character strings, for example. Before discussing these forms of I/O, however, we need to discuss the corresponding types.

Integer Types

An integer, as we have seen, is a whole number, one with no fractional part or decimal point. The basic C type for integer values is `int`, but C offers several variations on the integer theme. There are two facets that are varied: the memory size used to store the integer and whether or not negative numbers are permitted.

Integer Sizes. The basic integer type is declared using the keyword `int`. The amount of storage used for `int` depends upon the computer and upon the discretion of the implementer. Hippo C uses 32 bits, meaning a pattern of 32 0s and 1s is used. This makes for 4,294,967,296 possible combinations! The basic `int` type is "signed," meaning that both positive and negative numbers are allowed. In Hippo C, the `int` type is set up so that the possible range of values is -2,147,483,648 to +2,147,483,647. Several Macintosh implementations, on the other hand, use a 16-bit `int`, making the range -32,768 to +32,767.

Does your program use integers in the billions? If it only uses integers in the hundreds or thousands, it is wasteful to set aside such large storage spaces for integers. C allows you to modify the basic storage size by using the keyword `short` to modify a declaration. In Hippo C, a `short int` uses just 16 bytes, providing a range of -32,768 to +32,767. You can declare a variable to be this type by either of the following methods:

```
short int estines;    /* full form */
short fingers;        /* shortened form */
```

On Macintosh implementations using a 16-bit `int`, `short` is the same as `int`.

If you deal with really small numbers, less than 128, you can use type `char` to store them, for `char` uses just one byte.

C also has a keyword **long**. It, too, can be used as an adjective or as a stand-alone word in declaring variables:

```
long int repid;    /* full form */
long faces;       /* shortened form */
```

For Hippo C, the **long** type is the same as regular **int**, the idea being that **int** is big enough anyway. Macintosh implementations using a 16-bit **int**, however, use a 32-bit **long**.

C guarantees that **short** will be no longer than **int** and that **int** will be no longer than **long**. Small implementations may make all three the same size, but more typically two are the same, and one different, with either **long** bigger than the other two types or with **short** smaller than the others.

Unsigned Integers. The three classes of integer we have discussed are all signed integers. By using the keyword **unsigned**, we can create types that only hold nonnegative numbers. For example, in Hippo C, the type **unsigned int** covers the range 0 to 4,294,967,295. Similarly, **unsigned short** covers the range 0 to 65,535. Using **unsigned** alone, as in

```
unsigned stars;
```

is interpreted to mean **unsigned int**.

Unsigned types are useful for quantities that are intrinsically nonnegative, such as computer memory addresses, and for values a bit too large for a signed integer, such as 3,155,692,597 — the number of seconds in a century.

Table 3.1 summarizes these integer types for Hippo C on the Macintosh. We include **char** as an integer type, for the integers stored in it can be interpreted as numbers rather than as ASCII code. And if you think this profusion of types is overwhelming, keep in mind that you usually can get by just using type **int**. The other varieties, for the most part, are there for fine-tuning your programs.

TYPE	RANGE
int	-2,147,483,648 to 2,147,483,647
long int or long	-2,147,483,648 to 2,147,483,647
short int or short	-32,768 to 32,767
char	-128 to 127
unsigned int or unsigned	0 to 4,294,967,295
unsigned long	0 to 4,294,967,295
unsigned short	0 to 65,535
unsigned char	0 to 255

Table 3.1 Integer Types on the Macintosh (Hippo C)

Integer Representation

C allows you to express integers in decimal, octal, or hexadecimal form. This is for your convenience as a programmer; no matter what form you use to express the number, it is stored in binary form.

Octal Numbers. The octal, or base 8 system, uses the digits 0 to 7 to express numbers. This is how you would count to decimal 10 in octal: 1, 2, 3, 4, 5, 6, 7, 10, 11, 12. C interprets numbers with an initial digit 0 to be octal numbers. Thus, in a C program, 12 would be an ordinary dozen, but 012 would be the octal expression for decimal 10. See Appendix C for a further discussion of the octal system.

One exception to the required initial 0 is when the character constant notation is use. Thus, in '\101', the 101 is interpreted to be octal even though there is no initial 0.

Hexadecimal Numbers. The hexadecimal system is base 16. This requires more digits than 0 through 9, so the letters A through F (or a through f) are pressed into duty to represent the numbers 10 through 15. Again, see the Appendix C for further details. To indicate a hexadecimal number in C program code, precede the number with 0x (zero x) or 0X (zero X). Thus, 0X12 is the hexadecimal number $1 \times 16 + 2 \times 1$, or 18.

Keep in mind that these special notations are for your convenience so that you needn't make conversions. As far as the computer is concerned, the following statements all have the same effect:

```
x = 22;      /* 22 in decimal form */
x = 026;     /* 22 in octal form */
x = 0x16;    /* 22 in hexadecimal form */
```

Each places the binary equivalent of 22 in the x storage location.

Long Constants. On systems where long is different from int, it is sometimes necessary to indicate that a constant is to be stored as a long integer. This is done by appending a l or an L to the number's end. Thus, on a typical 16-bit system, the integer 34 would be stored in 16 bits of storage, but 34L would be stored in 32 bits. The capital L is preferable to the lowercase l because it is less likely to be confused with the digit 1.

Floating-Point Types

The other main class of basic types is the *floating-point* number. These are numbers with decimal points, like 3.14159 or 2.00. Sometimes they are written in "exponential notation," such as 1.86E5, which means 1.86 times 10 to the fifth power. Level 1 of Hippo C doesn't support floating-point numbers, so we will discuss them only briefly. One point to note is that they are stored differently. A floating-point number is still stored as a pattern of 1s and 0s, but some of the bits are interpreted to be a fraction and some are interpreted to be a power of 2 that is multiplied by the fraction. It is quite important for the computer to know whether a memory location holds a floating-point number or an integer, for the identical bit pattern will be interpreted quite differently.

The basic floating-point type is called **float**. For example, this declaration says the variable mpg is of type float:

```
float mpg;
```

The second floating-point type is called **double**, for double precision. Typically, it uses more storage bits than float so that it can store more decimal places.

Character Strings

The integer family (including `char`) and the floating-point family constitute the basic C types. Besides them, there are many derived types. One of the most important is the character string. We'll just talk about character string constants here, deferring character string variables until we introduce arrays.

Character String Constants. The program code representation of a character string constant is just a sequence of characters enclosed in double quotes. Our main use of them so far has been as arguments to the `printf()` function:

```
printf("Behold a character string!");
```

The tricky thing about character strings is that they come in all sizes. If the program has to store an `int` or `char` variable or constant, it knows exactly how much storage is needed. For a character string, however, the computer needs some way of keeping track of how much storage is required for each storage string. C does this by using a special character to mark the end of each string. This "string-termination" character is called the "null character," for its ASCII code number is 0. Thus, the C representation for this character is `'\0'`.

Consider our example above: "Behold a character string!" It contains 26 characters, including spaces and punctuation. It would be stored as 27 characters, the original 26 followed by a null character. See Figure 3.2

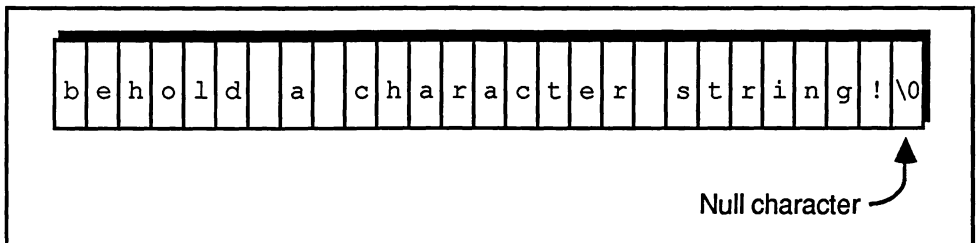


Figure 3.2 C String Storage

Note that we don't have to explicitly write out the null character. When the compiler sees a series of characters between double quotes, it knows to add the null character when storing the string.

A very important point to note is that the Macintosh Toolbox uses a different system for storing strings. Instead of using an end marker, it *precedes* the string with a character count showing the length of the string. See Figure 3.3.

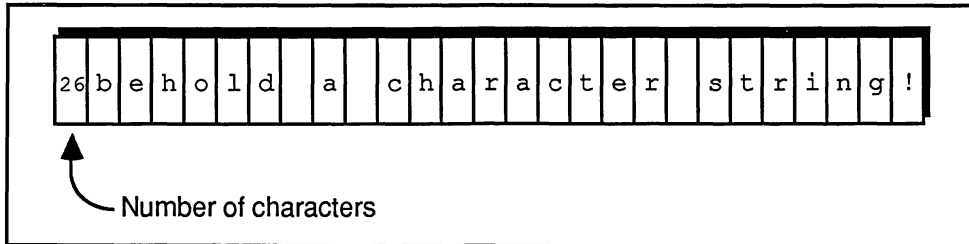


Figure 3.3 Toolbox String Storage

As a consequence, C programs using Toolbox functions sometimes have to convert from one form to the other. Fortunately, Hippo C provides functions to do just that, and we will discuss them when they become needed.

Formatted Output: `printf()`

We've seen how to print a character using `putchar()`. How do we go about printing other forms of data? In standard C, the `printf()` function handles a variety of types. This is an unusual function in that it can take a variable number of arguments. The trick is that if there are more than one argument, the first argument indicates how many additional arguments are present. Let's start with the one argument case.

Printing A Character String

Printing a character string is simple with `printf()`; just use the desired string as the argument. We've done that several times already. There are a some points to keep in mind. First, the `printf()` function doesn't automatically start a new line. The usual practice is to include a newline character at the end of each string so that the next output will start on a new line. With the nonbuffered input used by Hippo C, it is often convenient to

use an initial newline in order to separate a string response from the user's input. Again, we've done this in our examples.

Second, you can use the various alternate character representations as part of a string. We've often used `\n` at the end of a string, of course, but you could, for example, use it in the middle of a string:

```
printf("Twinkle twinkle\nLittle star!\n");
```

This produces the following output:

```
Twinkle twinkle
Little star!
```

Note that when a character is part of a string, we don't enclose it in single quotes. The double quotes alert the computer that everything within them represents a character.

Third, special care is needed for some characters. For example, the following won't work:

```
printf("Sally said, \"See me run!\" ");
```

The problem is that the computer would think the string ended at the second double quote, the one before `See`. This is where the backslash version is handy. The correct form is this:

```
printf("Sally said, \\\"See me run!\\\" ");
```

Now the outer quotes mark the string, and the inner backslashed, quotes get printed as ordinary double quotes.

The backslash alerts the compiler that some special attention is needed. But what if you wish to print a backslash? Use a double backslash to get a single one:

```
printf("/ These walls slope in. \\ ");
```

prints as

```
/ These walls slope in. \
```

One more difficult character to print is %, for, as we discuss next, it has a special meaning to the printf() function. To print a single %, you must use two:

```
printf("I'll give you 4%%.\n");
```

yields

```
I'll give you 4%.
```

Let's move on to more involved usages of printf().

Multiple Arguments

If printf() has more than one argument, the first argument should be a string. This string will contain a % symbol for each additional argument. The location of the % shows where the corresponding argument gets printed, and a code following the % indicates the form in which the value gets printed. An example will make this clear.

```
main()
{
    int num = 5;
    char ch = 'V';

    printf("Here is a number: %d\n", num);
    printf("The Roman version of %d is %c.\n", num, ch);
    printf("Here are two numbers: %d %d\n", num, num + 5);
}
```

Here is the output:

```
Here is a number: 5
The Roman version of 5 is V.
Here are two numbers: 5 10
```

In the first print statement, the `%d` shows where in the string the value of `num` will be printed; the `d` in `%d` indicates that `num` is an integer to be printed in decimal (base 10) form.

The second print statement has two additional arguments, so it needs (and has) two `%` entries. Once again, the `%d` indicates a decimal number is to be printed, while the `%c` form indicates a character will be printed.

The third print statement illustrates (again) that C functions pass arguments by value; what gets printed for the final argument is the value of 10 and not the literal `5 + 5` that appears in the argument list.

We call `%d` a "format specifier." Table 3.2 lists the basic format specifiers used by `printf()`.

SPECIFIER	OUTPUT
<code>%d</code>	decimal integer
<code>%c</code>	single character
<code>%o</code>	octal integer
<code>%x</code>	hexadecimal integer
<code>%u</code>	unsigned integer
<code>%f</code>	floating-point, decimal notation
<code>%e</code>	floating-point, exponential notation
<code>%g</code>	use the shorter of <code>%f</code> or <code>%e</code>
<code>%s</code>	character string

Table 3.2 Basic Format Specifiers

We'll take up the string format later.

The various integer formats correspond to the appearance of the output, not to the exact integer type. That is, if we want a decimal number, we would use %d for any of the integer types. Similarly, we can print the same integer using both %d and %o if we want to see decimal and octal versions of the same number. Here, for example, is an interactive program that prints out the ASCII code for input letters in decimal, octal, and hexadecimal:

```
#include "stdio.h"  /* program uses EOF */
main()
{
    int ch;  /* preferred declaration for EOF check */

    printf("This program prints ASCII codes. Enter a\n");
    printf("character and see the code in decimal, octal,\n");
    printf("and hexadecimal. Strike [OPTION] d to quit.\n");
    while ( ( ch = getchar() ) != EOF )
        printf("\n%c  dec: %d  oct: %o  hex: %x\n",
               ch,      ch,      ch,      ch);
}
```

Note that it is okay to spread a printf() statement over more than one line as long as the breaks occur between arguments. Here is a sample run:

```
This program prints ASCII codes. Enter a
character and see the code in decimal, octal,
and hexadecimal. Strike [OPTION]-d to quit.
K
K  dec: 75  oct: 113  hex: 4b
a
a  dec: 97  oct: 141  hex: 61
t
t  dec: 116 oct: 164  hex: 74
e
e  dec: 101 oct: 145  hex: 65
[OPTION]-d
```

Field Widths

In the examples you've seen so far, `printf()` uses however much space is needed to print a number or character. The number of character widths used in printing a value is the "field width," so the `printf()` default is that the field width matches the size of the number. You can choose a field width, however, by including the desired width in the format specifier. For example, to cause 5 spaces to be set aside for a number, use the format specifier `%5d`. If the number is bigger than that, then the field is expanded automatically.

Normally, numbers are "right-justified" in a field; that is, they are aligned with the right side of the printing field. Using a `-` sign in the format specifier causes the number to be "left-justified." For example, suppose we have this statement:

```
printf("***%5d**%-5d**\n", 12, 12);
```

Then the output would be this:

```
**    12**12    **
```

In each case, a field 5 spaces wide was used, with the 12 printed in the right of the first field and the left of the second.

Printing Long Integers

In Hippo C, since `long` is the same as `int`, no special steps need be taken to print long integers. In implementations having `long` longer than `int`, however, you need to use the `l` modifier to print long integers. The modifier would come just before the format specifier. Here are two examples:

```
printf("Here is a long integer: %ld\n", alongint);  
printf("Here is another look: %l5lx\n", alongint);
```

The second example prints the integer in hexadecimal form, left-justified in a field 15 characters wide.

Floating-Point Types

C offers three format specifiers for float and double values. The first is `%f`. It causes a value to be printed using decimal fractions. By default, six places to the right of the decimal are printed. Suppose, for instance, we have the following code on a system that supports floating-point values:

```
float cost = 259.99;
printf("The cost is $%f\n", cost);
```

The output would look like this:

```
The cost is $259.990000
```

The field width can be specified by placing the desired field width after the `%`. Also, the number of places to the right of the decimal can be indicated by placing a period followed by the desired number of places just before the `f`. Thus, the specifier `%10.2f` means to use a field width of ten with two places to the right of the decimal. Hence the following code

```
float cost = 259.99;
printf("The cost is $%10.2f\n", cost);
```

produces this output:

```
The cost is $      259.99
```

Note that the decimal counts as one character width. As with the other specifiers, the field width is expanded to accommodate the number if the specified width is too small. The hyphen modifier can be used to make the number left-justified in its field.

The `%e` format is used to produce the power-of-ten notation. The exponent is selected so that there is one digit to the left of the decimal; by default, six digits are displayed to the right of the decimal. The exponent is written as a signed, two-digit number. Consider this example:

```
float cost = 259.99;
printf("The cost is $%e\n", cost);
```

The output would look like this:

```
The cost is $2.599900E+02
```

The field width and the number of decimals can be specified in the same manner as for `%f`:

```
float cost = 259.99;
printf("The cost is $%10.2e\n", cost);
```

produces this output:

```
The cost is $ 2.60E+02
```

If you use the `%g` format, the computer uses the `%e` or the `%f` format, whichever is the shorter.

Formatted Input: `scanf()`

The `scanf()` function is the input counterpart to `printf()`. It, too, uses a format string to describe how many and what type values to read. For instance, the format string `"%d %d"` would tell the function to read two integers in decimal form. Here is an example using `scanf()`; look for something unusual:

```
main()
{
```

```

int num1, num2;

printf("Please enter two integers:\n");
scanf("%d %d", &num1, &num2);
printf("That was %d and %d.\n", num1, num2);
}

```

Here is a sample run:

```

Please enter two integers:
39 124[RETURN]
That was 39 and 124.

```

Well, it seems to work fine, but why are there ampersands in front of `num1` and `num2`? The ampersand (`&`) is yet another C operator, the "address operator." The combination `&num1` represents the *address*, or memory location, of the `num1` variable. What this means is that instead of using `num1` for an argument, we used its address. The reason we did this is important to Understanding C, so let's take a moment or two to discuss it.

Address Arguments: Pointers

Suppose we used this function call:

```
scanf("%d %d", num1, num2); /* naive, incorrect usage */
```

What would it do? As we have emphasized before, it will pass the numerical *values* of `num1` and `num2` to the `scanf()`. At this point, `num1` and `num2` don't even have values yet. More importantly, `scanf()` doesn't *want* to know what the values of `num1` and `num2` are. Its purpose is to *give* values to those variables.

Providing address arguments allows `scanf()` to perform its giving. Let's see how that works. In our example, `scanf()` obtained the values 39 and 124 from the keyboard. It knows the addresses of `num1` and `num2`, because those are the arguments we passed to it. The function then takes the numbers it gathered and places them into the specified memory locations. Voila! The memory locations labeled `num1` and `num2` now contain the

values 39 and 124. Giving a function the *address* of a variable allows the function to change the *value* of a function. For those of you familiar with Pascal, using the address operator in an argument is similar to declaring a Pascal argument to be a variable parameter.

In C, the symbolic representation of an address is called a *pointer*. Many Toolbox functions use pointers as arguments, so learning pointers is essential to programming for the Macintosh in C. We will discuss them throughout this book.

Later, when you study functions further, you'll see how to write code using pointer arguments. Meanwhile, we will just note which standard C and Toolbox functions require address arguments. In general, these will be functions whose purpose is to alter the value of variables in the calling program.

Specifying scanf() Formats

Like printf(), scanf() has several possible format specifiers. Table 3.3 lists them.

SPECIFIER	INPUT INTERPRETATION
%d	decimal integer
%o	octal integer
%x	hexadecimal integer
%s	character string
%e or %f	floating-point number

Table 3.3 **scanf() Format Specifiers**

We'll discuss string input later. The various integer forms of input do not expect you to use the in-program rules for representing integer constants. That is, if you wish to enter the hexadecimal number 0x38, you just type 38, and the %x specifier insures that it is interpreted as hexadecimal rather than as decimal.

In Hippo C, if you wish to read a value into a short variable, you need to include an h modifier in the format specification:

```
scanf("%hd", &iq);  
scanf("%hx", &temp);
```

Implementations having short and int the same don't need to do this.

Similarly, implementations that have long longer than int should use the l modifier when using scanf() to read an integer destined to stored in a long variable:

```
scanf("%ld", &population);
```

It causes no harm to use the l or h modifiers in implementations that do not require them.

Either the %e or the %f formats can be used to read values into type float variables. The two are equivalent; both can read fixed decimal point or power-of-ten notation. To read values for a type double variable, use the l modifier. Thus, %le or %lf can be used for that purpose.

How scanf() Scans

Except in the %c mode, scanf() skips over spaces, blanks, and newlines. Thus, when our program asked for two numbers, we could have answered this way:

```
39 [RETURN]  
[RETURN]  
128 [RETURN]
```

How does scanf() know to read two digits for the first number and three digits for the second? It scans the input characters one at a time until it finds a digit. That starts the number-reading process. Then it continues reading until it finds a nondigit. That tells scanf() it has reached the end of the number. Then scanf() places the nondigit back in the input queue. The next input call, be it getchar() or scanf(), will start with that character. Note

that our program will not continue until you strike one more key after the 8 in 128; until a nondigit shows up, `scanf()` has no way of knowing that you are done. We hit the [RETURN] key, but any other regular key would work. However, hitting, say, a [V] key would cause problems if the program were supposed to read another number later, for it would then try (and fail) to read V as a number.

In the `%c` mode, `scanf()` works pretty much like `getchar()`; that is, it will read the next character, be it letter, space, newline, or whatever.

What scanf() Returns

The `scanf()` function, as we have seen, uses pointers to transport keyboard input to variables. The function also has a return value, which we didn't use in our program. It uses the return value to provide a status report on its efforts. Normally, it returns the number of items read. That would be two in our program. But suppose we had responded to the request for two integers by typing the following:

```
two integers
```

Then `scanf()` would have balked because it would have found the character `t` instead of a digit. In this case, it would stop reading, put the `t` back in the input queue, and return a value of 0.

Another possibility is that the first thing we typed could be the end-of-file signal ([OPTION]-d). When it encounters end-of-file, as in that case, `scanf()` returns a value of -1. This lets you use the return value to control a loop, as we will soon see.

scanf() versus getchar()

Both `scanf()` and `getchar()` can be used to read input characters. The approach is different. `Getchar()` uses the return value to communicate the character to the calling program:

```
ch = getchar();
```

Scanf(), on the other hand, uses an address, or pointer, to communicate the value back:

```
scanf("%c", &ch);
```

These are the two chief ways in which a function can communicate values to the calling program. The return value method is the simpler and the more direct, but it is limited in that it can provide just *one* value at a time. The pointer method, however, can provide several values at a time through the expedient of using several pointer arguments.

On a more practical level, getchar() is much more efficient for single-character input than scanf() because getchar() is designed solely for that task.

Toolbox Examples

Now that you have seen some of the standard C techniques for I/O, let's look at some specifically Macintosh examples. One difference between Macintosh screen text output and that for traditional screens is that the Macintosh offers typesetting options: various fonts, faces, and character sizes. Naturally, it also must provide functions for controlling these choices. We'll look at one of these functions now.

Font Size: textsize()

The function that controls the font size is called **textsize()**. It takes one argument, an integer representing the text size. The integers correspond to the sizes listed in the Style menu for, say, Macwrite. Thus, 12 is the standard (default) size, with larger numbers corresponding to larger types. You can specify any size, but the best results come from using one of the Style menu selections. One special case is that an argument of 0 produces the regular system size. Below is an interactive program demonstrating the use of this function. Note that we use the return value of the scanf() function to control the while loop. Entering a nonnumerical character (other than a space, tab, or newline) makes the return value 0, and an end-of-file causes a -1 to be returned; either condition stops the loop.

TextSize Sets type size

```
main()
{
    int size;

    printf("Enter font size; to quit, enter EOF or a ");
    printf("nondigit.\n");
    while ( scanf("%d", &size) == 1 )
    {
        textsize(size);          /* successful read */
        printf("This is font size %d.\n", size); /* set font size */
    }
}
```

Note that the while loop test examines the return value of `scanf()` (the status report) and not the value of `size` for its continuation test. That is, entering a numerical value other than 1 won't stop the loop, but entering a letter or signaling EOF (either of which return a non-1 status) will stop the loop. Figure 3.4 shows a sample run. Note that once a font size is set, the new size stays in effect until the font size is reset.

We used a standard C output function in our example, but the Toolbox has its own output functions. For example, to print a single character, we can use `drawchar()`. It works like `putchar()` with some minor differences. First, it has no return value. Second, it is not buffered. Compare the following two programs and their outputs; the program is a slight variant of one earlier in this chapter.

First, use `putchar()`:

```
#include "stdio.h"    /* for EOF definition */
main()
{
    int ch;
```

```

    while ( (ch = getchar()) != EOF)
        putchar(ch);
}

```

Here is a sample run:

```

Don't repeat this! [OPTION]-d
Don't repeat this!

```

```

*
*
* a.out
Enter font sizes; to quit, enter EOF or a non-digit.
10
This is font size 10
12
This is font size 12
18
This is font size 18
24
This is font size 24
0
This is font size 0
d
*

```

Figure 3.4 Sample Run of Font Size Program

Now replace `putchar()` with `drawchar()`:

```

#include "stdio.h"
main()
{
    int ch;
    while ( (ch = getchar()) != EOF)
        drawchar(ch);
}

```


DrawChar

Draw a character on the screen

Here is a sample run:

```
Ddoonn'tt  rreeppeeaatt  tthhiiss!![OPTION]-d
```

Here, each character (including spaces) is repeated as it is entered. With the `putchar()` version, output was saved in a buffer, then flushed when the program ended.

String Output: drawstring() and strtoc()

The **drawstring()** function takes a string as an argument and prints it. However, because it is a Toolbox function, it uses Macintosh's augmented Pascal strings, and not C strings. As we mentioned before, these two strings are stored differently. The C string has its end marked by the null character, while a Pascal string uses the initial byte to store the string length. Writers of Macintosh C compilers supply conversion functions to bridge the gap. In Hippo C, the conversion function is called **strtoc()**, for "string: C to Pascal." Other compilers may implement the conversion differently or not at all, but it is simple enough (once you know enough) to write your own function to do it. The next example shows how to use these functions:

```
#define STRING "Tie me kangaroo down, boys..."
main()
{
    printf(STRING);
    drawstring( strtoc(STRING) );
}
```

DrawString

Draws a string of characters on the screen

Running it produces this output:

```
Tie me kangaroo down, boys...Tie me kangaroo down,  
boys... ↩
```

Each I/O function did its job properly. One difference, as we noted, is that `drawstring()` cannot use `STRING` as an argument. But it can and does use the return value of `strctop()` as an argument. Second, `drawstring()` ignores formatting instructions such as the newline character; instead it prints the missing-character symbol.

Summary

C programs use a variety of functions to handle input and output. The `getchar()` and `putchar()` functions are designed to handle character I/O output efficiently, while `printf()` and `scanf()` are multimode I/O functions, capable of dealing with numbers and strings as well as with single characters. The `scanf()` function uses address, or pointer, arguments to indicate which variable locations are to have data placed into them. Thus, to read in a value for the variable `x`, we would use `&x` as an argument, where `&` is the address operator.

The two basic classes of storage are **integer** and **floating-point**. Both use patterns of 0s and 1s, but they are interpreted differently. Macintosh C provides three sizes of integer storage: `int` (4 bytes), `short` (2 bytes), and `char` (1 byte). A fourth C type (`long`) is identical to `int` in Hippo C's Macintosh implementation. In addition, each of these types can hold positive and negative values, or, if preceded by the keyword `unsigned`, just nonnegative integers.

The character string is a derived type. A C string has its end marked by the null character, `'\0'`. Toolbox strings, on the other hand, use the initial byte to provide the length of a string.

C provides a **preprocessor** that processes program code before it is compiled. One key preprocessor directive is `#define`, which allows you to define symbolic constants. A second important directive, `#include` allows you to easily incorporate information from other files into your program.

Toolbox I/O functions can be used to control the rather flexible Macintosh output environment. The `textsize()` function, for example, controls the font size.

4

Control Statements

In this chapter you will learn about:

- Three loops: while, do...while, and for
 - Choosing with if and if...else
 - Relational operators
 - Logical operators
 - Switch and other jump statements
 - The conditional and comma operators
-

The simplest form of program is a sequence of instructions that the computer follows in order with no variation. Such a program does not take full advantage of a computer's strengths and abilities. C, like most other programming languages, offers a class of instructions, called "control statements", that put much more of the computer's power at your disposal. Control statements make it possible for a computer to deviate from a simple sequential execution of instructions, making programs more flexible and powerful. One group of control statements allows "branching". This means at certain points a program can choose among two or more alternative paths, depending on conditions at those times. A second group of control statements allows "looping", which means running through the same instruction set repeatedly.

In C, the branching statements are the if statement, the if...else statement, and the switch statement. There is also a "conditional operator" which serves the same purpose. Looping is provided by the while statement, the do...while statement, and the for statement. We will look at all of them in this chapter.

These control statements (aside from the switch statement) share in common the use of conditional expressions that are checked for truth or falsehood. We will discuss conditional expressions and the associated operators.

The while Loop

You've already met this loop informally. More formally, its general form is this:

```
while ( condition )  
    statement
```

If the condition is true, the statement is executed and the condition rechecked. This continues until the condition becomes false. Once this happens, program flow continues on to whatever follows the while statement.

The *statement* section must be a single statement or else a block of statements contained between braces. Suppose we have a program fragment like this:

```
count = 0;  
while ( count != 10 )  
    printf("The count is %d.\n");  
    count++;          /* don't be fooled by the  
indentation */  
    printf("Done!\n");
```

We've indented it to imply that `count++`; (increasing count by 1) is part of the loop. But the compiler ignores extra spaces, and it thus does not include the `count++`; statement as part of the while statement. Indeed, we have created a dread infinite loop, one that prints

```
The count is 0.
```

endlessly. Only the *first* statement following the condition is part of the loop.

Compound, or Block, Statements

The key to including several actions within a loop is to use a "compound statement", also known as a "block". This, as you saw earlier, is a sequence of statements enclosed within a pair of braces. The block counts as a single statement, so all the statements within a block can be

included within the loop. Thus, the proper way to write the preceding example is this:

```
count = 0;
while ( count != 10 )
{
    printf("The count is %d.\n"); /* start of block */
    count++;                      /* end of block   */
}
printf("Done!\n");
```

Some programmers prefer to place the opening brace on the same line as the while. Because C is a free-format language, the choice is purely a matter of stylistic preference.

A Graphics Loop

Let's try out the loop structure using some of the Macintosh graphing functions. Here is a program that draws nested squares; it uses the `moveto()` and `lineto()` functions that we introduced in Chapter 2:

```
/* box.c -- a program that draws nested boxes */

#define H0 256      /* horizontal screen center */
#define V0 171      /* vertical screen center */
#define LIMIT 100   /* size factor for largest box */
#define START 10    /* initial size factor */
#define INCREASE 10 /* size increase */

main()
{
    int scale = START; /* size scale */

    while ( scale <= LIMIT)
    {
        moveto(H0 - scale, V0 + scale);
        lineto(H0 + scale, V0 + scale);
        lineto(H0 + scale, V0 - scale);
        lineto(H0 - scale, V0 - scale);
        lineto(H0 - scale, V0 + scale);
        scale = scale + INCREASE; /* increase size */
    } /* loop end */
}
```

Before running this program, use the mouse to move and expand the Hippo C Command Window to approximately full screen. Figure 4.1 shows the output of this program.

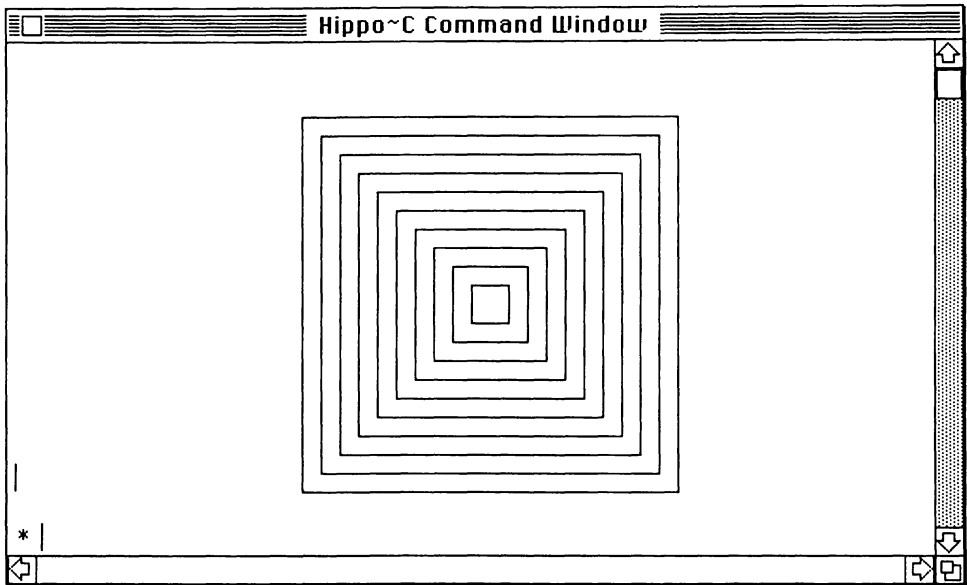


Figure 4.1 Output of Box-drawing Loop

The `<=` symbol means "less than or equal to". Note that we used `#define` directives for all the constants within the program. Again, this is a matter of recommended programming style. Another recommended documentation step is to include a program name and brief description in opening comments. This heading can be expanded to include such items as author, version, and date of writing.

The do...while Loop

The while loop is an "entry condition loop". That means the condition is checked *before* the statements in the loop are executed. It's a matter of looking before you loop. If the condition is false to begin with, then the loop may be skipped entirely.

C also offers an "exit condition loop", in which the condition is checked *after* each loop execution. This form of loop must be executed at

least once, for one execution comes before the first test. This loop is the do...while statement, and its general form is this:

```
do
    statement
    while ( condition );
```

The statement is a single statement, which can be a block. Note the terminal semicolon after the condition test.

This form of loop is used much less frequently than the others. One circumstance in which it is used is when you wish to process input up to *and including* some particular character. For example, if you wanted to read and reprint a sentence, you could have a program fragment process material up to and including a period:

```
do {
    ch = getchar();
    putchar(ch);
} while ( ch != '.' );
```

By the time this fragment finds that *ch* is a period, it already has printed it. But that was what we wanted, and the loop stops there.

Another situation in which the do...while loop is used is when the user is given a list of acceptable responses (such as "y" or "n"). A do...while loop can fetch an answer and test its worthiness, recycling until a valid reply is made. Because the response has to be obtained before it is checked, an exit condition loop makes sense. You'll see an example later in this chapter.

The for Statement

C has one more loop, the **for** loop. The C for loop is very flexible; it can do anything a while loop does, and vice versa. Typically, however, the for loop is used when the number of iterations is known before the loop begins, while a while loop is typically used when the number of iterations is not known in advance, as in our character-counting program in Chapter 3.

Here is a for loop that prints the first five integers and their squares:

```
main()
{
    int n;

    for ( n = 1; n <= 5; n++ )
        printf("%2d %4d\n", n, n*n);
}
```

Here is the output; note that by using field width specifiers in the printf() format, we achieve uniform columns:

1	1
2	4
3	9
4	16
5	25

Note that the loop's control instructions (the part in parentheses) contain three distinct expressions separated by semicolons. The first expression ($n = 1$) is an initialization expression. It is performed *once*, before the loop is started. The second expression ($n \leq 5$) is a test condition. It is evaluated *before* each cycle of the loop. The for loop is an entry-condition loop, so once the test condition becomes false, the loop stops before cycling again. The third expression ($n++$) is executed at the end of each loop cycle. Here it serves to update the value of n by increasing it by 1. Here is the general form for the for loop:

for (*initialization*; *test condition*; *update*)
 statement

As with other loops, the *statement* section is a single statement that may be a block.

The most typical use for a for loop is to run a process through a fixed number of cycles. Here are two common idioms for cycling a fixed number of times:

```
for ( n = 1; n <= 10; n++)
    ...;

for ( n = 0; n < 10; n++)
    ...;
```

Both go through 10 cycles. The first cycles *n* from 1 through 10, and the second cycles *n* from 0 to 9. The second form will come in handy for arrays, whose elements are numbered starting at 0.

One advantage of the for loop is that it gathers together in one place all the expressions that control the behavior of the loop. Our example, for instance, shows the initial and final values of *n* and that *n* increases by 1 each cycle. This makes the for loop much less susceptible to "one-off" counting errors than are loops like the following:

```
n = 1;
while ( ++n < 10 )
    printf("%d", n);

n = 1;
while ( n++ < 10 )
    printf("%d", n);
```

The first prints the integers 2 through 9 and the second prints the integers 2 through 10, but it takes a moment or two of inspection to make that clear to a reader.

We don't have to plod along increasing *n* by 1 each time. We can just as easily add 2 or 3 to *n* or even multiply it each cycle. Here is an example that prints out powers of two less than 2000. The variable *n* starts out with a value of 1 and is doubled each cycle:

```
#define BASE 2
main()
{
```

```

int n;

for ( n = 1; n <= 2000; n = n * BASE )
    printf("%4d\n", n);
}

```

Here is the output:

```

1
2
4
8
16
32
64
128
256
512
1024

```

After printing the value 1024 the loop updates `n` to 2048, notices that this is larger than 2000, and quits.

Because the `for` loop gathers together several conditions in one place, it's often useful to express the conditions as concisely as possible. C has several operators that allow you to express yourself succinctly. You've seen the increment operator already. Now let's look at another class of space-saving operators.

Other C Assignment Operators

In the powers-of-two example, we used this expression:

```
n = n * BASE
```

C has a shorthand notation for this operation:

```
n *= BASE
```

The `*=` combination represents the "multiplicative assignment" operator. It means take the variable at the left, multiply it by the value to the right, and assign the product to the variable.

This form is more compact to type and typically is implemented more efficiently on a computer. C has several such operators. Here is a list of the arithmetic ones.

OPERATOR	MEANING
<code>+=</code>	Add value on the right to variable on the left
<code>-=</code>	Subtract value on the right from variable on the left
<code>*=</code>	Multiply value on the right times variable on the left
<code>/=</code>	Divide value on the right into variable on the left
<code>%=</code>	Take modulus of variable on the left with respect to the value on the right

In each case, the resulting value is assigned to the variable on the left. For example, the statement

```
finger -= 2;
```

decreases the value of `finger` by 2.

Nested Loops

The statement section of a loop can include another loop. A loop inside another is said to be *nested*. What happens in a nested loop is that each single cycle of the outer loop causes the inner loop to go through its entire sequence of loop cycles. Let's study an example. We'll start with a single loop, one that prints a row of big asterisks:

```
#define HSTEP 30      /* horizontal movement steps */
#define HMAX 450      /* horizontal limit */
#define BIGFONT 24
main()
{
```

```

    int h;                /* horizontal position */
    textsize(BIGFONT);    /* set large font size */
    for ( h = HSTEP; h <= HMAX; h += HSTEP)
    {
        moveto( h, 20);
        drawchar('*');
    }
}

```

One interesting feature of this program is that we use the `moveto()` function to position the screen "pen" for making a character rather than for drawing a figure. With the Quickdraw package, graphics and text are integrated.

The program moves the pen in horizontal steps of 30 across the screen, drawing a large asterisk at each step. We used the additive assignment operator described earlier to increase the value of `h` each cycle.

To get several rows of asterisks we can nest the original loop in one that adjusts the vertical position. Here is that modification:

```

/* stars.c -- makes large asterisks (using Quickdraw) */
#define HSTEP 30      /* horizontal movement steps */
#define HMAX 450      /* horizontal limit */
#define VSTEP 20      /* vertical movement steps */
#define VMAX 240      /* vertical limit */
#define BIGFONT 24
main()
{
    int h,v;           /* horizontal, vertical positions */

    textsize(BIGFONT); /* set large font size */
    for ( v = VSTEP; v <= VMAX; v += VSTEP)
        for ( h = HSTEP; h <= HMAX; h += HSTEP)
        {
            moveto( h, v);
            drawchar('*');
        }
}

```

Figure 4.2 shows a sample run. Note that the `a.out` command we gave to start the program remains visible. In Chapter 8 you'll learn how to clear the part of the screen used by the program.

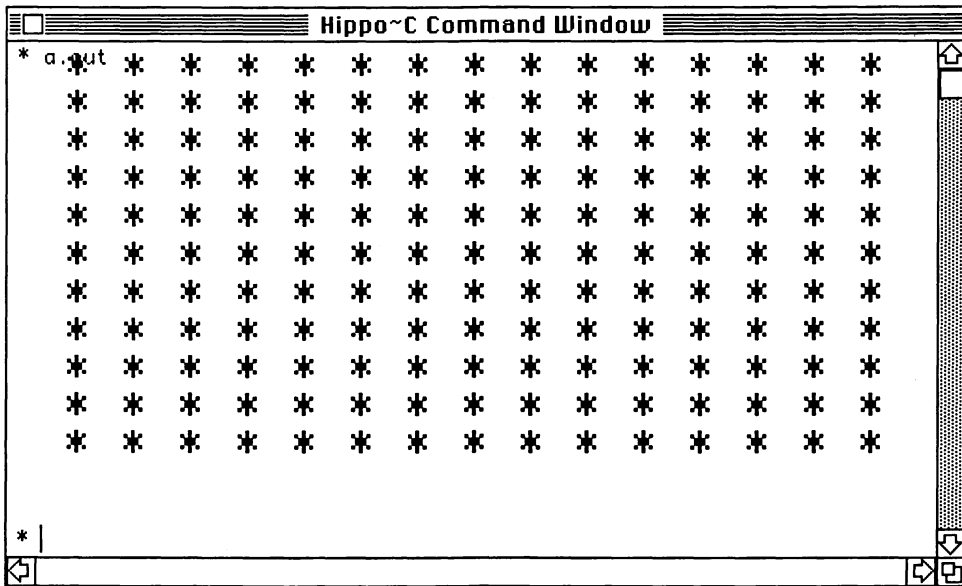


Figure 4.2 Field of Asterisks

At first a whole line of asterisks is printed with *v* (the vertical coordinate) equal to 20; that is one complete cycle set for the inner loop. When the inner loop finishes, *v* is increased, and the second cycle of the outer loop is begun. This results in another complete cycle set for the inner loop, this time printing a row with *v* equal to 40.

The entire inner loop is a single *for* statement, so it is not necessary to enclose that loop in braces.

This program prints the asterisks a row at a time; by reversing the order of the loops, you can make it print a column at a time.

Let's move now from looping to choice-making.

Two if Statements

An if statement lets a program choose among options, depending on conditions at the time the choice is made. C has two varieties of *if* statements: the simple *if* and the *if...else*.

The if Statement

The general form of the if statement is this:

```
if ( condition )  
    statement
```

The *statement* part can be a single statement or a block. If the *condition* is true, then the statement is executed. Otherwise, the program skips on to the next statement. Here is an example:

```
/* spacecount.c -- counts spaces up to EOF */  
#include "stdio.h"          /* for EOF definition */  
main()  
{  
    int ch;  
    int spaces = 0;  
  
    while ( (ch = getchar()) != EOF)  
        if ( ch == ' ' )      /* is ch a space? */  
            spaces++;        /* if so, count it */  
    printf("\nSpace count = %d\n", spaces);  
}
```

As indicated, it counts spaces in input. Note that the complete if statement is a single statement, so that we did not have to include it in braces to show that it all was part of the while loop. The spaces variable gets incremented every time a space shows up. Remember that == is the C operator for comparing for equality. We'll run down all the comparison operators soon.

The if...else Statement

The if...else statement offers two choices. One is done only if the test condition is true, and the other is done only if it is false. The general form is this:

```
if ( condition )
    statement 1
else
    statement 2
```

Statement 1 is executed if the condition is *true*, and statement 2 is executed if the condition is *false*. Each can be a single statement or a block.

Here is an example that selectively modifies its input:

```
#include "stdio.h"
main()
{
    int ch;

    while ( (ch = getchar()) != EOF )
    {
        /* nonessential brace */
        if ( ch == 'o' )      /* be sure to use == */
            putchar('a');    /* substitute a letter */
        else
            putchar(ch);     /* or print as is */
    }
}
```

Here is a sample run:

```
Love a hog[RETURN]
Lave a hag
[OPTION]-d
```

Note that the entire if...else statement counts as a single statement, even though each statement section within it has its own semicolon. Thus no braces are needed to show the extent of the loop. However, when a

single loop statement gets that long, it is a good idea to use braces anyway. The compiler may not need them, but they may be useful to a reader.

We included the caution to not use `=` when `==` is intended. The `==` is a comparison operator; it asks if `ch` and `'o'` have the same value. The `=`, on the other hand, is the assignment operator, and it would assign the value `'o'` to `ch` so that every `ch` would be converted to an `'o'`. The compiler will not catch the misuse of `=` for `==`, because `=` can also be used legally in a test condition. We'll return to this topic soon, but first let's look at one further extension of the `if` family.

Multiple Choice

The basic `if...else` statement offers two choices. But the second choice can be another `if...else` statement, and so on. This allows you to encompass many choices. Here is a program fragment illustrating how to set up *multiple* choices.

```
printf("How many legs does your pet have?\n");
scanf("%d", &legs);
if (legs == 2)
    printf("I think your pet may be a turkey.\n");
else if (legs == 4)
    printf("I think your pet may be a mutant
           turkey.\n");
else
    printf("I think your pet is unusual.\n");
```

The indentation scheme makes it simple to scan the list of choices, here 2 legs, 4 legs, or some other number of legs. You could also format the fragment this way:

```
printf("How many legs does your pet have?\n");
scanf("%d", &legs);
if (legs == 2)
    printf("I think your pet may be a turkey.\n");
else
    if (legs == 4)
        printf("I think your pet may be a mutant
               turkey.\n");
    else
        printf("I think your pet is unusual.\n");
```


This emphasizes the fact that the actual form is one if...else using a second if...else as its second alternative action. One disadvantage is that it keeps pushing successive if...elses farther to the right on the screen.

Conditional Expressions

The loop structures and if structures we've seen have all used "conditional expressions" to control the program flow. (They can also use arithmetic expressions, but that's a matter for later discussion.) Conditional expressions are ones we can think of as being "true" or "false", and they are created by using relational operators, which compare expressions; and logical operators, which combine expressions. We've already seen most of the relational operators, but let's gather them together now and introduce them formally.

Relational Operators

Relational operators examine the comparative values of two expressions. Is one bigger than the other? Are they the same? Table 4.1 summarizes C's relational operators.

OPERATOR	MEANING
<	is less than
<=	is less than or equal to
==	is equal to
>=	is greater than or equal to
>	is greater than
!=	is not equal to

Table 4.1 Relational Operators

The relational operators have lower precedence than the arithmetic operators but higher precedence than the assignment operators. For example, we have been using the idiom

```
(ch = getchar() ) != EOF
```

The parentheses around the first part of the expression are necessary, for without them, the relational comparison gets done first. That is, omitting the parentheses has the same effect as parenthesizing in the following fashion:

```
ch = ( getchar() != EOF)
```

First, this asks if the return value of `getchar()` is EOF or not. Then the answer to this question ("true" or "false") is assigned to `ch`. Remember, every expression in C has a value, and in this case `ch` is assigned the value of the relational expression `getchar() != EOF`.

True and False in C

This raises an important point: what is the value of a relational expression? We've been terming the values "true" and "false", but how is that expressed in C? Unlike, say, Pascal, C does not have a special "Boolean" type restricted to just "true" and "false" values. Instead, "true" and "false" are represented by integer values. Let's run a short program to see what they are:

```
main()
{
    int true, false;

    true = ( 100 > 4);
    false = ( 4 > 20);
    printf("TRUE is %d and FALSE is %d\n", true, false);
}
```

And the answer is ...

```
TRUE is 1 and FALSE is 0
```

These are the values used by Hippo C and most other C compilers. (Some use other values for true, but we will stick with the standard values.) Thus the numerical value of a relational expression is either 1 (true) or 0 (false).

Other Truths

When we use a relational expression as a test condition for a loop or if statement, the condition has a value of 1 or 0. However, the test condition can be any expression, not just a relational one. In general, *any nonzero* value is interpreted as true. This is both a blessing and a curse. First, let's look at the curse aspect. Earlier we had this program fragment:

```
if ( ch == 'o' )
    putchar('a');
else
    putchar(ch);
```

This was part of a loop that replaced each o with an a, leaving the other characters unchanged. Now replace the == with a = operator:

```
if ( ch = 'o' )      /* an easily made error */
    putchar('a');
else
    putchar(ch);
```

Here, the test condition assigns 'o' to ch. The value of the test expression becomes the value of ch, which is now the ASCII value of the character 'o' which is 111. But 111 is "true", so no matter what the original value of ch was, the putchar('a') choice is executed; every letter is replaced with an a.

The use of = instead of == is a subtle error to detect. It can escape a quick visual scan of the program, and the compiler thinks it's just fine. If you find a program is always making the same choice, regardless of the data it's fed, look for this sort of error.

Notes About C : Increment and Decrement Operators

The increment operator (++) adds one to its operand, and the decrement operator (--) subtracts one from its operand. Each comes in two forms: the prefix form, as in ++x, and the postfix form, as in x++. The two forms have the same effect when used as a stand-alone statement. That is, each of the following two complete statements produces the same result, increasing x by one:

```
++x;  
x++;
```

The difference between the two forms becomes apparent when they are used as *part* of an expression. The distinction is this: in the prefix form, the incrementation takes place *before* the expression is evaluated, and in the postfix form, the incrementation takes place *after* the expression is evaluated.

Let's look at some specific examples. Suppose we start with the variable x having the value 5 and that we have the statement

```
y = 10 * x++;
```

For this case, 10 is multiplied times x (which has the value 5), and *then* x is incremented. Thus, after this statement is executed, x is 6 and y is 50.

Now, starting with x equals 5 again, consider the prefix form:

```
y = 10 * ++x;
```

In this case, *first* x is incremented to 6, and *then* x is multiplied by 10. This statement results in x being 6, as before, and in y being 60, not as before.

Once again, for the prefix form, the order is increment and evaluate, while for the postfix form, the order is evaluate and increment.

On the other hand, the fact that all nonzero values are true can be used to simplify your programming. For example, here is one common idiom:

```
while ( i-- ) ...
```

The -- is the "decrement" operator. It subtracts 1 from the value of its operand, here the integer variable i. This loop, assuming i initially has

some positive value, will continue until *i* reaches zero. In other words, the loop works the same as the following example:

```
while ( i-- != 0 ) ...
```

In the first case, the loop ends when *i* itself becomes zero. In the second case, the loop ends when the expression *i-- != 0* becomes zero ("false"). Since both conditions happen simultaneously, the two forms work the same.

Once again, we've introduced something new: using the decrement operator as part of a larger expression. See the box for more on this topic.

Truth and Functions

Since true and false are treated as integer values, it is possible to use integer-returning functions as Boolean (true-false) functions. These functions return 0 if the condition they test for is false, and nonzero otherwise.

Suppose, for example, we want a program to count characters in input (as we have done before) and also to count alphabetic letters. We could use 52 if statements (one for each lowercase and one for each uppercase letter) to identify all the possible letters, but that borders on the foolish—and from the wrong side. We could use logical operators to define a range of characters, but we haven't discussed that yet. Best of all, we can use the `isalpha()` function, which tells us whether or not its argument is a member of the alphabet. Here's how to do it with Hippo C:

```
/* chspct.c -- counts characters and spaces */
#include "stdio.h"
main()
{
    int ch;
    int chars = 0;
    int letters = 0;
    while ( (ch = getchar() ) != EOF)
    {
        if ( isalpha(ch) )
```

```

        letters++;      /* only for letters */
        chars++;        /* for all characters */
    }
    printf("\nchars = %d and letters = %d\n", chars,
letters);
}

```

Here is a sample run:

```

See this work.[RETURN]
[OPTION]-d
chars = 15 and letters = 11

```

Note again that spaces, punctuation, and newlines are characters and are part of the character count.

Standard C has a flock of is functions. The exact list varies from implementation to implementation, but Table 4.2 lists the most common ones.

NAME	TRUE IF ARGUMENT IS
isalpha()	alphabetic
isupper()	upper case
islower()	lower case
isdigit()	a digit
isalnum()	alphanumeric
isspace()	a space, tab ('\t'), or newline ('\n')
ispunct()	punctuation

Table 4.2 Common "Boolean" Functions

On many systems these functions are actually "macros", pseudofunctions created with #define statements. We will discuss macros in Chapter 5. One point to note now, however, is that macro definitions are

typically stored in a file named `ctype.h`, and that this file has to be `#included`. However, with Hippo C, this isn't necessary.

Logical Operators

Often it is convenient to combine two or more test conditions together. Was the response a `y` or a `Y`? Is the boxer's weight above the minimum and below the maximum? Or perhaps you wish to modify a condition so that it tests for nonletters instead of for letters. Logical operators give us the means to express such combined conditions and modifications. C has three such operators: the **NOT** operator, the **AND** operator, and the **OR** operator.

The NOT Operator: ! The NOT operator is a contrary one. It makes true false and false true. It takes a single operand, which it precedes. For example, earlier we used

```
if ( isalpha(ch) )
```

to test if `ch` is an alphabetic character. If we wanted to test to see if `ch` is not an alphabetic character, we would use this fragment:

```
if ( !isalpha(ch) )
```

Because `isalpha(ch)` is true when `ch` is a letter, the expression `!isalpha(ch)` is *not* true when `ch` is a letter, hence true when `ch` is not a letter.

Let's do a Toolbox example. The Toolbox includes the `button()` function. This function returns "true" if the mouse button is being pushed down, and "false" otherwise.

From Mac's Toolbox: New Routines

Button

Returns "true" if the mouse button is down

Here is an example that performs an uncalibrated reaction time test:

```
/* reaction.c -- primitive reaction time test */
#define DELAY 50000
main()
{
    int i = DELAY;

    printf("When I say \"GO\", push the mouse button.\n");
    while ( i-- )
        ; /* time delay loop -- does nothing */
    if ( button() ) /* don't push button early */
        printf("Not yet...\n");
    i = DELAY;
    while (i-- )
        ; /* more delay */
    while ( !button() )
        printf("GO\n");
    printf("Done");
}
```

And here is a sample run:

```
When I say "GO", push the mouse button.
GO
GO
GO
GO
GO [MOUSE BUTTON]
Done
```

We had to put in a time delay loop; otherwise, the GO message would start printing before you had time to read the instructions. We used the `while(i--)` form we mentioned before; the loop cycles until `i` reaches 0. The loop syntax requires a statement for the loop to execute, so we used the "null statement", which does nothing and is represented by a lone semicolon. One side benefit of this program is that it shows you how fast the Macintosh can count to 50,000 twice.

Note that the final loop keeps printing GO as long as the mouse button is *not* pushed, thanks to the use of the NOT operator.

If you would like a more quantitative measure of your reaction time, you can modify the program to use the Toolbox function `tickcount()`. This function returns the elapsed time in "ticks" (a tick = 1/60th second) since the system was last started up. We leave this modification as a not too difficult exercise for you.

The AND Operator: && The AND operator (written `&&`) is used when we want two conditions to be true simultaneously. Suppose, for instance, that the sole qualifying standard for acceptance to Mediocre University is a combined SAT score in the range 600 to 1000. If we use the variable `sat` to represent a score, the standard amounts to two conditions: `sat >= 600` and `sat <= 1000`. Here's how to incorporate the two into one expression:

```
if ( sat >= 600 && sat <= 1000)
    printf("Welcome to Mediocre University.\n");
```

The if condition is true only if both subexpressions are true. The logical operators have a lower precedence than the relational operators, so the two relational expressions are evaluated first, and then the results are combined logically.

Note that you must use a logical operator to combine the two tests. You cannot borrow from mathematics and do the following:

```
if ( 600 <= sat <= 1000)    /* unsatisfactory */
```

Actually, this construction is valid C; it just doesn't mean what it appears to mean. The relational operators associate from left to right, so first the expression `600 <= sat` is evaluated. It has a value of 1 if true and 0 if false. Then this value of 1 or 0 is compared with 1000. Either is less than 1000, so the entire expression is always true, regardless of the value of `sat`.

The OR Operator: || Sometimes it's a matter of one or another condition sufficing. Suppose to get into Status University one needs a parental income of \$100,000 or else the ability to run 40 yards in full football gear in 5 seconds or less. With the OR operator (written `||`), we can combine the two conditions this way:

```
if ( income >= 100000 || time40 <= 5)
    printf("Welcome to Status University.\n");
```

Satisfying either subcondition (or both) causes the whole condition to be satisfied.

Logical Evaluations You can use several logical operators in the same expression. For instance, to look for lowercase vowels, you could do this:

```
if ( ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o'
    || ch == 'u' )
    vowels++;
```

Note how C's free format feature lets us spread a long expression over two lines.

An important feature of the C language is that it evaluates logical expressions from left to right, stopping as soon as it gets a definite result for the whole expression. In the preceding example, for instance, if `ch` is an 'e', the program won't bother going on to do the rest of the comparisons in the list.

The NOT operator has the highest precedence of the three, followed by AND, then by OR. Suppose, for instance, that Statusplus University incorporates Status University's standards plus an SAT requirement of 900 or better. We can express that condition this way:

```
if ( (income >= 100000 || time40 <= 5) && sat >= 900 )
    printf("Welcome to Statusplus University.\n");
```

The OR section is enclosed in parentheses; if we left them out, the meaning of the test would be changed. The higher priority of `&&` would link the time test to the SAT test. Thus, the new standard would be 1) income of \$100,000 or more, or 2) a time less than or equal to 5 seconds and a score of 900 or better. With the parentheses, the standard is 1) income of \$100,000 or more and a time less than or equal to 5 seconds, and 2) a score of 900 or better. If you can't keep the correct precedences in mind or else don't trust other program readers to do so, just use parentheses to make your groupings clear.

Other Jumps

One thing that loops and if statements have in common is that they use Boolean test conditions. We've just discussed those. A second point in common is that they involve "jumps." That is, the program, instead of going step by step through the code will skip or jump to another location in the code. In a loop, the program skips back to the beginning of the loop, and in an if statement, the program skips (possibly) to a particular alternative.

C has several other statements that involve jumps, but that do not use Boolean test conditions. We'll look them over now.

The goto Statement

The most freewheeling, the most abusable, and the least recommended jump statement is goto. Its use is simple, yet avoidable. Here is the general form:

```
goto label  
...  
label : statement
```

Here *label* is a user-selected name. When the keyword goto is encountered, program control jumps to the statement following the corresponding label. The colon after the label name identifies it as a label and not just some stray name.

Suppose, for instance, that you didn't know about C's loop statements. Then you could construct a loop along these lines:

```
i = 0;  
loopstart : printf("Counting: %d\n", i);  
            i++;  
            if ( i < 10 )  
                goto loopstart;
```

So long as *i* is less than 10, program flow keeps returning to the line labeled loopstart. Of course, a for loop can do the same task:

```
for ( i = 0; i < 10; i++)
    printf("Counting: %d\n", i);
```

The problem with the goto statement, as demonstrated through countless FORTRAN and BASIC programs, is that it can be used to create a maze-like program flow. Such "spaghetti" programs are very difficult to debug, maintain, and modify. By sticking to C's built-in structured statements, you help avoid those problems.

The continue and break Statements

The continue statement is a much more controlled jump. It is used inside a loop. When it is executed, it causes a program to skip to just after the last statement in the current cycle of the loop. Then the loop resumes. If it is a while or do...while loop, execution goes to the test condition. For a for loop, the "update" expression is executed first.

The continue statement is not that useful, for it generally can be avoided by rewording the program slightly. Here is an example using it:

```
for( i = -2; i <= 2; i++)
{
    if (i == 0)
        continue;
    printf("%d\n", i);
}
```

It prints the integers -2 to 2, skipping 0. But the following is simpler:

```
for ( i = -2; i <= 2; i++)
    if (i != 0)
        printf("%d\n", i);
```

The break statement, too, is used in loops. More commonly, it is used in the switch statement, which we come to next. Its effect is to break out of the loop (or switch) entirely. Here is an example:

```
for ( i = 0; i < 100; i++)
{
    printf("Enter the score:\n");
    scanf("%d", &score);
    if (score < 0 )
        break;
    sum += score;
}
```

This fragment keeps a running total of scores. It stops after 100 scores have been entered (the loop control) or else after a negative score is entered (the break statement). If we had used continue instead of break, then negative scores would have been skipped, and the loop would have continued until 100 nonnegative scores were entered.

The switch Statement

The switch statement can be an efficient alternative to a multiple if...else statement in certain cases. It allows a program to select from a list of choices. Thus, it means "switch" in the sense of a railway or electrical "switch" and not in the sense of "swap". Its form is this:

```
switch ( expression )
{
    case label1 : statement1
    case label2 : statement2
    ...
}
```

The *expression's* value should be one of the integer types (including char), and the *labels* are constants (or constant expressions) of the same type as the expression. We can think of the switch as offering a menu of choices, with the expression selecting one of them. Here is an example using a char variable called ch :

```
/* foolish.c -- questions the user */
main()
{
```

```

char ch;

printf("Enter your first initial:\n");
ch = getchar();
switch ( ch )
{
    case 'a' : printf("Are you an aardvark?\n");
               break;
    case 'b' : printf("Are you a baboon?\n");
               break;
    case 'c' : printf("Are you a capricorn?\n");
               break;
    default  : printf("Are you sure?\n");
}
}

```

If the user responds with, say, the character 'b', then the switch transfers the program flow to the line labeled case 'b'. Note the default entry. Program control jumps to here if ch doesn't match any of the other labels in the list. What happens if there is no default label and if there is no match? Then program control skips to the next statement following the switch statement.

What about all those breaks? The switch is just a fancy (but disciplined) goto, so once program flow jumps to the labeled statement, it would then pass on through the rest of the statements in the switch, executing each statement from the first one selected to the end of the switch. The subsequent labels would be ignored. By using a break, we ensure that *only* the statements associated with the label are executed. The break then causes program flow to jump out of the switch. If this is not clear to you, try removing the breaks from foolish.c and running the program that way.

More than one label can be attached to each case. Do you recall the long if condition we used to set up a vowel-counting fragment? We can accomplish the same end with this:

```

switch (ch)
{
    case 'a' :
    case 'e' :

```

```

    case 'i' :
    case 'o' :
    case 'u' : vowels++;
}

```

Here all 5 labels take the program to the same statement.

switch and if...else

Often either a switch or an if...else statement can be used to solve a programming problem. Our first switch example, for instance, could be replaced by this:

```

printf("Enter your first initial:\n");
ch = getchar();
if (ch == 'a')
    printf("Are you an aardvark?\n");
else if (ch == 'b')
    printf("Are you a baboon?\n");
else if (ch == 'c')
    printf("Are you a capricorn?\n");
else
    printf("Are you sure?\n");

```

Which is better? From the standpoint of computer efficiency, the switch statement is better than the if...else if several choices are involved. However, the if...else is much more flexible and can do things impossible for the switch. The key point here is that the switch statement is limited to comparing an expression to a integer constant for equality. That constitutes three limitations. First, you can't directly compare the expression to a variable. Thus,

```

if (ch = response1)

```

is valid, even if both `ch` and `response1` are variables. But `response1` cannot be used as a case label.

Second, a switch is not intended to deal with inequalities or complex logical expressions. Something like

```
if ( weight > 120 && weight < 135 )
```

has no reasonable switch equivalent. (If reasonability is not your guide, you could use 14 separate labels corresponding to the integers between 120 and 135.)

Third, a switch can't use floating-point numbers as labels, so it is not equipped to handle situations involving those numbers.

A Menu switch

One situation that is best handled with a switch is a menu-selection process. A menu offers a set of individual choices, so each choice can be made a case label for the switch. In a full-fledged Macintosh program, menu selections are usually handled using the mouse. We'll stick to the more primitive method of keyboard input, but the principles involved in using the switch are the same.

Our program will offer the user three choices of font size and then print a message in the selected size. If the user doesn't make one of the offered choices, he or she will be asked to choose again until one of the permissible choices is made. This is one case in which an exit-condition loop like the do...while loop is appropriate, for the loop must wait until after getting a response before judging whether or not to continue. Here is the program:

```
/* menu.c -- illustrates a simple keyboard-driven menu
*/
main()
{
    char response;      /* holds the user's reply */

    printf("Enter the letter marking your font choice:\n");
    printf("s      : small font\n");
    printf("m      : medium font\n");
    printf("l      : large font\n");
    do {                /* loop until valid response */
        response = getchar();
```

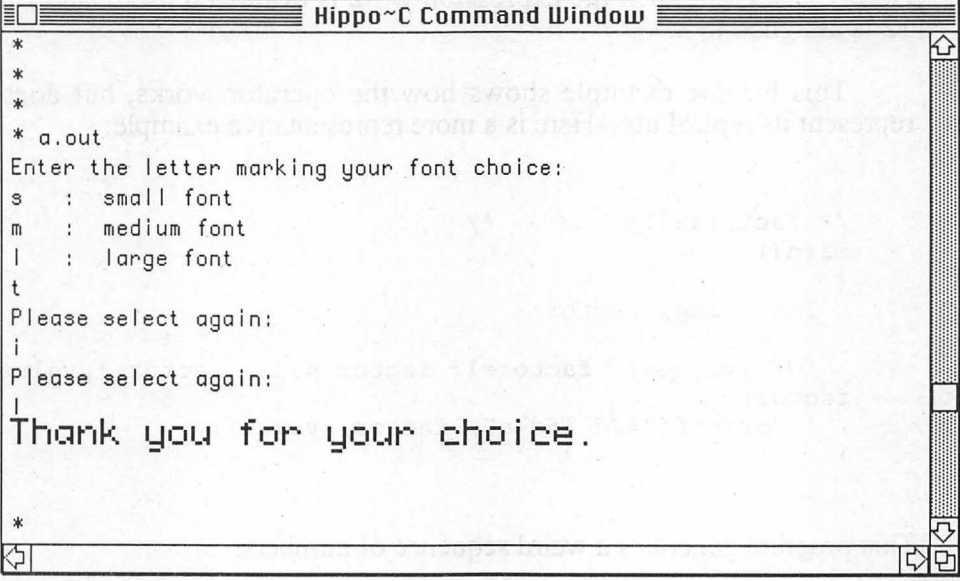


```

switch ( response )
{
    case 's' : textsize(9);
                break;
    case 'm' : textsize(12);
                break;
    case 'l' : textsize(18);
                break;
    default  : printf("\nPlease select again:\n");
}
} while ( response != 's' && response != 'm'
        && response != 'l' ) ; /* screen
choices */
printf("\nThank you for your choice.\n");
}

```

Note that the switch statement is contained inside the do...while loop. The loop continues until one of the three valid letters is chosen. The switch itself tends to the setting of the text size and to prompting for new input. Figure 4.3 shows some sample output.



```

*
*
*
* a.out
Enter the letter marking your font choice:
s : small font
m : medium font
l : large font
t
Please select again:
i
Please select again:
l
Thank you for your choice.
*

```

Figure 4.3 Output from menu.c

More Operators

Two more C operators fit into this chapter's themes. One, the comma operator, is often used in for statements to squeeze more information into the control section. The second is called the conditional operator and offers a compact alternative to some forms of if statements.

The Comma Operator: ,

The comma operator allows us to combine two expressions into one. When two expressions are separated by a comma, the lefthand expression is evaluated first, then the righthand expression is evaluated. The value of the entire expression is just the value of the righthand component. Consider, for instance, this statement:

```
x = (y = 6, y * 2);
```

Here y is set to 6, then the expression y * 2 is evaluated to 12, and finally 12 is assigned to x.

This bizzare example shows how the operator works, but does not represent its typical use. Here is a more representative example:

```
/* factorial.c          */
main()
{
    int value, factor;

    for (value=1, factor=1; factor < 10; factor++,value *= factor)
        printf("%2d %6d\n", factor, value);
}
```

This program generates a weird sequence of numbers:

1	1
2	2
3	6
4	24
5	120

6	720
7	5040
8	40320
9	362880

Actually this is a list of "factorials"; 3 factorial is the product of the integers 1 through 3, and so on. The program uses the comma operator twice. First, it lets the two expressions `value = 1` and `factor = 1` to be combined into one expression to fit into the `for` format. This combined expression then serves to initialize two separate values.

The update portion of the control statement is interesting. First, `factor` is increased by 1, then `value` is multiplied by the new value of `factor`. Here the comma operator serves two purposes. First, it lets us squeeze two operations into the update section. Second, it ensures that `factor` is updated *before* `value` is. Because of that, we can use either the prefix or postfix form of the increment operator.

If we desire further conciseness, we can replace the update portion with this single expression:

```
value *= ++factor
```

Here we use the prefix form of the increment operator to ensure that `factor` is incremented before any other operations are performed.

The Conditional Operator: ?:

The conditional operator is a "trinary" operator, one with three operands. An expression using the conditional operator has this form:

condition ? expression1 : expression2

If the *condition* is true, the value of the entire expression is set equal to the value of *expression1*. Otherwise the entire expression is set to the value of *expression2*.

Here is an example that assigns the larger of two values to a variable called bigger:

```
bigger = x > y ? x : y;
```

If x is greater than y, then the entire expression has x's value, which gets assigned to bigger. Otherwise y's value is assigned.

Summary

C offers three statements for looping, or running through a segment of program repetitively. The **while** and **for** statements set up entry-condition loops, in which a test condition is checked before cycling through a loop. The less-used **do...while** loop is an exit-condition loop, in which the test condition is checked at the end of each cycle. The for and while loops are sufficiently flexible to be used interchangeably; however, the for loop is usually used when the number of iterations is determined before the loop starts, while a while loop is preferred when the number of iterations is not known in advance, as in a letter-counting program.

For choosing alternatives, C offers the **if** and **if...else** statements, the **switch** statement, and the *conditional* operator. The if family is the most versatile, allowing a program to make decisions on the bases of relational and logical expressions of arbitrary complexity. The switch is useful for selecting from a labeled list of choices. The conditional operator is useful for assigning a variable one of two values.

Most of these forms use test conditions. These conditions may be any sort of expression. A zero value for the expression is treated as "false", and a nonzero value as "true". Most commonly, the expressions use relational operators, which compare magnitudes of quantities, and logical operators, which serve to combine or modify relational expressions.

5

Functions

In this chapter you will learn about:

- Function arguments
 - Function return values
 - Function types
 - Type conversions
 - Pointers as arguments
 - Pointer arithmetic
 - Scope: local and global variables
 - Recursion
 - Preprocessor macros
-

We have seen how vital functions are to C programming. So far we have concentrated on using existing functions, but now we are ready to investigate more deeply the writing of functions. We will look into how a function makes use of its arguments. Also, we will investigate the "scope" of a variable, that is, the question of to which functions a variable is known. Finally, we will look at macros, the preprocessor cousins to functions.

First, let's review what we have learned so far.

Review

Functions are the modular programming units of C. Every program must contain a function called **main()**, which then becomes the first function executed in the program. A function calls or invokes another function by using its name. Information is communicated to a function through its argument list, which is a list of one or more expressions separated by commas and enclosed in parentheses following the function name. Communication is by value, meaning that each argument expression is evaluated and then that value is sent on to the function. In turn, a function can "return" a value to the calling function. Or, through using the address of a variable as an argument, a function can modify the value of that variable.

The simplest way to incorporate a user-defined function into a program is to include the function definition in the same file as main(). Here is a simple example illustrating that technique:

```
main()          /* defining the main() function */
{
    printf("This message is from main(): hello.\n");
    comment();
}

comment()       /* defining the comment function */
{
    printf("That main() is sure a dull function!\n");
}
```

Running it, of course, produces this output:

```
This message is from main(): hello.
That main() is sure a dull function!
```

Notice how the definition of the comment() function parallels that of main(). First comes the function name, followed by a parentheses pair. This constitutes the "head" of the function. Note that no semicolon follows the function name when it is used as a heading for a function definition. Next comes a section enclosed between braces. This constitutes the "body" of the function.

An important point is that all C functions are defined on the same level. This contrasts to Pascal usage, in which functions and procedures are defined inside a program block or inside other functions and procedures. In C, you cannot define one function inside the body of another.

This sample used neither a return value nor an argument. Let's go on to examples that do use them.

Arguments

How does a function make use of an argument passed to it? The easiest way to see the answer is to look at an example. To highlight the mechanics, let's use a very simple one, a function called **twice()** that returns twice its

argument. It is easy enough to foresee how to use the function. For instance, if we need to find what twice 5 is (perhaps we have all misplaced our pocket calculators), we would use a call like this:

```
y = twice(5);
```

Here 5 is the argument passed to the twice() function. But how do we write the twice() function so that it will accept and use the 5? Here's one way:

```
/* function that doubles a number */
twice(x)    /* x is the "formal" argument */
int x;      /* and is of type int */
{
    x = 2 * x;
    return x;
}
```

Now the parentheses following the function name are no longer empty. They contain one variable name (x, in this case) to indicate that this function takes one argument. Then the next line declares that the argument should be of type int. Note that this declaration occurs in the head section of the function definition, *before* the brace that marks the start of the body of the function.

C allows you to omit the declaration statement *if* the argument is of type int, but we will declare all arguments.

What happens, then, when we make a function call like this?

```
y = twice (5);
```

When twice() is invoked, the function head tells the computer to assign storage for an int variable called x. Then x is assigned the value of the argument (here 5) used in the function call. Thus the function call serves to create the variable x and to initialize it to the value provided by the function call.

We have used the term "argument" two different ways. First, we've used it to mean the *value* used in the function call. Second, we've used it to

mean the *variable* used in the function definition. To get around this ambiguity, C calls the value in the function invocation the "actual argument", while the variable used in the function definition is called the "formal argument". So we can say that a function call results in the actual argument value being assigned to the formal argument. (Or, if we feel the need for variety, we can use the word "parameter" instead of argument.) One extremely important point is that the formal arguments for a function are "local" variables. That means they are known and used only by the function in which they are defined. If we called `twice()` from a `main()` function that also had an `x` variable, the two `x`'s would be independent of one another. Each would have its own storage location. In fact, all the variables we have used so far are local. This helps compartmentalize the functions. If you have a large program and are writing a new function, you don't have to worry about whether or not you've used a variable name in a previous part of the program. If you call a variable `x` in a function, it will not be confused with an `x` anywhere else.

Let's put together and run a complete program to illustrate some of the points we have made:

```
main()
{
    int x,y;

    x = 10;
    y = twice ( x + 2);      /* function call */
    printf("x = %d, y = %d\n", x, y);
}

twice(x)                    /* function definition */
int x;
{
    x = 2 * x;
    return x;
}
```

Here is the program output:

```
x is 10, y is 24
```

Let's run through some of the points illustrated by this example. The actual argument is `x + 2`, or 12. This value gets assigned to the formal argument,

x of twice(). This x is distinct from the x of main(), just as Salem, Oregon is distinct from Salem, Massachusetts. In twice(), x gets doubled, and that value (24) is returned to main() and assigned to y. When the results are printed, we see that x in main() is still 10, totally unaffected by what happened to the x in twice().

Multiple Arguments

Suppose we need more than one argument. Then we must list however many arguments are needed and declare their types. Here is an example that takes two arguments: a character and an integer. The function then prints the character the indicated number of times and starts a new line:

```
/* prints a character c n times */
chartimes( c, n)    /* provide an argument list */
char c;
int n;              /* declare them in the order listed */
{
    int count;      /* a local variable */

    for (count = 0; count < n; count++)
        putchar(c);
    putchar('\n');
}
```

Note that the arguments are declared in the same order as they appear in the formal argument list. This program uses a variable (count) in addition to the formal arguments. Any such additional variables are declared *inside* the body of the function. Only the arguments are declared before the opening brace. Like the arguments, count is a local variable, private to the chartimes() function.

When the function is called, the actual arguments must come in the same order as the formal arguments. Don't try making a call like chartimes(10, "T") and expect the function to sort out what you mean.

Let's look at a short example using this function. It also serves to refresh our memories about for loops and the comma operator:

```
main()
{
    int i;
```

```

    char ch;
    for ( i = 1, ch = 'a'; i <=5; i++, ch +=2)
        chartimes(ch,i);
}

/* prints a character c n times */
chartimes( c, n)
char c;
int n;
{
    int count;

    for (count = 0; count < n; count++)
        putchar(c);
    putchar('\n');
}

```

Here is the output:

```

a
cc
eee
gggg
iiii

```

Look at the for loop in main(). It uses the comma operator to initialize both i and ch. The third expression in the control section also uses a comma operator to allow both i and ch to be updated. Recall that += is the additive assignment operator; in this case it adds 2 to ch. But how can you add integers and characters? In C, easily. What happens here is that 2 is added to the ASCII code for ch, so the new ch is the character whose code number is bigger by 2. Since the letters come alphabetically in the ASCII sequence, the program just moves two letters down the alphabet. Thus our example prints every other letter until it reaches the limit for i.

Adding 2 to a is an example of mixing types in an expression. C provides a set of rules governing mixed type operations, and we will discuss them later this chapter.

Return Values

We've looked at the flow of information into a function, now let's look at the flow out. The standard channel for communicating back to the calling program is to use the return facility, as we did in the `twice()` example. This works a bit like arguments in reverse. Just as only the value of an argument is transmitted to a called function, so is only the value of the return expression made available to the calling function. For instance, suppose the `dork()` function looks like this:

```
dork( x, y )
int x, y;
{
    return (3*x + y*y*(5*x -y) );
}
```

It takes two arguments, plugs their values into the expression, and boils the expression down to a single number. A call like

```
doug = dork(a,b);
```

causes this return value to be assigned to `doug`.

A function can have at most *one* return value. If more than one value is required, we need to use pointer arguments; that, too, we will cover later in the chapter.

Having only one return value, however, doesn't mean that a function can't use return more than once. The function will terminate as soon as it reaches a return. Here, for example, is one way to code an absolute value function:

```
abs(x)
int x;
{
    if x < 0
        return (-x);
    return (x);
}
```

If *x* is less than zero, the first return is executed, and the function terminates. Otherwise, the first return is skipped and the second is executed. (The parentheses are optional.) However, it is considered better programming practice to have just one return right at the end of the function. We could, for example, use an intermediate variable to store various choices:

```
abs(x)
int x;
{
    int y;

    if x < 0
        y = -x;
    else
        y = x;
    return y;
}
```

In this particular case, of course, we could condense the function by using the conditional operator:

```
abs(x)
int x;
{
    return x < 0 ? (-x) : x;
}
```

If this seems obscure, you may wish to reread the paragraphs in Chapter 4 on the conditional operator.

Return Values and Function Types

Our examples so far have returned an integer value, type *int*, to be precise. C functions, however, can return types *char*, *short*, *float*, and any other single-valued type. To return these other types, we must properly define the function and properly prepare the calling function. Let's look at these two steps in turn.

Defining a Function Type

Defining a function type is simple; we just precede the function name with the desired type. For example, suppose we want a function that returns type `char`. Here is one that takes an integer argument and converts it to an uppercase letter:

```
char inttochar(n)    /* declare function type */
int n;              /* declare argument type */
{
    return 'A' + n % 26;
}
```

Note that the function type and argument type need not match. The function type refers to the type *returned*, not to the argument type(s). This example uses the modulus operator (%) to convert `n` to an integer in the range 0 to 25. Thus the return value is always in the range A to Z.

Why didn't we have to declare types for the other functions we used? Actually, the best programming practice would have been to explicitly declare the functions type `int`. However, C tolerates a bit of sloppiness and assumes that any function of undeclared type is type `int`. Since many functions are indeed of that type, this scheme is a timesaving one.

Using a Non-int Function

An additional complication arises when we go to use a non-int function. We must declare the function type in the calling function, just as we declare variable types. Here, for example, is a program designed to try out the `inttochar()` function.

```
main()
{
    int i;
    char ch;
    char inttochar(); /* declare function type */
    while ( scanf("%d", &i) == 1 )
    {
        ch = inttochar(i);
        printf("%c\n", ch);
    }
}
```

```

char inttochar(n)
int n;
{
    return 'A' + n % 26;
}

```

Note that when we declare the `inttochar()` function in `main()`, we omit the argument list. All that this declaration describes is the value returned. Only the formal definition of the function includes the formal argument list.

Here is a sample run:

```

2 [RETURN]
C
26 [RETURN]
A
234567 [RETURN]
V
[OPTION] -d

```

We hit the `[RETURN]` key to let `scanf()` know we were finished typing in a number.

What happens if you forget to declare the function type in the calling function? The compiler will assume that any undeclared function is type `int`. Then, when the compiler discovers the same function to be declared, say, type `char` in the function definition, it decides you've made an error (you have) and halts compilation.

Type void

A function can be of any type that corresponds to a single value. This includes all the basic types we've discussed and many of the derived types we will introduce later. Some functions, however, do not even return one value. They just perform some action and quit. An example would be the `lineto()` function we used in some of the graphics examples. Historically, such functions have been taken to be the default type, `int`. More recently, however, a new type has been introduced for just such functions. The type identifier is the word `void`; it implies the function has no return value.

Suppose we want a function that draws a box shape. The Macintosh Toolbox has such a function, but it uses structures, a topic we haven't come to yet. So let's write our own. Because this function (call it `box()`) performs an action rather than returning a value, we can make it type `void`. Here it is, contained in a program to test it:

```
/* boxes.c -- uses box() to draw several boxes */
main()
{
    int ulx,uly,lrx,lry; /* upper-left and lower-right x
                        and y */
    void box();          /* declare any non-int
                        function */

    for (ulx=40,uly=80;ulx < 256 && uly < 170; ulx += 10,
        uly +=3 )
    {
        lrx = 512 - ulx; /* set lower-right x */
        lry = 300 - uly;
        box(ulx,uly,lrx,lry); /* draw a box */
    }

    void box( x1, y1, x2, y2)
    int x1, y1, x2, y2;
    {
        moveto(x1,y1);
        lineto(x2,y1);
        lineto(x2,y2);
        lineto(x1,y2);
        lineto(x1,y1);
    }
}
```

Notice that the loop uses the comma operator to process the `x` and `y` coordinates in parallel. Figure 5.1 shows the output for this program. The Hippo C Command Window was expanded to nearly full screen before the program was run.

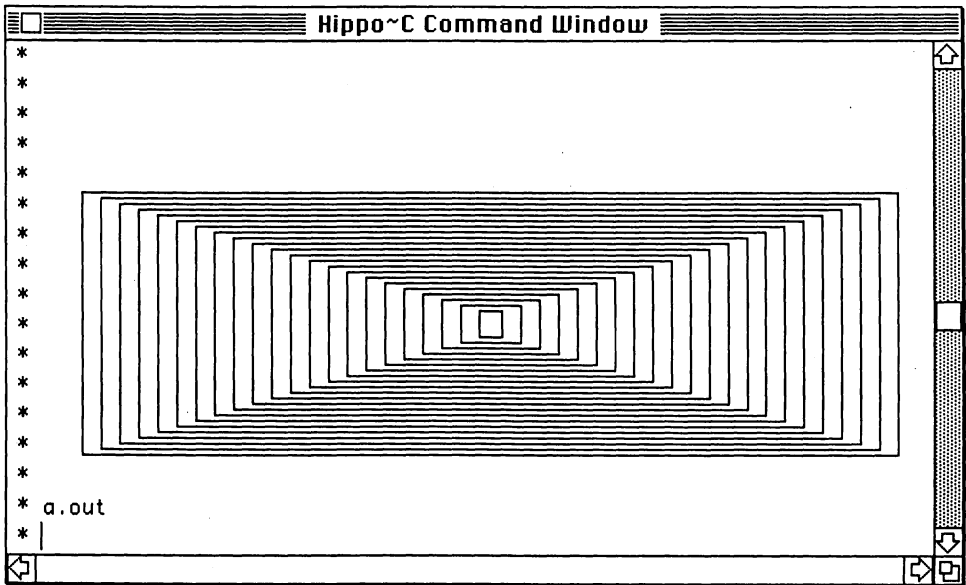


Figure 5.1 **Output of boxes.c**

The void type is unusual because it can be applied only to functions and not to variables. Also, the type need not be used. We could have left the type for `box()` undeclared, both in the calling program and in the function heading. Then the compiler would assume `box()` was type `int`, and the program would work. This, historically, is the way such functions were handled. The main reason for using the void declaration is for logical consistency.

The Macintosh Toolbox and Function Types

The Macintosh Toolbox routines were written as Pascal functions and procedures. Pascal functions correspond to C functions with return values. In particular, a Pascal function returns a single value, and the function is characterized by the type of its return value. Thus, it is a straightforward matter to set up the function type for a C call to the Toolbox: Pascal functions that return, say, a `char` value are a `char`-type function for a C call. There are minor differences. For example, the `button()` function we used in Chapter 4 is a Boolean function in Pascal, returning `true` or `false`. But since these values are represented internally by nonzero and zero values respectively, we can make the C representation type `int`.

Pascal procedures differ from Pascal functions in that the procedures do not have a return value. Thus, they correspond conceptually to a type void C function. Logically, then, Toolbox procedures could be represented by void functions. Unfortunately, this would require that all these procedures be declared void explicitly in each program using them. Hippo C avoids that problem by representing procedures implicitly as type int functions. Because that is the default type assumed for functions, Toolbox procedures can be used without declaring them.

Type Conversions and Type Casts

Now that we have revived the topic of types, let's look into type conversion. Although C is a typed language, it is not strongly typed. Unlike, say, Pascal, C allows you to mix types in expressions. We've already mixed int and char types in expressions a few times. C has a systematic set of rules for the process.

Integer Conversions

When you have an expression with more than one integer type, the values are converted to largest type present, then, if there is an assignment operator, the result is forced to fit the type for receiving variable. Generally, this causes no problems going from smaller to larger types but can be troublesome going the other direction. Let's look at some examples.

Suppose we have this expression, where ch is type char:

```
ch = 'A' + 3;
```

The character 'A', which is stored as an 8-bit number, is converted to type int. This means, for Hippo C, that it is copied into a 32-bit location. Then it is combined with 3, which is already stored in a 32-bit location. The result is a 32-bit integer. Since the answer is to be assigned to a char type, the 32-bit number is truncated to 8 bits, and the 8-bit number is placed in ch. The truncation discards the "high-order" bits, that is, the ones used to express larger numbers. For this example, they were all zeros anyway, so we had no problems. Adding 2000 instead of 2, however, would have resulted in an answer larger than 8 bits. In that case, some of the bits would be lost during truncation, changing the value of the number.

According to the C standard, char and short types are always converted to int when evaluating expressions, even if no larger type is present. Some compilers, however, don't follow that particular standard.

Floating-Point Types

Type float numbers are converted to double in expressions. This ensures maximum precision for the calculations. (It also ensures maximum time of calculation, and there is some talk of relaxing this feature.) If the result is assigned to a float variable, it is rounded down to the proper precision.

When float or double types are mixed with integer types, everything is converted to double. First, consider pure integer calculations:

```
int intvar;  
float flvar;  
  
intvar = 11 / 5;  
flvar = 11 / 5;
```

The rules for integer division yield 2 as the result. This integer is assigned to intvar. The float variable flvar, however, is assigned the value 2.0. That is, the integer 2 is converted to its floating-point equivalent.

Now suppose we replace 11 with 11.0:

```
int intvar;  
float flvar;  
  
intvar = 11.0 / 5;  
flvar = 11.0 / 5;
```

This time the expression 11.0 / 5 is evaluated as if both numbers were floating-point. Thus the value of the expression is 2.2, which is stored as a double value. In the assignment for intvar, the value is converted to the integer 2. For the assignment to flvar, the double value is converted to float, meaning fewer bytes are used.

Conversion from integer types to floating point causes no problems, but conversion from floating point to integer can produce results too large for successful conversion. Another point to keep in mind is that decimal fractions are truncated when converted to an integer type. This means they are always rounded down. Thus 1.9 would convert to 1, not 2.

Type Casts

Sometimes it is necessary to be explicit about what type conversions you want. Perhaps the automatic conversion rules don't meet your needs, or the compiler doesn't exactly follow the standard, or a system function may return the wrong type for your needs. Then you can use the type cast. This is accomplished by enclosing the type name in parentheses and preceding the value to be converted with this combination. For instance, to explicitly convert a char value to int, you can use the expression

```
(int) 'K'
```

This would result in the ASCII code for 'K' being stored in a 32-bit location.

First, let's look at an example that uses automatic conversion. Let fries be an int variable. Then the statement

```
fries = 12.7 + 13.6;
```

results in fries being assigned the value 26. First, the two floating-point numbers are added to get 26.3, then the result is truncated to 26.

Now let's use type casts to force the conversions to take place before addition:

```
fries = (int) 12.7 + (int) 13.6;
```

In this case fries is assigned 25. First, 12.7 is truncated to 12, then 13.6 is truncated to 13, producing a sum of 25.

Eventually, we'll have to use type casts for certain function return values.

Now let's move on to another topic.

Using Addresses and Pointers

Sometimes we want a function to alter the values of variables in the calling program. As we have mentioned many a time, the arguments to a function are new variables, and altering them has no effect on the original variables. The C solution, as we have seen in using `scanf()`, is to provide the *address* of the variables to a function. The function, knowing the location of the variable, can then fiddle with its contents. To use this technique, we need to know how to obtain the address of a variable, and we need to know how to represent that address as a formal argument. In C this is done using **pointers**, so let's take an excursion into pointer land. We'll start by looking at addresses.

Obtaining Addresses: the & Operator

If `x` is a variable, then `&x` is the address of that variable. The `&` is the address operator. Typically, such addresses are used as function arguments, but you can use the operator to print out addresses. Use the `%u` (unsigned integer) format. Here is an example:

```
main()
{
    int x = 5;

    printf("x is %d and is stored at %u.\n", x, &x);
}
```

The output is this:

```
x is 5 and is stored at 106820.
```

In C, addresses are called pointers. We say that `&x` points to the variable `x`. The address operator yields a pointer constant, for even if we change the value of `x`, the location where `x` is stored (`&x`) stays unchanged.

Pointer Variables

C also has pointer variables. These are variables which can be assigned addresses as values. For example, if `pi` is a proper pointer variable (we'll discuss what "proper" means soon), we can make statements like this:

```
pi = &x;    /* assign an address to a pointer variable */
```

If this is the `x` of the preceding value, then `pi` is assigned the value 106820, the address of `x`. The difference between `&x` and `pi` is that the first is a constant and the second is a variable. We can't assign values to `&x` any more than we can assign values to other constants. We can assign values to `pi`, and we can change the value of `pi`.

How do we declare a pointer variable? An address is an integer, so it would seem that an integer form, such as unsigned long might be suitable. But if a program works with an address, it needs to know more than the numerical value of the address. It also has to know the data type of the contents. After all, the address of a `char` looks just like the address of an `int` or `float`; it is just the numerical value of the first byte holding a data item. Only by knowing the data type will the computer know how many bytes to use and how to interpret them. For these reasons, there are a variety of pointer types. A pointer that points to a type `int` value is called a "pointer-to-`int`". Similarly, a "pointer-to-`char`" points to a `char` type, and so on.

To declare a pointer, then, we need to show that the variable is a pointer, and we have to indicate the type it points to. Here is how it is done in C:

```
char *pc;    /* pc is a pointer to type char */  
int *pi;     /* pi is a pointer to type int */
```

The asterisk is used to indicate that `pc` and `pi` are pointers, and the usual C type names indicate what each can point to. Thus, `pc` can be assigned the addresses of type `char` values, and `pi` can be assigned the addresses of type `int` values.

The asterisk does more than indicate that `pc` and `pi` are pointers; it can be used to access the values contained in the pointed-to addresses. Let's look at that next.

*The Unary * Operator:*

What is `*pi`? It represents the *value* stored at the pointed-to location. That is, if `pi` points to `x`, then we have the following equalities:

```
pi == &x      /* pi equals the address of x */
*pi == x      /* *pi equals the value of x */
```

This "indirect value" operator is a unary operator (one operand), and the compiler uses the context to distinguish it from the multiplication operator.

The operand for `*` should be a pointer (an address). The expression `*pi` means "go to the indicated address and use the value there." Here is a program that clarifies the meaning and use of the `*` operator:

```
main()
{
    int faces = 2;
    int heads = 10;
    int *pi;          /* pi is a pointer-to-int */
    pi = &faces;      /* assign faces's address to pi */
    printf("pi = %u; *pi = %d\n", pi, *pi);
    pi = &heads;      /* now have pi point to heads */
    printf("pi = %u; *pi = %d\n", pi, *pi);
    *pi = 30;         /* change heads */
    faces = *pi * 2;
    printf("pi = %u; *pi = %d\n", pi, *pi);
    printf("heads = %d; tails = %d\n", heads, faces);
}
```

Here's the output:

```
pi = 106872; *pi = 2
pi = 106868; *pi = 10
pi = 106868; *pi = 30
heads = 30; faces = 60
```

Let's go through the program step by step. First, the declarations establish `heads` and `faces` as type `int` and `pi` as a pointer-to-`int`. Next, the address of `faces` is assigned to `pi`. Both `&faces` and `pi` are type pointer-to-`int`, since `&faces` is the address of an `int`, and an address is a pointer. The first `printf()` statement reveals that the address of `faces` is 106872 and that the value stored at that address (`*pi`) is 2. Figure 5.2 illustrates these relationships.

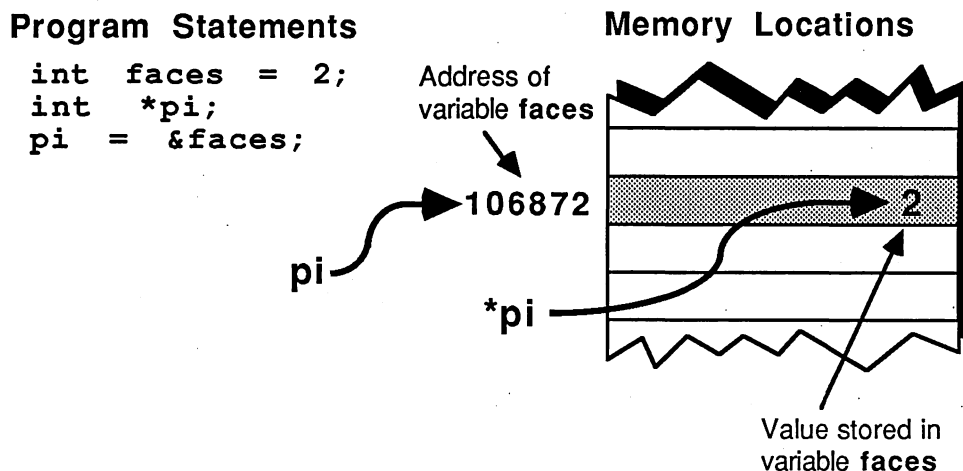


Figure 5.2 Pointers and Indirect Values

Next, the address of `heads` is assigned to `pi`. As the associated `printf()` statement shows, `pi` is the new address, and `*pi` is the value stored at that address.

Then we have the line

```
*pi = 30;
```

This means, "go to the location pointed to by pi and change the value there to 30". As the subsequent printf() statements show, this leaves the value of pi unchanged; it still points to heads. But it does change the contents of the pointed-to location. Now *pi and heads are 30. In other words, if pi points to heads, then

```
*pi = 30;
```

has the same effect as saying

```
heads = 30;
```

More generally, if pi points to a variable, then *pi can be used as a synonym for the variable itself. The last operation in the program illustrates the same point, for

```
faces = *pi * 2;
```

has the same effect as

```
faces = heads * 2;
```

This is such an important point, we'll repeat it one more time. If pi is a pointer variable assigned the address of the variable x, then *pi can be used as a synonym for x itself.

The way we have used pointers so far makes this fact an interesting novelty. Sure, we can use *pi instead of heads or faces, but we don't have to. We could more easily use heads or faces directly. But what if we have a function call in a program using these variables? Because these variables are local to the calling program, the called program won't be able to use heads or faces directly. But if we pass, say, the address of heads and assign it to a pointer in the called program, then the called program can use the * operator to alter heads. We take up this approach next.

Pointer Arguments

Suppose we want a function whose effect is to double a variable. We could do that using a return value, but let's try doing it using a pointer instead. First, let's construct a calling program:

```
main()
{
    int x = 5;
    void double_it(); /* our new function to be */
    printf("x is %d and is stored at %u.\n", x, &x);
    double_it(&x);
    printf("x is %d and is stored at %u.\n", x, &x);
}
```

The `double_it` function takes one argument, the address of an integer. It will use this address to modify the contents of the `x` variable in the calling program. How should we set `double_it()` up? It should have one formal argument, a variable that can hold the address of a type `int` value; that would be a pointer-to-`int`. Thus the head of our function definition should look like this:

```
void double_it( p )
int *p;        /* argument p is a pointer-to-int */
```

This informs the compiler that the function expects one argument, the address of an `int` value. Note that `p`, and not `*p`, is the pointer.

What about the body of the program? The program is supposed to double `x`. The function call

```
double_it(&x);
```

assigns the address `&x` to the pointer variable `p`. As we just discussed, this makes `*p` synonymous with `x`, so whatever we want done to `x` we can do to `*p` instead. So to double `x`, we can use the multiplicative assignment operator and say:

```
*p *= 2;    /* double whatever p points to */
```

Now we can finish our program and test it:

```
main()
{
    int x = 5;
    void double_it();

    printf("x is %d and is stored at %u.\n", x, &x);
    double_it(&x);
    printf("x is %d and is stored at %u.\n", x, &x);
}

/* a function using a pointer */
void double_it( p )
int *p;
{
    *p *= 2;  /* double what's stored at p */
}
```

Running the program, we get this:

```
x is 5 and is stored at 106820.
x is 10 and is stored at 106820.
```

Pointers actually do work, especially when used correctly. A common error is to pass an ordinary variable instead of an address to a function designed to work with pointers. For example, when wishing to read in a value of the integer `x`, we might use `scanf("%d", x)` instead of the proper `scanf("%d", &x)`. What happens if we do the former? Whatever value `x` has (and it may be garbage if `x` hasn't been assigned a value yet) is interpreted to be an address. That address may correspond to, say, part of your program code, but most likely it will lie outside the program boundaries. In either case, it spells death to your program.

A Two-Pointer Example

We could have used a return value instead of the pointer approach for the last example. Let's look at an example for which a return value will not suffice.

A common manipulation in programming is exchanging values between two variables. Since two variables are affected, a simple return value from a function does not suffice. But we can use a pointer to each variable to effect the switch:

```
/* a function that exchanges two int values */
void exchange(a,b)
int *a, *b;
{
    int temp;    /* temporary storage */

    temp = *a;   /* save value pointed to by a */
    *a = *b;     /* move *b to *a */
    *b = temp;   /* move former *a to *b */
}
```

Let's go over a few points. First, note that the program works with **a* and **b*, not with *a* and *b*. This is because we wish to change the values of the pointed-to variables; using *a* and *b* would mean changing the addresses that *a* and *b* point to, a meaningless operation in this context.

Second, note that we need a temporary variable. The problem is similar to having a glass of water and a glass of milk. If we wish to exchange the contents, we need a third, empty glass in which to place the water before we pour milk into the water glass. Here, *temp* is that glass. Note that we use *temp*, not **temp*. The reason is that *temp* is not a pointer; it already is of type *int*.

Finally, how do we use the function? Suppose we have *int* variables *x* and *y*; what is the proper function call? If your answer is

```
exchange (x, y) ;
```

you need a little more practice. Remember, the arguments to `exchange()` should be pointers. Thus, the correct call is this:

```
exchange (&x, &y);
```

Pointer Types

On the Macintosh all addresses, whether of `int`, `char`, `float`, or any other type of value, are stored in a 32-bit word. Why, then, do we need to distinguish between different types of pointers?

One reason is that the address is the address of the *first* byte of the value. Thus the address by itself does not tell the compiler how many bytes the value takes up. But that information is needed for the `*` operator. By declaring `p1` to be a pointer-to-`char`, we inform the compiler that `*p1` refers to a single byte of memory. If, however, we declare `p2` to be a pointer-to-`int`, we tell the compiler that `*p2` refers to four bytes of memory.

A second reason is that knowledge of the type pointed to is needed for "pointer arithmetic." Let's look at that topic now.

Pointer Arithmetic

C allows you to add and subtract values from pointers and to take the difference between two pointers; these operations constitute pointer arithmetic.

Since a pointer is a memory address, it has a numerical value. But adding 1 to a pointer is not necessarily the same as adding 1 to the numerical address. Let's look at a short program to see what actually happens:

```
/* pointer.c -- print results of pointer arithmetic */
main()
{
    char ch, *pc;      /* declare various variables */
    short stop, *ps; /* and pointers */
    int oto, *pi;
```

```

    pc = &ch;          /* assign values to pointers */
    ps = &stop;
    pi = &oto;
    printf("pc is %u and pc + 1 is %u\n", pc, pc + 1);
    printf("ps is %u and ps + 1 is %u\n", ps, ps + 1);
    printf("pi is %u and pi + 1 is %u\n", pi, pi + 1);
    printf("--pi is %u\n", --pi);
}

```

We've set up three variables, then assigned the addresses of these variables to pointers of the proper type. Here is the output:

```

pc is 106823 and pc + 1 is 106824
ps is 106816 and ps + 1 is 106818
pi is 106808 and pc + 1 is 106812
--pi is 106804

```

See how clever C is! Adding 1 to a pointer changes the address by the size of the pointed-to type. For instance, adding 1 to a pointer-to-short changes the address value by 2, since a short occupies two bytes. This is very handy when you have an array of values. An array is a bunch of adjacent memory units all of the same type, so adding 1 to a pointer moves it to the next member of the array. We'll get into that topic in Chapter 6. In the meantime, note that if the program doesn't know what type a pointer points to, it won't know how much to add to the address when 1 is added to the pointer.

Another point to know is that in Hippo C, the difference between two pointers is type int. Two pointers must point to the same type if one is to be subtracted from the other. The result is the actual address difference divided by the number of bytes in the pointed-to type.

Finally, be aware that when you add a number to a pointer, there is no guarantee, other than your programming, that the new address actually points to an object of the same type you started with. For example, in the preceding program, after applying the decrement operator, the value of pi is 106804. We could then say this:

```

pi += 3;

```

This would add 3 int-sized chunks, or 12 bytes, to pi, making its value 106816, which is the address of the short variable stop. Or you could subtract 18000 from pi and have it point Mac-knows-where. The moral is that it is your responsibility to see that pointer values make sense.

Now let's turn from pointers to a discussion of different storage schemes for variables.

Scope: Local and Global Variables

So far we have discussed arguments, return values, and pointers as means for interfunction communication. There is one more approach, the use of "global" variables. A global variable is one that is shared by more than one function. Thus, when it is changed by one function, it is changed for all the functions that share that variable.

If this is possible, why bother with arguments? Just make all the variables global, and all functions can access whatever variables they need. This approach may sound attractive, but it usually leads to all sorts of problems in larger programs. The programmer forgets which variable is what, or uses the same name for two different uses, or devises a subroutine that inadvertently changes the value of an innocent variable, and so on. Experience has shown conclusively that it is much better to use local variables whenever possible. That is why C uses local variables by default. Yet sometimes a program will need a shared set of data of the sort global variables can provide, so C does allow for their use. In fact, C has several different storage classes for variables. The storage class determines the scope of a variable (which functions know of it) and the persistence of a variable.

The four storage classes are **automatic** (local) variables, **external** (global) variables, **static** (persistent) variables, and **external static** (global to a single file) variables. The exact storage class is determined by where a variable is declared and by the use of certain keywords. We'll look at the four classes in turn.

Automatic Variables

All the variables we've used so far have been automatic variables. This means that they are local to the function in which they are defined. It also means that storage for the variable is allocated each time the containing function is called and released each time the function terminates. Thus, if a

function is called twice in a program, the automatic variables it uses are undefined between calls. A function using only automatic variables has no memory of what happened the preceding call.

What qualifies a variable to be automatic? It is automatic by default if it is a function argument or if it is defined within the body of a function. We can explicitly declare such a variable to be automatic by using the keyword `auto` in the variable definition:

```
auto int goo;
```

Since it would be an automatic variable by default anyway, the main use of this keyword is documentation. The need for this will be clearer after we discuss the external storage class.

C does provide for variables whose scope is even more restricted than a single function. Up to now, we have declared variables after the opening brace of a function. It is legal, however, to declare variables at the beginning of any statement block. (A statement block, recall, is a group of statements enclosed within braces.) The scope of such a variable is restricted to the block in which it is declared. We won't use this feature.

The Register Storage Class The register variable is a special case of automatic variable. The difference is that a regular automatic variable is stored in the computer memory, while a register variable is kept in a register in the Central Processing Unit. This speeds up processing the variable. To request a register variable called `quickx`, do this:

```
register int quickx;
```

As indicated, this is just a request. There are a limited number of registers, so none may be available. In that case, the variable is made into a regular automatic variable. We won't use this feature, but we mention it so you won't be shocked if you see it somewhere.

The External Storage Class

To create a variable that can be shared by several functions, place the variable declaration outside of, or external to, all functions. All functions in the same file following the declaration will have access to that variable. Here is an example:

```
int x = 27;      /* an external variable */

main()
{
    printf("main() knows that x is %d.\n", x);
    fap();
}

fap()
{
    printf("fap(), too, knows that x is %d\n.", x);
}
```

And the output:

```
main() knows that x is 27.
fap(), too, knows that x is 27.
```

If we had placed the declaration between `main()` and `fap()`, only `fap()` would know of `x`, for the compiler does not look ahead for external definitions unless instructed to do so. We'll get to how to do that soon.

What happens if we declare, inside a function, an automatic variable with the same name as an external variable? Then the local definition overrides the external definition within that function. A slight alteration of the last example illustrates this point:

```
int x = 27;      /* an external variable */

main()
{
    int x = 42;   /* an automatic variable */
```



```

    printf("main() knows that x is %d.\n", x);
    fap();
}

fap()
{
    printf("fap(), too, knows that x is %d\n.", x);
}

```

And the output:

```

main() knows that x is 42.
fap(), too, knows that x is 27.

```

As you can see, `main()` uses the local `x`, while `fap()` uses the global `x`.

To document your programs better, you can use the keyword **extern** to identify those variables used in a function but defined externally. That is, a better way to write the initial scope example would be this:

```

int x = 27;      /* creates x */

main()
{
    extern int x; /* use the external x */

    printf("main() knows that x is %d.\n", x);
    fap();
}

fap()
{
    extern int x; /* use the external x */

    printf("fap(), too, knows that x is %d\n.", x);
}

```

Using the **extern** keyword makes your intentions clear. And if you want to use a local variable with the same name as an external variable, you can use the keyword **auto** to emphasize that you are deliberately reusing a variable name.

Note that the keyword `extern` is only used to identify a variable already defined elsewhere. Do not use it in the initial definition of the variable.

One further effect of using the keyword `extern` is that the compiler will then search the whole file for an external definition, so the definition could come later in the file.

One minor point: if you simply declare a variable to be `extern` instead of `extern char` or `extern int`, etc., the compiler will assume you mean `extern int`.

Because an external variable is defined outside of any function, it exists from the time the program starts to when the program ends—unlike automatic variables, which come and go as individual functions are called and terminated.

The Static Storage Class

Some functions benefit from knowing what they did the preceding call. A random number generator, for instance, has to move along one more number from the preceding call. Otherwise it would always return the same random number, which is not that useful. Using the static keyword in a declaration causes the variable and its value to be retained in memory between function calls. Here is an illustrative, if not illustrious, example:

```
main()
{
    int ct;

    for ( ct = 0; ct < 4; ct++)
        statictest();    /* call function 4 times */
}

statictest()
{
    int fade = 1;          /* an ordinary variable */
    static int nofade = 1; /* a static variable */

    printf("fade = %d, nofade = %d.\n", fade++,
nofade++);
}
```

Here is the output:

```
fade = 1, nofade = 1.  
fade = 1, nofade = 2.  
fade = 1, nofade = 3.  
fade = 1, nofade = 4.
```

Notice how the increment operator increments `fade` and `nofade` after each printing, but only `nofade` "remembers" that fact on the next function call. Another point to note is that if a static variable is initialized in the declaration statement, that initialization takes place only once. Automatic variables, if initialized, get initialized every time the function is called.

A static variable defined within a function, then, combines the local nature of the automatic variable with the persistence of the external variable.

The External Static Storage Class

The keyword `static` can be applied to externally defined variables, too. Since external variables already last the duration of the program, this might seem to be gilding the lily. However, in this context, the keyword `static` plays a role other than extending the lifetime of a variable. When used with an external variable, `static` *limits* the scope of the variable to a single file. Regular external variables, on the other hand, can be shared over several files. Clearly, we need to discuss multiple-file programs now.

Multiple-File Programs

Up to now we have assumed that the entire program is contained in one file. But it is not necessary nor even always desirable to use just one file. Any one function should be confined to one file, but a multifunction program can spread the functions through several files. Sometimes this has to be done because of size limitations to the largest compilable chunks of program. Or it can be a matter of convenience; one file can hold an integrated package of functions that you use in several programs.

How do you compile programs spread over more than one file? In general, each file is compiled individually, then the results are "linked" together into a single program. In Hippo C, use the pull-down Programs menu to identify the files to be used, and Hippo C takes care of the rest. We won't go into the details of the process. Instead, let's look into the question of variable scope in multiple-file programs.

Suppose we have two files. The first looks like this:

```
/* the file scopea.c */
int world = 3; /* an external variable */
main()
{
    printf("In main(), world = %d.\n", world);
    notmain();
}
```

The second looks like this:

```
/* the file scopeb.c */
notmain()
{
    printf("In notmain(), world =%d.\n", world);
}
```

Can these two files be compiled successfully together? No. The problem is that the variable `world` is known only in file `scopea.c` and not in file `scopeb.c`. But this need not be the case. We merely need to redeclare `world` in the second file, using the keyword `extern`. We can do this either inside the `notmain()` function or outside of it. Thus, either of the following two versions will compile successfully with `scopea.c`:

```
/* the file scopeb.c, version 1.1 */
notmain()
{
    extern int world;

    printf("In notmain(), world =%d.\n", world);
}
```

```
/* the file scopeb.c, version 1.2 */
extern int world;

notmain()
{
    printf("In notmain(), world =%d.\n", world);
}
```

Either version informs the compiler to search elsewhere for the original definition of world.

External variables are often used in large, multifile programs. Sometimes, one of those files will need a set of variables private to that file but global to all the functions within that file. That is when the external static storage class is used. The file format could look like the following:

```
static short flags;      /* global to this file */
static int g1, g2, g3;   /* ditto */
extern int bigflags;     /* defined in another file */

void function1()
...
int function2()
...
void function3()
...
```

As indicated, the variables flags, g1, g2, and g3 are shared by and restricted to the functions in the file, while bigflags comes from another file.

In short, automatic and local static variables are limited in scope to a single function. An external static variable is limited in scope to a single file, and an external variable can have its scope extended to several files. The keyword extern must be used to declare a variable used in one file but defined externally in another file. The original definition does not use the extern keyword.

Recursion

One notable property of a C function is that it can call itself. This is called **recursion**. Of course, if a function calls itself, then the new call will call itself again, and so on, ad infinitum, unless the function incorporates some test to terminate the calling.

Each time the function is called, a new set of variables is created for it, so the x of one call is not the x of a subsequent recursive call. Here is a short sample that illustrates how recursive functions behave.

```

main()
{
    int i;
    void recur();

    printf("Enter an integer:\n");
    scanf("%d", &i);
    recur(i); /* call a recursive function */
}

void recur(n);
int n;
{
    printf("Going in: %d\n", n);
    if ( n > 0 )
        recur ( n - 1 ); /* the recursive call ! */
}

```

Here is an annotated sample run:

```

Enter an integer:
2 [RETURN]
Going in: 2          <-Level 1 call
Going in: 1          <-Level 2 call
Going in: 0          <-Level 3 call

```

Is it clear what is happening? Here is a summary. When the `recur()` function is first called (Level 1), memory is allotted for the variable `n`, and then the value 2 is assigned to `n`. Next, the value of `n` is printed. Then, since `n` is greater than 0, the `recur()` function is called again (Level 2), this time with an argument of 1 less than before. A new, distinct `n` is created and assigned the value 1. It, too, is printed, and `recur()` is called again (Level 3). One more `n` is created and printed. However, at this level, the `if` test fails, so there are no more `recur()` calls.

The program doesn't stop when the third level call to `recur()` ends. Whenever a function terminates naturally, control returns to the function that called it. The calling function then resumes its progress at the statement following the function call. Thus, when Level 3 ends, control returns to Level 2. But the function call was the last statement of Level 2, so it, too, comes to an end, returning control to Level 1, which then returns control to `main()`. The important point here is that in a recursive chain, each call to a function is balanced by a return to the calling function.

Is that perfectly clear? See if you can predict what the following modification of our program will print; you may have to master the desire to look ahead at the output:

```
main()
{
    int i;
    void recur();

    printf("Enter an integer:\n");
    scanf("%d", &i);
    recur(i);
}

void recur(n);
int n;
{
    printf("Going in: %d\n", n);
    if ( n > 0 )
        recur ( n - 1 );
    printf("Coming out: %d\n", n); /* the modification */
}
```

Suppose we once again enter 2. The program will start out as before, creating and printing a series of variables (all named *n*) until a 0 value is reached. At that point, the final call to `recur()` will skip the `if` statement and proceed to the final `printf()` statement. Thus, it prints the 0 value twice. Then control returns to Level 2. This level resumes at the statement following the function call; that would be the final `printf()` statement. Thus, Level 2 will print out *its* *n* value, which is still 1. Control passes to Level 1, which prints a 2, then control passes to `main()`, at which point the program ends.

Here is an annotated sample run:

```
Enter an integer:
2 [RETURN]
Going in: 2          <--Level 1 call
Going in: 1          <--Level 2 call
Going in: 0          <--Level 3 call
Coming out: 0        <--Level 3 call
Coming out: 1        <--Level 2 call
Coming out: 2        <--Level 1 call
```

You should remember these two points about recursive calls: each level of call maintains its own variables, and each level, when it finishes, returns control to the preceding level. Figure 5.3 offers a visual presentation of these ideas.

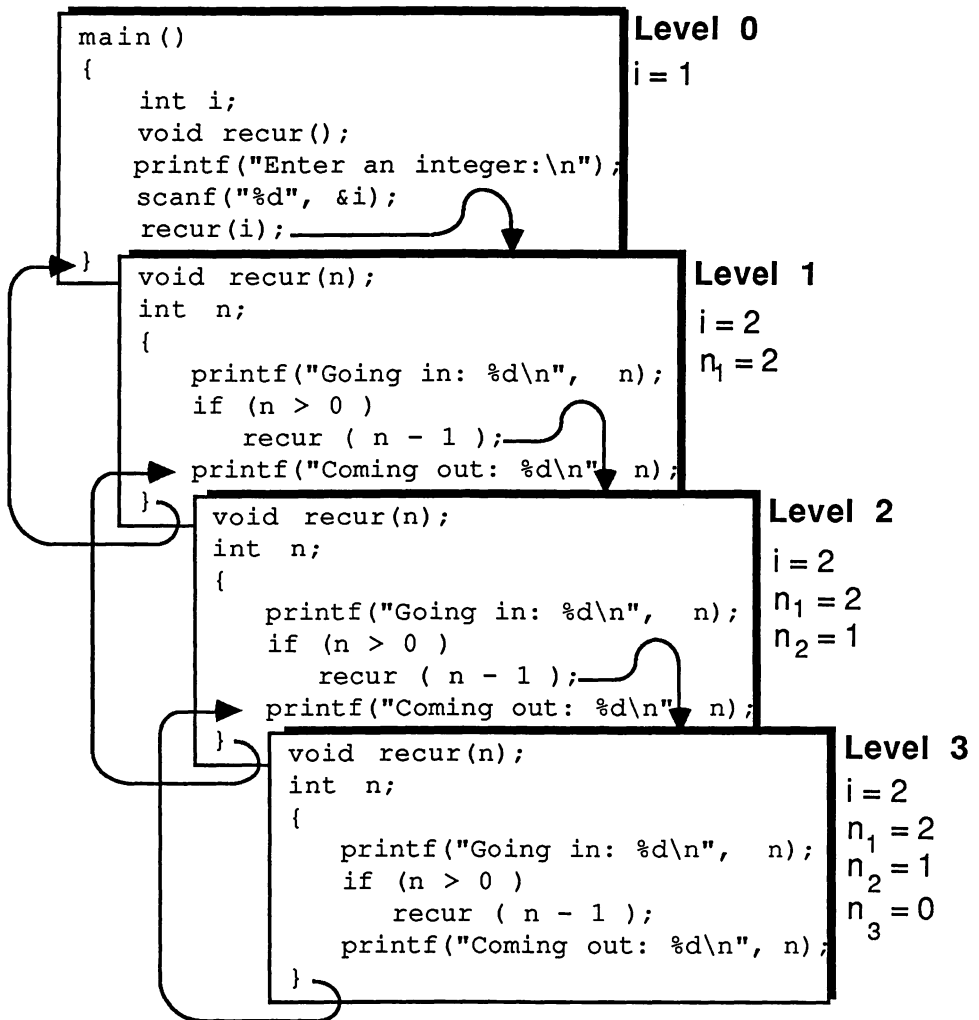


Figure 5.3 Recursive Calls

Recursive programming is useful for certain situations. It raises the possibility, however, of running out of memory, for each recursive call generates a new set of function variables while the old sets still remain in memory.

Macros

One advantage of functions is that they can be used for code segments that are used repeatedly in a program. By placing such code in a function instead of "in-line", we need type it but once. Furthermore, only one copy is stored in memory, so functions reduce the amount of memory needed to hold a program. However, using functions instead of in-line code increases the running time of a program because it takes time to shift to the function, create its variables, and return to the calling program.

C offers an alternative method for handling repetitive code, one that produces in-line code with its faster run-time and greater memory requirements. The method is to use the preprocessor `#define` directive to create "macros". The macro gives us a shorthand way to express a functional relationship. For example, suppose we have a program that often requires the absolute value of a quantity. Earlier in this chapter we saw three ways to write a function for this purpose. Here's the third version:

```
abs(x)
int x;
{
    return x < 0 ? (-x):x;
}
```

Here is how we would set up a macro to perform the same task:

```
#define ABS(X)  ( (X) < 0 ? -(X) : (X) )
```

The use of uppercase is common usage, but it isn't mandatory. This example is similar to the past `#defines` we've used, but it adds an *argument* to the definition. When the macro is used, this argument is replaced literally by the actual argument appearing in the program. Note: there must be *no spaces* in the first expression following the `#define`. That is, don't use an expression such as `ABS(X)`. Why not? We'll get to that soon.

We then can use ABS() much the same as we would use a function, providing it with arguments which are then used in place of X. (We'll come back to the multitude of parentheses later.) For examples, we can have program lines like the following:

```
x = ABS (y) ;  
z = ABS ( q*r - 3) ;
```

The mechanism, however, is quite different from that of a regular C function. For instance, if we use the function call

```
z = abs ( q*r - 3) ;
```

then, during *run* time, the expression $q*r - 3$ is evaluated, and the value is passed on to the abs() function. But if we use the macro call

```
z = ABS ( q*r - 3) ;
```

then, *prior* to compilation, the preprocessor *replaces* the macro with the corresponding code. The code actually submitted for compilation for this example would be this:

```
z = ( (q*r - 3) < 0 ? -(q*r - 3) : (q*r - 3) ) ;
```

Every occurrence of X in the macro definition is replaced with the expression $q*r - 3$. That code appears in the program at the same location previously occupied by the macro. Thus, a macro is not a function call; rather, it is shorthand for in-line code.

Macro Oddities

The fact that a macro results in a substitution rather than a function call necessitates all the parentheses we used in the above example. To take a simpler example, suppose we used this macro definition:

```
#define SQR(X) X*X
```

Now look at some of the possible substitutions resulting from using this macro:

<code>x = SQR(5);</code>	<code>-> x = 5*5;</code>
<code>x = SQR(y+2);</code>	<code>-> x = y+2*y+2;</code>
<code>x = z/SQR(y);</code>	<code>-> x = z/y*y;</code>

Only the first example produces the required result! For the other two, operator precedence warps the intended meaning. The safest thing to do to avoid this sort of problem is to use the following rules:

1. Enclose each instance of an argument in parentheses.
2. Enclose the entire defining expression in parentheses.

Thus a better definition of SQR(X) is this:

```
#define SQR(X) ( (X) * (X) )
```

You may wish to check that this handles the earlier examples correctly.

Another point is to avoid using the increment operators when using a macro. For instance, suppose we used this:

```
y = SQR(x++);
```

This would be translated to the following:

```
y = ( (x++) * (x++) );
```

Ugh! Not only does *x* get incremented twice instead of once, but one incrementation takes place between evaluating expressions, so that the actual value produced is *x*(x+1)*.

The prohibition against spaces in the first expression following *#define* also stems from the way substitution works. Suppose we defined *advice* and made this definition:

```
#define SQR( X ) ((X)*(X))
```

and then made this call:

```
y = SQR( z );
```

A *#define* directive takes the first contiguous group of nonspace characters as the pattern to be substituted for. The rest of the line becomes that which is substituted. Thus, our attempt would produce this butchered code:

```
y = X ) ((X)*(X)) z );
```

Only the *SQR(* portion is replaced.

Another macro oddity is that they are typeless. Our *ABS()* macro can be used with floating-point or with integer variables. The type of the expression depends on the type of the argument present when the macro is used.

As a final oddity, macro argument symbols that appear inside of double quotes are substituted for by the preprocessor but not during run-time. Consider this macro:

```
#define PR(X) printf("The value of X is %d\n", X)
```

Suppose we use this in a program:

```
y = 5;  
PR(y);
```

The substituted code for the macro is this:

```
printf("The value of y is %d\n", y);
```

Running the program produces this output:

```
The value of y is 5
```

The value 5 was used for the second y, but the first y, being in double quotes, remained a literal y.

Multiple Arguments

Macros allow for multiple arguments. Just provide an argument list with the arguments separated by commas and without spaces. For example, we can define a maximum value macro this way:

```
#define MAX(X,Y) ( (X) > (Y) ? (X) : (Y) )
```

Note the use of parentheses.

Summary

When a program is broken into separate modules, communication between the modules is usually necessary. C provides three mechanisms for communication between functions. The first is the **argument list** of a function. This is a list of local variables created when a function is called. These variables (the formal arguments) are then assigned the values from the corresponding expressions that constitute the actual argument list in the function call.

The argument list mechanism is a one-way communication channel from calling function to called function. However, by using addresses as actual arguments and pointer variables as formal arguments, we can construct functions that manipulate values in the calling program.

The second communication channel is the **return** mechanism. It, too, is a one-way channel, providing for a value to be sent back to the calling program. Only one value can be returned, and the type of the returned value is the type of the function. Functions are assumed to be type `int` by default. If a function is of some other type, this type must be declared in the function definition and in the calling program. Functions that do not return a value may be declared type `void`.

The third communication device is to establish **global, or shared, variables**. An external variable is one declared outside of any function. Its scope may be extended to several files. A static external variable can be accessed by all the functions in a single file. Variables declared within a function are, by default, automatic, or local variables and are limited in scope to the function containing them. A local declaration overrides an external definition within the confines of the function.

If a function is to alter the value of a variable, it should be passed the address of the variable. The corresponding formal argument in the function should be declared as a **pointer** to whatever type the original variable is. A pointer variable is a variable whose value is an address. If `p` is a pointer, then `*p` refers to the value stored at the indicated address. Thus, if `p` is, say, a pointer-to-`int`, then `*p` can be used as an `int`.

Pointer arithmetic allows you to add to or subtract from a pointer. Adding 1 to a pointer increases its value by the size of the pointed-to type in bytes. A pointer can be subtracted from a pointer of the same type. The actual difference in bytes is divided by the number of bytes in the pointed-to type.

C functions are all on the same level; that is, no function is defined inside of another. Thus, any function can call any other function. Even `main()` can be called, should you care to try it. Also, a function can call itself. This sets up recursion. A function designed to call itself should also be designed to have some sort of mechanism to halt the sequence of **recursive calls**.

The C preprocessor offers a **macro facility** that is used much like a function. Instead of producing function calls, however, a macro results in code being substituted for the macro before compilation. Macros run faster than functions but may require more storage if used several times.

6

Arrays and Structures

In this chapter you will learn about:

- Defining and using arrays
 - Pointers and arrays
 - Arrays as function arguments
 - Defining and using structures
 - Functions and structures
 - Quickdraw structures
-

A program is a blend of action and information. In C, the *operators* generate the action, and various forms of *data structures* store the information. The simplest way to store data is to use a single-valued variable, as we have been doing all along. We'll call this data form a simple variable. Often, however, a program deals with a wealth of related data, and it becomes desirable to store several items of information in a single data structure of some sort. In this chapter, we will look at two C forms for storing data: the "array" and the "structure".

There is a bit of terminology we should mention. The term "data structure" is used in computer science to denote a variety of data forms, including the array. C uses the word "structure" to denote a particular variety of data structure. This makes the use of the word "structure" a bit ambiguous in C; however, the context should make the meaning clear.

An array is a device for storing several items of the same data type in adjacent memory locations. For instance, if we were developing a program for Wormtech, Inc., we might use an array to store the monthly worm sales over a three-year period.

A C structure is also used for storing several items of data. In this case, they need not be of the same type. Typically, a structure is used to hold several diverse items of information pertaining to the same entity or object. For example, a payroll program might use a structure to store the name, social security number, pay rate, and other such information about a particular employee. The C structure corresponds closely to the Pascal

record. The Toolbox depends heavily upon structures, so understanding them is essential to programming for the Macintosh. Indeed, structures provide the key to digging more deeply into the Toolbox, which we will do in this chapter. We'll use several Quickdraw routines based on structures.

These basic forms can be extended. We can use arrays of structures, structures containing arrays and other structures, arrays of arrays, and so on. But before getting too carried away, let's explore the fundamentals first.

The Array

An array consists of a sequence of identical storage elements. We can have an array of ints, an array of chars, an array of pointers-to-char, and so on. Each individual unit within the array is an "element" of the array.

To create an array, we declare it, indicating the number and the type of elements. For instance, to create an array of 12 ints, we would use this declaration:

```
int tapesales[12]; /* tapesales is an array of
                   12 ints */
```

The square brackets announce that the identifier is an array name; the number within the brackets gives the number of elements. You must use square brackets; parentheses or braces will not do. This particular declaration instructs the computer to set aside 12 contiguous locations for storing ints.

We need a method to refer to the individual elements of an array. One way is to use the "index", or "subscript", notation. In this notation `tapesales[0]` would be the first element of the array, `tapesales[1]` would be the second element, and so on. Note that the numbering always starts with 0. Thus, the last element of a 12-element array has a subscript of 11. Figure 6.1 shows the correspondence between array elements and subscripts.

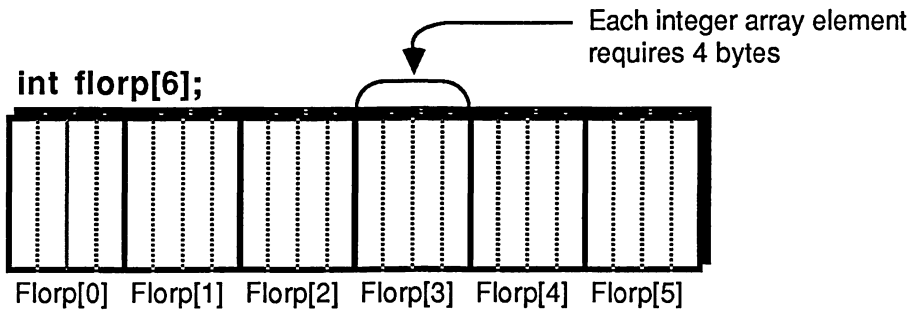


Figure 6.1 Array Elements

Each element in an array essentially is a variable of the array type, `int` in this case. Thus we can use, say, `tapesales[3]` as an argument to a function that takes an `int` argument. Or we can make assignment statements like this:

```
tapesales[8] = 341;
```

Arrays are often used with `for` loops, with a loop variable serving as an index for the array. For instance, this programette would serve to read data into a waiting array and then print the data back:

```
main()
{
    int tapesales[12];
    int index;

    printf("Please enter the monthly tape sales for last
year:\n");
    for ( index = 0; index < 12; index++)
        scanf("%d", &tapesales[index]);
    printf("Okay, this is what you entered:\n");
    for (index = 0; index < 12; index++)
        printf("%4d ", tapesales[index] );
    putchar ('\n');
}
```

Each of the two loops processes each element of the `tapesales` array in turn. Note that we can use the address operator with an array element just as with a simple `int` variable. That is, `&tapesales[5]` is the address at which the sixth element is stored.

We could have placed the `scanf()` and `printf()` statements in the same loop. In that case, each figure would be echoed as it was entered. With the present form, the figures aren't reprinted until all the entries have been made. As a matter of programming method, it is better to place distinct activities in separate sections of code. That way, you can modify one aspect without getting involved in the other.

Here is a sample run:


```
Please enter the monthly tape sales for last year:
```

```
22 27 36 33 44 121[RETURN]
```

```
88 92 60 102 101 143[RETURN]
```

```
Okay, this is what your entered:
```

```
22 27 36 33 44 121 88 92 60 102 101  
143
```



Remember, the `scanf()` function skips over spaces and newlines, so we can spread the input data over as many lines as we like.

Incidentally, don't take this example as a model for user-friendly interactive input. Typing an incorrect character anywhere along the 12-number input sends the program into a tizzy. But we are illustrating arrays now, not input-verification techniques.

An easy error to fall into is forgetting that the maximum index size is 1 less than the array size and using a test such as `index <= 12`. Fortunately, we did not make that mistake. Because the final index of an array is one less than the array size, the correct condition is `index < 12`. Another possibility is `index <= 11`, but that makes it less obvious that we are working with 12 elements.

Rather than using a number like 12, however, try using a defined constant for the array size and in the loop controls. One advantage is that you can test the program for a small array size. Once you work the bugs out, you then can redefine the constant once and not need to go through the whole program changing array sizes and loop limits. Here is a short program using a defined constant for the array size; it also illustrates that we can access the elements of an array in any order we want.

```
#define SIZE 10  
main()  
{
```

```

char chs[SIZE];
int i;
printf("Please enter %d characters:\n", SIZE);
for ( i = 0; i < SIZE; i++)
    chs[i] = getchar(); /* assign input to
                        successive array members */
printf("\nHere they are in reverse order: ");
for ( i = SIZE - 1; i >= 0; i--)
    putchar(chs[i]);
putchar('\n');
}

```

Here is a sample run:

```

Please enter 10 characters:
ostensible
Here they are in reverse order: elbhisnetso

```

Once again, note that an array member (here `chs`) is used just like a simple variable of the same type (`char`).

Arrays and Pointers

In C, there is a strong connection between arrays and pointers, and C programmers often use pointers to process arrays. The connection begins with the fact that the name of an array also serves as a pointer to the first element of the array. Thus, for the `tapesales` array, we have the following identity:

```
tapesales == &tapesales[0]
```

Recall that the address operator (`&`) yields the address of its operand; thus `&tapesales[0]` is the address of the array element `tapesales[0]`.

That an array name is a pointer to the array is the most important aspect of the array-pointer connection to remember. As we'll see, functions that manipulate an array typically require a pointer to the array. Calls to these functions, then, would use array names (without the `&` operator) as actual arguments.

One consequence of this property of an array name is that we can use pointer notation instead of array notation to describe an array. Let's see how this works. We know `tapesales` is a pointer to the first array element. By using pointer addition, we can obtain pointers to the other elements. Recall that adding 1 to a pointer increases the actual address size by the size of the type pointed to. Since `tapesales` points to type `int`, `tapesales + 1` will point 4 bytes (for a Hippo C `int`) farther down the line. That's the next element of the array! That is, `tapesales + 1` points to `tapesales[1]`. In general, `tapesales + index` is a pointer to `tapesales[index]`. We can express that relationship by this equality:

```
tapesales + index == &tapesales[index]
```

If we use the indirect value operator to obtain the value a pointer points to, we can re-express the preceding relationship this way:

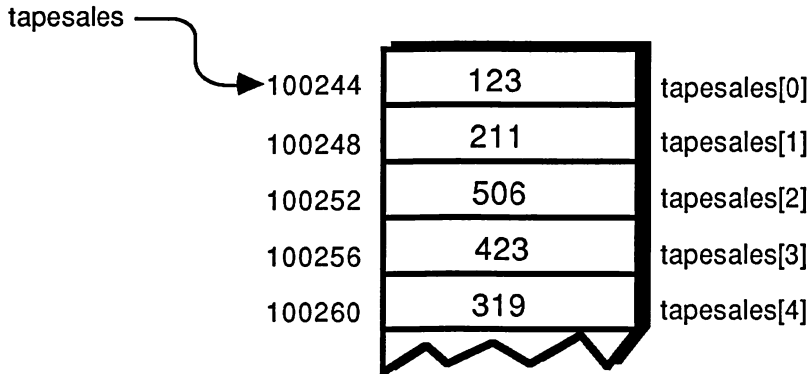
```
*(tapesales + index) == tapesales[index]
```

In other words, we can use pointers instead of array notation to access array members.

In this example, we had to use parentheses because the `*` operator has higher precedence than `+`. Suppose we leave them off; what does the following expression mean?

```
*tapesales + 3
```

Well, `*tapesales` is the value of the first element, so this means add 3 to the value of the first element. With parentheses, we were adding to the address, not to the value. Figure 6.2 illustrates these points.



**Pointer addition
followed by
indirect value**

```
tapesales == 100244
tapesales + 3 == 10256
* (tapesales+3) == 423
```

**Indirect value
followed by
ordinary addition**

```
tapesales == 100244
* tapesales == 123
* tapesales + 3 == 126
```

Figure 6.2 Pointers and Arrays

As an exercise, let's rewrite the last program, this time using pointer notation instead of array notation. All we need to do is replace each reference to `tapesales[index]` with `*(tapesales+index)` and each reference to `&tapesales[index]` with `tapesales+index`. Thus the program would look like this:

```
#define MONTHS 12
main()
{
    int tapesales[MONTHS]; /* still use array notation to
                           declare */
    int index;

    printf("Please enter the monthly tape sales for last
    year:\n");
    for ( index = 0; index < MONTHS; index++)
        scanf("%d", tapesales + index);
    printf("Okay, this is what you entered:\n");
    for (index = 0; index < MONTHS; index++)
        printf("%4d ", *(tapesales + index) );
    putchar('\n');
}
```

It's instructive to know that we can use pointer notation instead of array notation, but is it useful? After all, `tapesales[3]` is easier to type and simpler to interpret than the equivalent `*(tapesales + 3)`. One point to keep in mind is that C uses pointers itself; during compilation, the array forms are converted to pointer forms. Thus, using the pointer forms may make you feel like an insider. A second point is that many C programmers prefer pointers, so that if you wish to read their programs, you need to know about the pointer forms. A third point is that you do need to use pointers in one form or another when you use functions to process arrays.

Arrays and Functions

Suppose we want a function that does something with an array. Can we pass an array as an argument? Not exactly. In C, each argument must be a single value. If you wanted a function to process an array with 365 elements, you could use 365 separate arguments, but that is no fun at all. Is there a simpler way to represent an array? Certainly there is; you can describe an array by providing a pointer to the first element of the array (you can use the array name for that!) and by providing the number of elements. Then the function can start at the first address and keep going until all the elements are processed.

Once again, let's resort to a simple example to illustrate the mechanics. Let's create a `daytosec()` function that converts an array of day values to an array of second values. The function needs two arguments, a pointer to the first array element and the number of elements. Suppose the original array is of type `int`; then the first argument will be a pointer to `int`. The number of elements can be type `int`, although unsigned short might be more appropriate, unless you anticipate enormous array sizes. Anyway, the function can look like this:

```
void daytosec( parray, n)
int *parray; /* pointer to first array element */
int n;      /* number of elements in array */
{
    int index;

    for ( index = 0; index < n; index++)
        *(parray + index) *= 86400;
}
```

The key line here is

```
*(parray + index) *= 86400;
```

Let's paraphrase it. Go to the location pointed to by parray. Move index locations further over. Find the number there, multiply it by 86400, and assign this new value to the same location. By letting index vary, the function performs this operation upon each array member in turn.

Note the convenience offered by the multiplicative assignment operator. The statement we used is much more compact than the following:

```
*(parray + index) = *(parray + index) * 86400;
```

True, the expression `*(parray + index)` is odd-looking; just keep in mind that `parray + index` is just a way to point to the successive members of the array, so the complete expression indicates the successive values stored in the array. Indeed, if you feel resistant towards using pointers, you can write the program using array notation instead:

```
void daytosec( parray, n)
int parray[]; /* parray is a pointer to first array
               element */
int n;        /* number of elements in array */
{
    int index;

    for ( index = 0; index < n; index++)
        parray [index] *= 86400;
}
```

This may look different, but it is the same program! The declaration

```
int parray[];
```

is just an alternative way of saying parray is a pointer-to-int. Although it uses

brackets, no size value is enclosed. This form is used instead of

```
int *parray;
```

when you wish to emphasize to a reader that the integer that `parray` points to is an element of an array. As far as the compiler is concerned, however, the two forms are equivalent. Similarly, we've already seen that `parray[index]` and `*(parray + index)` are equivalent, so the two programs are the same. Indeed, you can switch the `parray` declarations between the two versions without affecting the workings of the function.

We will usually use the array notation because it seems more obvious, but be aware that when you use that notation, you are really using pointers in a disguised form.

It is important to realize that neither version creates a new array. Both use pointers to let the function manipulate the contents of the original array. If you want the function to work on a copy of the original array, you will have to program the function to create a copy. That involves using memory management functions, so we won't get into that now.

Another important point is that the approach we have used (passing an array pointer and an array size) allows the function to work on arrays of different sizes. This makes C different from (and most would say superior to) standard Pascal, in which a given procedure works only for arrays of one size. (In Pascal, arrays of different sizes actually represent different types, and Pascal is very strict about not letting you mix types.)

Let's try out one version so that we can see an actual function call. For checking the function, it would be convenient to start off with an initialized array so that we don't have to type in values. Because array initialization is a new topic, we will show the example in the next section.

Array Initialization

A simple variable can be initialized when declared, using statements like this:

```
int planets = 9;
```

Not all storage classes of arrays, however, can be initialized similarly. In particular, automatic arrays (the default type for in-function declarations) can not be initialized. But the other classes can be. To test our function, we will use a static array so that the array can be initialized. The following example shows how that is done.

```
#define DTOS 86400      /* number of seconds in a day */
#define SIZE 4          /* elements in test array */
main()
{
    int cheese;          /* cheese batch */
    static int ages[SIZE] = {10, 24, 15, 60};
                          /* declaring and initializing an array */
    void daytosec();

    printf("The cheese ages in days are\n");
    for( cheese = 0; cheese < SIZE; cheese++)
        printf("Batch %d: %d days\n",cheese+1, ↵
ages[cheese]);
    daytosec(ages, SIZE); /* pointer and
                           element number */
    printf("The cheese ages in seconds are\n");
    for( cheese = 0; cheese < SIZE; cheese++)
        printf("Batch %d: %d secs\n",cheese+1, ↵
ages[cheese]);
}

void daytosec( parray, n)
int parray[]; /* parray is a pointer to first array
               element */
int n;        /* number of elements in array */
{
    int index;

    for ( index = 0; index < n; index++)
        parray [index] *= DTOS;
}
```

Notice the actual arguments: the array name `ages` is a pointer to the array, and `SIZE` is the number of elements, both as required by the function. We use `cheese+1` in the `printf` so that the zero subscript will correspond to batch 1. Here is the output:

```
The cheese ages in days are
Batch 1: 10 days
Batch 2: 24 days
Batch 3: 15 days
Batch 4: 60 days
The cheese ages in seconds are
Batch 1: 864000 secs
Batch 2: 2073600 secs
Batch 3: 1296000 secs
Batch 4: 5184000 secs
```

The form we've used for initializing nonautomatic arrays is this:

```
type    arrayname [size] = {list};
```

Here *list* represents a list of comma-separated values. The list can have fewer entries than the array has elements, but not vice versa. Elements of nonautomatic arrays that are not explicitly initialized get initialized to zero.

C allows a lazy form of initialization that looks like this:

```
static short sizes[] = {30, 32, 34, 36, 38};
```

You can leave out the array size when you initialize an array, and the compiler will match the array size to the number of initialization entries. In the above case, `sizes` would be made into a 5-element array.

If you need to assign values to an automatic array, you can have the program read them in or else you can assign values to each element individually. For instance, this fragment creates an automatic array and initializes its elements to 0:

```
int lettercounts[26];
int i;

for (i = 0; i < 26; i++)
    lettercounts[i] = 0;
```

Copying Arrays

In Pascal, if *orig* and *workcopy* are two arrays of the same type (which in Pascal also implies arrays of the same size), you can copy one entire array to another with a single Pascal statement:

```
workcopy := orig
```

Can the same be done in C? No, it cannot. In C, the name of an array is a pointer constant. A constant can't be assigned a value (try sneaking $3 = 4+2$; by the computer), and even if it could, you would wind up with two pointers to the same array rather than two arrays.

We raise this point so that those of you with Pascal backgrounds don't try something foolish. Also, it gives us the opportunity to get more practice with functions using arrays. So let's write a function that copies one array into another. The function will assume that both arrays already have been created.

First, what arguments would this function need? It should have a pointer to each array, of course, and it should have the number of elements to be copied. That's all that's needed, so let's write the function. We'll begin with perhaps the most obvious version; it's also a correct version:

```
/* copies n elements from array a to array b */
void arraycopy( a,b,n)
int a[], b[]; /* a,b are pointers to an array */
int n;        /* number of elements to be copied */
```

```

{
    int i;

    for(i = 0; i < n; i++)
        b[i] = a[i];
}

```

This program sets the first element of one array equal to the first element of the next, then continues the process to the end of the array. Here is a sample program using it:

```

main()
{
    static int old[4] = { 2, 4, 8, 16};
    int new[4];          /* old and new are genuine arrays */
    int i;
    void arraycopy();

    arraycopy(old,new,4);
    printf("  old  new\n");
    for (i = 0; i < 4; i++)
        printf("%5d %5d\n", old[i], new[i]);
}

void arraycopy( a,b,n)
int a[], b[]; /* a,b are pointers to an array */
int n;        /* number of elements to be copied */
{
    int i;

    for(i = 0; i < n; i++)
        b[i] = a[i];
}

```

Here is the output:

old	new
2	2
4	4
8	8
16	16

The function does work.

There is an important difference between old and new on the one hand and a and b on the other. As we saw earlier, old and new are pointer constants, incapable of change. But a and b are pointer *variables*. Thus, while old++ is no more valid than 3++, a++ *is* valid. In fact, it advances the pointer a so that it points to the next array member. Our function added a different i to the original a value to march through the array, but we can use the increment operator to the same effect. Here is a more compact version of the copy function:

```
/* copies n elements of array a into array b */
arycpy(a,b,n)    /* even the name is more compact */
int *a, *b, n;   /* a, b pointers and n an int */
{
    while ( n-- ) /* count down until no elements left to
                  copy */
        *b++ = *a++; /* copy element, advance pointer */
}
```

For a picture of what's happening, visualize two arrays. Think of a as a finger pointing to the first element of one array, and of b as a finger pointing to the first element of the other. What b points to gets copied to where a is pointing, then each finger moves on to the next element. In the meantime, the while loop keeps track of how many elements we have copied. Recall that a zero value is false, so the while loop keeps going until n reaches 0. If n starts at, say, 10, then we get 10 cycles. We save a variable by counting n down instead of having an additional variable increase up to n.

Let's take a closer look at the expression *a++. The precedence table says * and ++ have the same precedence, but it also says these operators associate from *right to left*. Thus, *a++ is the same as *(a++), meaning "use what a points to, then move a to point to the next element". That is, the *pointed-to value* is used, but the *pointer* is incremented. On the other hand, (*a)++ would mean "take the value pointed to by a and increase the value by one." And, while we are at it, *++a would mean "make a point to the next element and use the value of that element." Thank heavens we didn't use either of those. As you can see, this compact version encompasses many subtle points. Understand it well, and you have gone a long way towards understanding pointers and the increment operators.

We'll return to arrays later; now let's move on to that mainstay of the Toolbox, the structure.

Structures

A C structure is a data form capable of holding several items of data of different types. FORTRAN and BASIC have no analogous form, but the structure is essentially the same as a Pascal record. In C, each individual component of a structure is called a "member". We'll look at how to define a structure and how to access its members.

Defining a Structure

A structure definition has, in general, four parts: the keyword **struct**, followed by an identifying "tag", followed by the body of the definition (enclosed in braces), followed by the name(s) of the structure(s) created. Sometimes the tag or the name portion (but not both) will be missing. Before entering further explanations, let's look at a sample declaration:

```
struct grades {  
    int quizave;  
    int examave;  
    char grade;  
} stilton;
```

Here we have defined a structure having the name stilton. The structure has 3 members: quizave, examave, and grade. The type for each member is declared in the same manner that we've used for ordinary variables. The tag is grades, and it provides a shorthand way to refer to the structure "template" we've set up. If, for example, you want to create other structures of the same form, you could now make declarations of this sort:

```
struct grades jessup, flatney;  
/* creates 2 structures */  
struct grades class86[30]; /* creates array of 30  
                           structures */  
struct grades *gp;         /* creates a pointer to a  
                           structure */
```

The phrase `struct grades` acts as a type name. The structures themselves are named `stilton`, `jessup`, and so on. In short, the tag `grades` identifies a pattern, or template, while `stilton` and `flatney` identify particular structure variables that conform to that template.

If you are creating just one structure of a certain form, you can omit the tag portion. Or, if you wish to create a template to be used for several functions, you can define the structure form externally, using a tag and omitting any names. Then you can use the tag within each function to define the structures you want. Before going into these elaborations, however, let's see how to use a structure.

Accessing Structure Members

Suppose we want to assign an 'A' to the `grade` member of the `stilton` structure. We can do it this way:

```
stilton.grade = 'A';
```

The period between `stilton` and `grade` is the C "membership" operator, and the term `stilton.grade` means "the `grade` member of the `stilton` structure." It is important to realize that `stilton.grade` is a variable of type `char`, the declared type for `grade`. Since it is important, we'll reword the thought: `stilton` is of type `struct grade`, but `stilton.grade` is of type `char`. Similarly, `stilton.quizave` is of type `int`. In other words, the *final* identifier in a structure membership phrase determines the type.

Figure 6.3 illustrates the memory layout of the `stilton` structure.

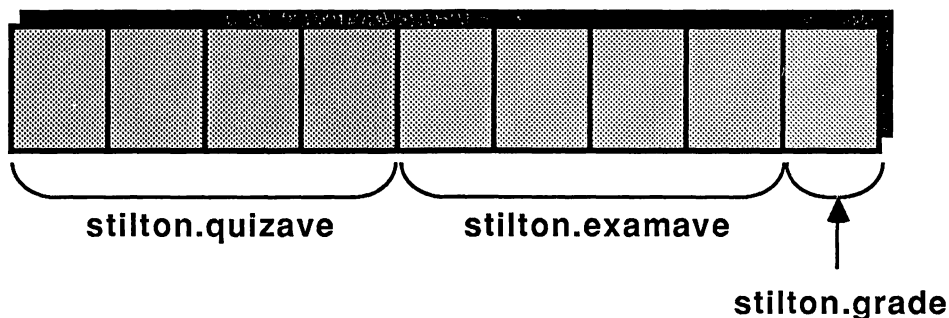


Figure 6.3 The `stilton` structure

Using a Structure in a Program

Let's put together a short program illustrating how a structure can be used in a program. We'll do the following:

Set up a structure template

Declare a structure variable

Assign a value to a structure member

Read a value into a structure member

Manipulate structure members

Print the contents of a structure

One key point to keep in mind is that each structure member can be used the same way as a simple variable of the same type. Here, then, is our simple program:

```
struct bills {
    int denom;
    int number;
    int value;
};    /* sets up template with the tag
      "bills" */

main()
{
    struct bills tens; /* establishes variable
                       "tens" */

    tens.denom = 10; /* assign value to "denom"
                     member */
    printf("How many %d bills do you have on you?\n",
tens.denom); /* use member as int argument */
    scanf("%d", &tens.number); /* use member
                               address */

    tens.value = tens.denom * tens.number;
    printf("Your %d %d bills are worth %d.\n",
tens.number,
tens.denom, tens.value);
}
```

Here is a sample run:

```
How many $10 bills do you have on you?  
123[RETURN]  
Your 123 $10 bills are worth $1230.
```

One important point to remember is to use the variable name, not the tag name, when specifying members. Thus, we used `tens.value` and not `bills.value`. Also, note that `&tens.number` is the address of the number member of the structure. The expression `&tens` would be the address of the beginning of the whole structure.

Now let's turn to the Toolbox for more examples of a structure and its uses.

A Quickdraw Structure

The Quickdraw package includes many functions that use structures. One structure used by several functions is the **rect** structure. The structure template can be defined this way:

```
struct rect {  
    short top;  
    short left;  
    short bottom;  
    short right;  
};
```

The four members are all type short. The top member represents the coordinate of the top of the rectangle, while the bottom represents the rectangle's lower coordinate. Similarly, left and right delimit the left and right ends of a rectangle. The coordinate system is the same as before, with a full screen ranging from 0 to 342 top to bottom, and from 0 to 512 left to right, measured from the upper left of the current window. Because the four members are all the same type, the same information could have been placed in an array. However, the fact that they represent four distinct attributes of a single object make it more logical to place them in a structure.

To illustrate how such a structure is used, let's try an example. It uses four Quickdraw routines based on the rect structure. Here's a quick rundown using the Toolbox names:

From Mac's Toolbox: New Routines

EraseRect	Erases interior of a rectangle
SetRect	Sets rectangle boundaries
FrameRect	Draws rectangular shape
FrameOval	Draws oval shape

The program, of course, uses the Hippo C representation of the function names. When you read the program and the following commentary, you'll note that most of these functions take just the address of a type struct rect variable as an argument. Only occasionally will you, as a programmer, refer to the individual members of a Quickdraw structure. The Quickdraw functions themselves do most of the actual manipulation of structure members, while you work with the structure as a unit.

```
/* boxoval.c -- use Quickdraw routines to draw oval in a
   box */
#define TOP 0
#define BOTTOM 342
#define LEFT 0
#define RIGHT 512
struct rect { short top, left, bottom, right }; /* short
                                                form */
main()
{
    struct rect box; /* box is a rect structure */
    box.top = TOP; /* set box to full screen */
    box.left = LEFT;
    box.bottom = BOTTOM;
    box.RIGHT = RIGHT;
    eraserect(&box); /* set everything inside box to
                     background */
    setrect(&box, 50, 50, 300, 200); /*another way to set
                                     bounds */
    framerect(&box); /* draw a rectangle as described by
                     box */
    frameoval(&box); /* draw an oval bounded by box */
}
```

Before running this program from the Hippo C Command Window, use the mouse to expand the window to approximately full screen. Figure 6.4 shows the program's output.

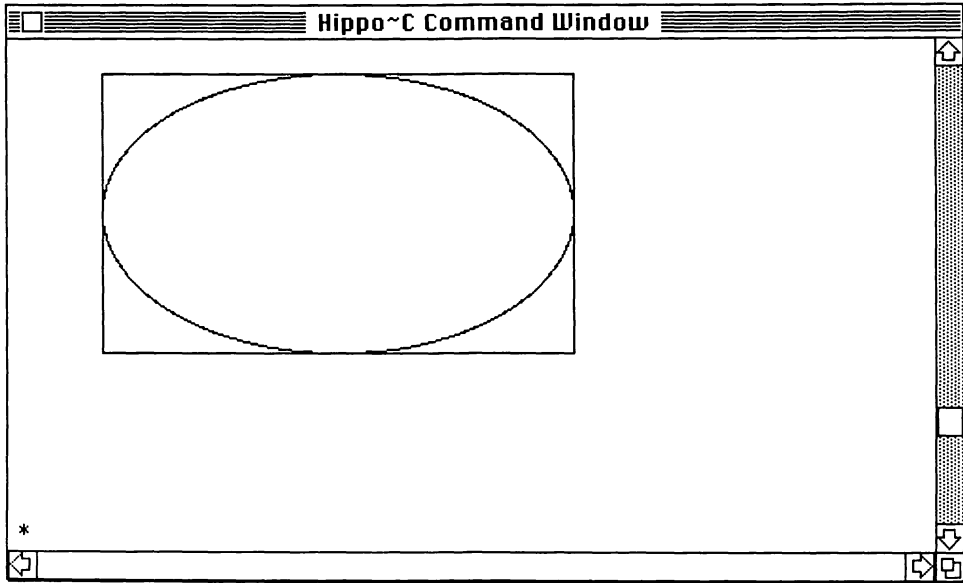


Figure 6.4 Output from `boxoval.c`

Now we have some comments to make. First, since all four members of the `rect` template are short, we lumped their declarations together. We defined the template externally, in case we ever expand the program to more than one function. Inside `main()` we define a struct `rect` variable called `box`, and then we set the limits of `box` to correspond to the screen limits. (Note that we have to say `struct rect` and not just `rect`.) At this point, the program has a mathematical description of the box, but it has not yet been told to do anything with it. Simply setting the `box` values does not cause a rectangle to be drawn.

The first graphics command is `eraserect()`. It takes a pointer-to-struct `rect` as an argument, so we provide it with the address of `box`. Note: unlike the case for arrays, the name of a structure is *not* a pointer to the beginning of the structure. We need to use the address operator explicitly to get the address. Once `eraserect()` gets the address of a `rect` structure, it looks into the structure to see boundaries of the area it is to affect. Then it sets everything within those boundaries to the screen background pattern, effectively erasing everything within the confines of the rectangle. We set

the boundaries to erase the whole screen, but only that part of the screen revealed by the current window is affected.

The next command we meet is **setrect()**. It takes a pointer-to-struct **rect** and four short integers as argument. The **setrect()** function offers an alternative way to assign values to the members of the structure. The order of the setting arguments is left, top, right, bottom -- slightly different from the order used in defining the **rect** structure, which was top, left, bottom, right. Using this function requires more computer time but less typing than assigning values individually, as we did at first.

The next command, **framerect(&box)**, instructs the computer to draw a rectangle fitting the boundaries given by **box**. Actually, the frame is drawn just inside the mathematical boundaries of the box. Thus, an **eraserect()** call for the same rectangle will erase a frame. Once again, a structure address is required as an argument. Also, once again, only that portion of the figure that lies within the current window is drawn, which is why you should first expand the Hippo C Command Window.

Finally, **frameoval()** works much the same as **framerect()**, except that it inscribes an oval within the indicated boundaries. Note that it makes use of the **rect** structure; no new structure type is needed.

Note the power of these commands. We can duplicate the effect of **framerect()** by using **moveto()** and **drawto()**, but why bother? And **frameoval()** and **eraserect()** add new abilities. Let's run another example and learn a few more Toolbox functions. Here are the new routines we'll use; the first three are Quickdraw routines:

From Mac's Toolbox: New Routines

OffsetRect	Shift rectangle boundaries
InsetRect	Shrink or expand rectangle boundaries
InvertOval	Invert interior of oval
TickCount	Returns a time value

We also define a function of our own called **wait()**; it uses **tickcount()** to provide a time delay.

```
/* ovals.c -- use Quickdraw routines to draw ovals */
#define TOP 0
```

```

#define BOTTOM 340
#define LEFT 0
#define RIGHT 512
struct rect { short top, left, bottom, right };
                /* short form */
main()
{
    struct rect box;
    void wait();

    setrect(&box, TOP, LEFT, BOTTOM, RIGHT);
    eraserect(&box);
    setrect(&box, 50, 50, 300, 200);
    frameoval(&box);
    while ( box.top < box.bottom && box.left < box.right)
        {
            wait(15);          /* wait 1/4 second */
            invertoval(&box);   /* invert interior of oval */
            offsetrect(&box, 10, 5); /* move conceptual oval */
            insetrect(&box, 15, 10); /* shrink conceptual oval */
            frameoval(&box);     /* draw the new oval */
        }

    void wait(ticks)
    int ticks; /* one tick = 1/60th of a second */
    {
        int start;

        start = tickcount(); /* set start time */
        while ( (tickcount() - start) > ticks );
                /* do nothing until allotted time passes */
    }
}

```

This program starts off like the last one, except that the first figure it draws is an oval. Then a loop prints additional ovals. To help the viewer see what happens, we've put in a time delay loop in the form of a `wait()` function. The argument of this function is the time delay in "tick" units, where one tick is one-sixtieth second. The `wait()` function uses the `tickcount()` function from the Toolbox. This function returns the number of ticks since the system started up. The `wait()` function is not exact, for it fails to take into account the time needed to run itself, but it is close enough for our purposes. We'll use it several more times in this book.

After a wait of about 1/4 second, the `invertoval()` function is activated. It inverts the contents of the currently defined oval, making bright pixels on

the screen dark, and dark pixels bright. Since we start with a bright screen, we wind up with a completely dark oval.

Then the `offsetrect()` function changes the elements of `box` in such a way as to shift the boundary rectangle over. The first argument is a pointer to the rectangular region to be shifted. The second argument is horizontal displacement (positive to the right, negative to the left). The third argument is vertical displacement (positive down, negative up). This function does not change the screen; it just changes the `box` structure.

Similarly, the `insetrect()` function changes the size of the boundary, keeping the center in the same position. Again, the first argument is a pointer to the `rect` structure to be affected. The second argument is the amount each vertical side is moved in, and the third argument is the amount each horizontal side is moved in. Negative values produce outward movement. Still, the screen is not affected.

Finally, `frameoval()` is used to draw an oval according to the modified specifications in `box`. Since it is drawn on a black background, it doesn't show. But the `invertoval()` at the start of the next loop cycle converts its interior back to white, so it does show. The loop continues until the adjustments try to make the top below the bottom or left to the right of right.

Figure 6.5 shows the final figure, but not the inversions that take place as the program runs.

Another Look at the Quickdraw Functions

You may have noticed that the rectangle functions we've discussed come in two classes. First, there are those that set up or modify the mathematical description of a rectangle: `setrect()`, `offsetrect()`, and `insetrect()`. Second, there are those that use the mathematical description as a guide to some graphics action on the screen: `eraserect()`, `framerect()`, `frameoval()`, and `invertoval()`. Only those in the second class affect the screen. For instance, if you use `framerect()` to draw a rectangle and then use `offsetrect()` to move the conceptual rectangle, the rectangle that was already drawn stays put.

A second point is that the coordinates used for these functions are measured from the upper-left corner of the current window. Thus, the position of the window affects the actual screen location of drawn figures.

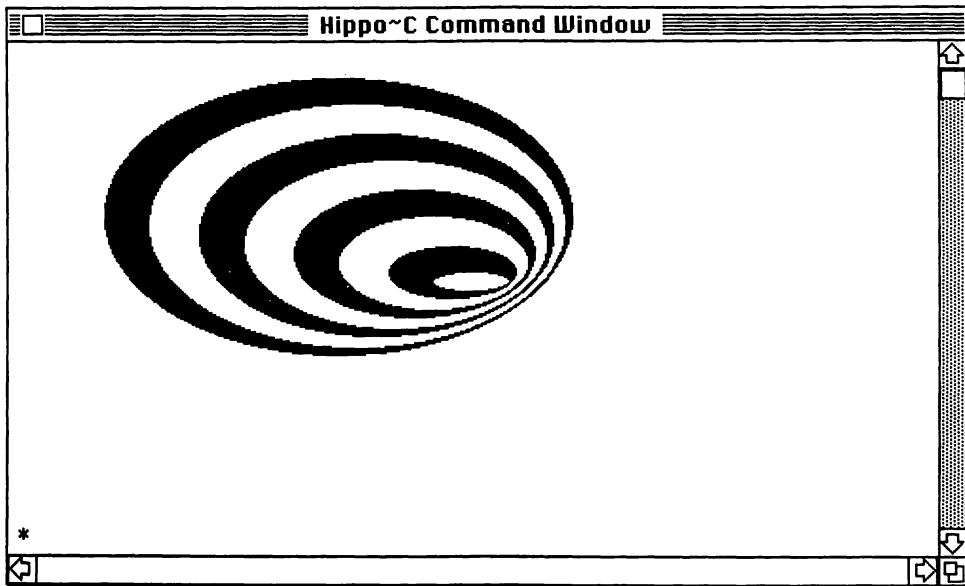


Figure 6.5 Output of ovals.c

Third, those functions that affect the screen only affect the parts inside the current window. Thus, if the program tries to draw a rectangle 300 pixels tall in a window that is only open 200 pixels in height, only part of the rectangle will show.

Hippo C and Quickdraw

It's probably time to remind you of what we said in Chapter 3, namely, that Quickdraw has to be initialized before its routines are used. In Hippo C, this is done for you in both the Hippo C environment (the Hippo C Command Window) and in the HOS operating system. In each case, your program takes over the environment set up by Hippo C. In HOS, the whole screen is provided. In the Hippo C environment, the Hippo C Command Window is provided. By default, it occupies about the lower half of the screen. You can use the mouse to move the window about and to alter its size. For many of our programs, you should expand the Command Window first. Alternatively, you can run the program from HOS.

In Chapter 9, you will learn how to let a program set up its own drawing area independent of the current Hippo C Command Window.

Functions, Structures, and Pointers

Functions like `insetrect()` make manipulating the `rect` structure convenient. Suppose, though, that we didn't have this function at our disposal. How would we go about writing one? In general, how do we write a function that uses a structure?

To answer the specific question, first consider how we could shrink down the contents of a given `rect` structure using in-line code. Then we can see how to convert that programming to a generalized function. Suppose, then, that we have a `rect` structure called `box` and that we want to move in the vertical sides by an amount `dh` and the horizontal sides by `dv`. Then we would adjust the structure members this way:

```
box.left += dh;      /* move left side to right */
box.right -= dh;     /* move right side to left */
box.top += dv;       /* move top side down      */
box.bottom -= dv;    /* move bottom side up       */
```

All we needed to do was to use the membership operator to access each member of the structure variable `box`. We used the additive and subtractive assignment operators. Of course, if you want your programming to look more like other languages, you could use the following expressions:

```
box.left = box.left + dh;
box.right = box.right - dh;
box.top = box.top + dv;
box.bottom = box.bottom - dv;
```

Most people coming to C from other languages find this form more comfortable. Yet the additive and subtractive assignment operators probably make more sense. Certainly, a statement like

```
x = x + 6;
```

is poor algebra and translates to clumsy English: "Take `x`, add 6 to it, and

assign the result to x". On the other hand, the statement

```
x += 6;
```

comes closer to how one might express the thought in English: "Take x and increase it by 6 ." If the form looks odd, that's just a matter of exposure. Use C's unusual operators, and soon they seem usual.

Now suppose we wish to convert the in-line programming to a function. As you recall, a function that needs to change values in a calling function does so by using a pointer to the variable(s) in the calling function. Thus we need to pass a pointer to a rect structure as one argument. Two more arguments can pass the horizontal and vertical changes; these can be type short. Fine, those are exactly the arguments used by insetrect(), so we must be on the right track. The head of our function (call it shrinkrect()) should look like this:

```
shrinkrect( rp, dh, dv)
struct rect *rp;    /* pointer to a rect structure */
short dh, dv;      /* horizontal, vertical shrinkage */
```

Now comes the tricky part: how do we access members of a structure when we have a pointer-to-structure instead of a structure name to work with. Actually, we have two choices. The clumsy choice goes along these lines: if rp is a pointer-to-structure, then *rp is the semantic equivalent of a structure name. Thus, the top member of the structure pointed to by rp can be expressed this way:

```
(*rp).top    /* the top member of the pointed-to
              structure */
```

The parentheses are needed because the period has a higher precedence than *.

The elegant way is to use the "indirect membership" operator, `->`. It is produced by typing a hyphen (`-`) followed by a "greater than" symbol (`>`). It is used just like the membership operator, but with a pointer-to-structure instead of a structure name. Here is how to represent the top member in this notation:

```
rp->top      /* top member of structure pointed to
              by rp */
```

In short, use the membership operator (`.`) when working with a structure name, and use the indirect membership operator (`->`) when working with a pointer-to-structure. With this knowledge, we can complete the function:

```
shrinkrect( rp, dh, dv)
struct rect *rp;    /* pointer to a rect structure */
short dh, dv;
{
    rp->left += dh;    /* left member of pointed-to
                       structure */
    rp->right -= dh;
    rp->top += dv;
    rp->bottom -= dv;
}
```

And that is how to use pointers-to-structures in functions designed to modify structures.

Passing Structures by Value

Originally, using a structure address was the only way to pass structure information to a function. However, many recent implementations of C, including Hippo C, allow a structure variable to be passed by value in the same manner as a simple variable. For example, if `box` is a structure, you can have a function call of this form:

```
afunction(box);
```

In this case, the function head would look like this:

```
afunction( bx )  
struct rect bx;
```

The argument this time is a structure instead of a pointer-to-structure. A new structure variable is created by a call to this function, and it is filled with the values held in the structure used in the function call.

Passing by value for structures has the same advantages as passing by value for simple variables: compartmentalizing functions and preventing inadvertent alteration of data. However, the pointer-passing technique is required for the Toolbox functions we've been using, since that is how they were set up. In any case, the pointer form is necessary for functions that alter the original structure.

Friendly Advice

This chapter has covered much new material: arrays, pointers, structures, functions using arrays and structures, and new Toolbox routines. Just reading the chapter is most likely not enough to make you comfortable with the new ideas. You need hands-on experience to really learn them. Thus, you should work through the examples, not just read them. Next, you should fiddle with the examples, modifying them to test your understanding. Finally, you should write some programs of your own. For example, by using `framerect()`, `eraserect()`, `offsetrect()`, and some form of time delay, such as `wait()`, you can write a loop that makes a rectangle appear to move about the screen. The Hippo C manual provides a brief description of several other rectangle-related functions; try using some of them. Write functions of your own using arrays and structures. It's possible you may make an occasional error, but that's okay. You probably can learn more from your mistakes than your successes, for mistakes make you think more about what is going on.

The next chapter elaborates upon this one, so make sure you have at least a workable mastery of this chapter before moving on. The next section gives a summary of the main points you should know.

Summary

C offers a variety of data forms. Two forms that can hold more than one data item are the **array** and the **structure**. An array holds a sequence of elements all of the same type. An index, or subscript, is used to indicate particular elements. To declare an array of 10 chars, do this:

```
char fishname[10];
```

The brackets identify fishname as an array name, and the 10 indicates the number of storage locations allotted to the array. Subscripts start at zero, so the first element of the array is fishname[0], the second element is fishname[1], and so on.

In C the name of an array also serves as a pointer to the first element of the array. In our example, for instance, fishname can be considered a named constant whose value is the address of the first byte of the array.

The structure, like the array, can hold several separate values. However, the structure is capable of holding a mixture of types. In general, a structure definition contains four parts: the **keyword struct**, a "**tag**" to act as a shorthand description for the structure, a **member-definition** section enclosed in braces, and a **list of structure variables** declared to be that type. Here is a sample structure declaration:

```
struct thetag {  
    int socsecno;  
    int years;  
    char name[25];  
} headman, assistant;
```

The tag thetag can be used to declare subsequent structure variables of the same type. It is a *template* tag.

The individual parts of a structure are called **members**. To access a member, follow the structure name with the membership operator, which is a period, and then the member name. Thus, headman.years and assistant.socsecno access individual members. The type for such expressions is the type of the right-most identifier.

Members can also be accessed using pointers to a structure. Suppose we have this sequence:

```
struct thetag *ps; /* a pointer to a structure */  
  
ps = &headman;    /* assign structure address to  
                  pointer */
```

Then the years member of headman can be referred to as ps->years, where -> is the indirect membership operator.

An array cannot be passed en masse as a function argument, but a pointer to an array element (such as an array name) is a valid function argument. In this case, the formal argument in the function definition must be declared as a pointer to the corresponding element type. The net result is a function that operates upon the elements in the original array.

Many newer compilers, including Hippi C, do allow a structure to be passed as a function argument. In that case, a copy of the entire structure is created and used within a function. More typically, structure-oriented functions are designed to accept a pointer to a structure as an argument. These functions would then use the original structures for their manipulations.

7

Compound Data Structures

In this chapter you will learn about:

- Arrays of structures
 - Structures of arrays
 - Arrays of arrays
 - Complex declarations and typedef
 - Pointers to functions
-

You have seen how arrays and structures can be constructed from fundamental data types, such as `int` and `char` variables. However, the members of a structure and the elements of an array need not be simple variables. They can be structures, arrays, pointers, structures of arrays of pointers, and so on. We'll look at some of the more common combined forms in this chapter, seeing how to define and use them. We will also investigate how to write functions that work with such combined data types. En route, we'll work some more with Quickdraw functions.

Declaring these forms can get complicated, so we will review the rules for making declarations. Also, we will investigate the `typedef` facility for creating easily used abbreviations for complex types. Finally, we'll make a side trip to view pointers to functions.

Arrays of Structures

Consider this declaration:

```
struct rect boxes[3];
```

This creates an array with 3 elements. Each element of the array is a structure of the `rect` type. Figure 7.1 illustrates how it is stored in memory.

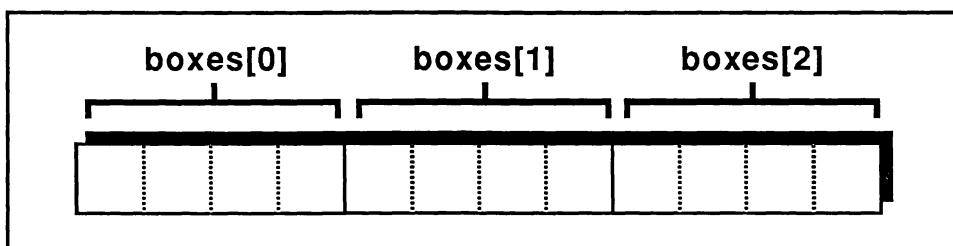


Figure 7.1 An array of structures

When you have a data form with components, a natural question is, how do you access the individual elements and members? With an array of structures, we have two levels of access. First, we should be able to access each structure as a unit. Second, we should be able to access each member within a given structure. The solution is to use both array and structure member notation. For instance, if you wish to use the second structure of the array, refer to it this way:

```
boxes[1]
```

After all, `boxes[1]` is the second element of the `boxes` array (remember that subscripts start at zero), and each element of the array is a structure.

Continuing, to access the top member of the second structure in the array, you would use this notation:

```
boxes[1].top
```

Here the membership operator has been applied to the structure called `boxes[1]`.

Let's look at the various stages involved in building up this identifier. There are three: `boxes`, `boxes[1]`, and `boxes[1].top`.

boxes: The identifier **boxes** is the name of an *array*, not of a structure. Following the usual C convention for arrays, *boxes* is a pointer to the first element of the array; that is, it points to the first structure in the array. Its numerical value is the address of the first byte of the structure. Its type is pointer-to-struct rect.

boxes[1]: Because each array element is a structure, the subscripted identifiers, such as `boxes[0]` and `boxes[1]`, *are* structure names. Thus, they can be used whenever structure names are required. For example, you can use them with the address operator when you use a function that requires the address of a structure:

```
framerect( &boxes[1]);
```

The argument here is just the address of the second structure in the array. The brackets have a higher precedence than the `&` operator, so the address operator applies to whole expression `boxes[1]`. Array elements also can be used with the membership operator, as we see next. The type for `boxes[1]` is the array type, `struct rect`.

boxes[1].top: Combining the structure name (`boxes[1]`) with the membership operator and member name yield an individual member of the structure called `boxes[1].top`. The whole identifier is the same type as declared for the member, here `short`. The expression can be used in any fashion that an ordinary variable of the same type can. For instance,

```
boxes[2].top = 200;
```

assigns the value 200 to the `top` member of the third structure in the array.

A Graphic Example. Here is a program that uses an array of structures. It also shows how to initialize a structure and introduces some new Toolbox functions.

From Mac's Toolbox: New Routines

InvertRect	Inverts interior of a rectangle
PaintRect	Fills interior of a rectangle

The program brings back two functions from the past. One is the Toolbox function `button()`, which returns "true" if the mouse button is

down and "false" otherwise. The second is the wait() function we introduced in Chapter 6 to provide a time delay.

```
/* checkpat.c -- a blinking pattern */
#define TOP 50
#define BOTTOM 100
#define LEFT 56
#define RIGHT 456
struct rect { short top,left,bottom,right; };
main()
{
    static struct rect box = { 0, 0, 512, 342};
    struct rect boxes[8];
    int i;
    void wait();

    erasect(&box);
    setrect(&box, LEFT, TOP, RIGHT, BOTTOM);
    framerect(&box);          /*outline graphics area */
    for ( i = 0; i < 8; i++)
        setrect(&boxes[i],LEFT + 50*i,TOP,LEFT + 50*(i+1),
            BOTTOM);
        /* set array boxes to side-by-side rectangles */
    for ( i = 0; i < 8; i++)
        if( i % 2 == 0)
            framerect(&boxes[i]);      /* draw outline */
        else
            paintrect(&boxes[i]; /* fill solid figure */
    while( !button() ) { /* interesting visual effect */
        wait(30);
        invertrect(&box);
    }
}
```

First, we initialized a structure. The rules are the same as for arrays. Only nonautomatic structures can be initialized, which is why we used the static storage class. The list of values is placed between braces, with the values separated by commas.

We've introduced another Toolbox function, **paintrect()**. It fills in the boundaries of the specified rectangle with the current paint pattern, which is normally solid black by default. (In Chapter 9 we will indicate how to reset such parameters.) By using the modulus operator (%), we instruct the program to draw the rectangles having an even index and to paint the

odd ones. This produces a row of checkerboard pattern, as shown in Figure 7.2.

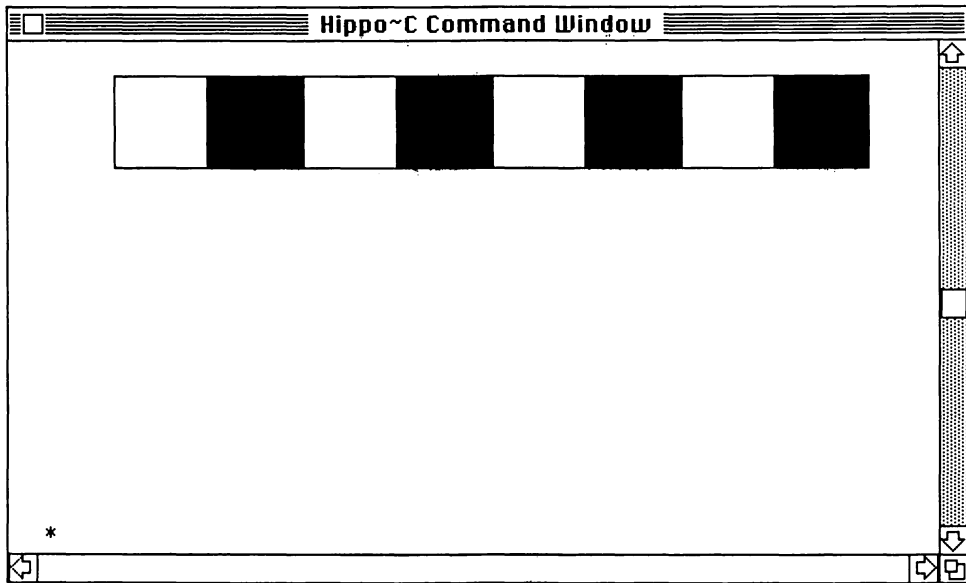


Figure 7.2 `checkpat.c` output

The box rectangle contains all the boxes' rectangles, and the closing while loop in `main()` makes use of this fact to invert the pattern every 30 ticks until you push the mouse button. This loop produces an interesting visual effect.

And where is the `wait()` function defined? We placed the one we wrote earlier in a separate file, then compiled the two files together, using the process described in Chapter 5.

A Structure in a Structure

A structure member can itself be a structure. Consider this definition:

```
struct lbox {  
    char letter;  
    struct rect box;  
} tess;
```

Here the second member of `tess` is a structure. This definition assumes that `rect` structure has been defined earlier. You know how to access the letter member; just call it `tess.letter`. But how do you access, say, the top member of the box member?

The answer is the logical one: use the membership operator twice. Thus, the top member of the box member of the `tess` structure is this:

```
tess.box.top
```

Let's review the steps leading to this identifier.

tess: This is the name of a structure of the `lbox` type. Hence this identifier is of type `struct tess`.

tess.box: This is a member of the `tess` structure. The type for `tess.box` is the type declared for `box`, which is `struct rect`. Hence `tess.box` also is a structure name, this time a type `rect` structure. The identifier `tess.box` can be used in the same manner as any other structure name of that type. In particular, we can use the membership operator to obtain its members.

tess.box.top: This is the name of the top member of the `tess.box` structure. The entire name `tess.box.top`, then, represents a variable of the type declared for `top`, which is short.

In short, with nested structures, keep using the membership operator to work down to individual members.

An Array in a Structure

An array, too, can be a structure member. Here is a declaration that creates just such a marvel:

```
struct rain {
    int year;
    int rainfall[12];
} bandon;
```

Here the year member would hold a year date, while the rainfall array would hold monthly totals.

Again, a logical approach suffices to establish proper identifiers. The name **rain** is the name of a structure. The name **rain.rainfall** is a member of the structure, and this member is an array of 12 ints. Thus, **rain.rainfall** is an array name, hence is a pointer to the first element of the array. The names of the individual array members, then, are **rain.rainfall[0]**, **rain.rainfall[1]**, and so on.

Note that when we had an array of structures, the array brackets were to the left of the membership operator, unlike the case here.

The next declaration takes us a step further to an array of structures containing an array:

```
struct rain {  
    int year;  
    int rainfall[12];  
} cities[200];
```

The **cities** array holds data for 200 cities. If you want access to the 3rd month's rainfall of the 87th city, merely combine the various rules to get **cities[86].rainfall[2]**. (Don't forget that array numbering always starts with 0.)

Arrays of Arrays

Our next selection of compound data type is the array of arrays. This is an array whose elements themselves are arrays. Here is how to declare such a critter:

```
int grid[3][4];
```

This states that **grid** is a three-element array, and that each element is an array of 4 ints.

Often, such arrays are visualized as "two-dimensional" arrays, since we can arrange the final elements in a two-dimensional pattern. For instance, we can picture **grid** as consisting of three rows, each with four

elements. To indicate any one element, use two subscripts. For instance `grid[2][3]` would be the element in third row, fourth column. Again, keep in mind that array numbering starts with 0. Figure 7.3 illustrates this representation.

```
int grid[3][4]
```

grid[0]	grid[0][0]	grid[0][1]	grid[0][2]	grid[0][3]
grid[1]	grid[1][0]	grid[1][1]	grid[1][2]	grid[1][3]
grid[2]	grid[2][0]	grid[2][1]	grid[2][2]	grid[2][3]

Figure 7.3 A two-dimensional array

It's worth going over the various identifiers involved with a two-dimensional array.

grid: This is an array name. As such, it is a pointer to the first element of the array, which, we've seen, is itself a four-element array. We can term its type "pointer-to-array-of-four-ints."

grid[0]: This is the first element of `grid`, so `grid[0]` *also* is an array name. In Figure 7.3, it is the name of the first row. Because it is an array name, `grid[0]`, too, is a pointer. In this case, `grid[0]` points to *its* first element, `grid[0][0]`. Similarly, `grid[1]` is an array name for the second row, and it points to the element `grid[1][0]`. Since that element is type `int`, `grid[1]` and its fellows are type pointer-to-`int`.

grid[0][0]: This is an element of the `grid[0]` array. It is of type `int` and can be treated as any other type `int` variable.

In short, `grid` points to `grid[0]`, and `grid[0]` points to `grid[0][0]`. Thus, `grid` is a pointer to a pointer, our first example of such.

Conceptually, the difference between `grid` and `grid[0]` is clear; `grid` is a two-dimensional array and `grid[0]` is a one-dimensional array, the first row of `grid`. In pointer terms, `grid` points to a whole array of four integers, while `grid[0]` points to a single integer. (Perhaps you've heard this before: the name of an array is a pointer to the first element of the array.) But, since both the array and the single element start at the same location in memory (see Figure 7.3), `grid` and `grid[0]` have the same numerical value, the

address of the first byte of the whole array. What, then, is the practical distinction between pointing to an array and pointing to an int? Check out this next little program and its output:

```
main()
{
    int grid[3][4];

    printf("grid: %u; grid[0]: %u\n", grid, grid[0]);
    printf("grid+1: %u; grid[0]+1: %u\n",
        grid+1, grid[0]+1);
}
```

The output:

```
grid: 106776; grid[0]: 106776
grid+1: 106792; grid[0]+1: 106780
```

As promised, both `grid` and `grid[0]` have the same value. But look what happens when 1 is added to each. Since `grid[0]` points to type `int`, adding 1 to the pointer makes it point to the next `int`, which is four bytes more. But `grid` points to an array of four `ints`, an entity that is sixteen bytes long. Thus, adding 1 to `grid` means adding sixteen bytes so that it would point to the next array. Remember, pointer addition is always in units of whatever object is pointed to.

Initializing Two-Dimensional Arrays

Only arrays of the static or the external storage class can be initialized. As with one-dimensional arrays, braces are used to enclose the initialization values. You can use additional braces to mark off each subarray, but they are not required. Thus, the following statements initialize the arrays `twink` and `twonk` to the same values:

```
static int twink[2][3] = { {10, 12, 13},
                           {22, 34, 15} };
static int twonk[2][3] = { 10, 12, 13, 22, 34, 15};
```

The first form emphasizes that we are initializing an array of two arrays of three ints. Each subset of braces corresponds to a row. Note that a comma is used to separate one subset from the next. When using the second form, keep in mind that the right-most array index varies most rapidly. That is, the first 4 values are `twunk[0][0]`, `twunk[0][1]`, and `twunk[0][2]`, then `twunk[1][0]`.

The main functional difference comes when you supply fewer values than the array can hold. When you use just one set of braces, the array elements are filled up in the order in which they are stored. Consider the following statement:

```
static twunk[2][3] = { 5, 6, 7, 8};
```

This results in 5 being assigned to `twunk[0][0]`, 6 to `twunk[0][1]`, 7 to `twunk[0][2]`, and 8 to `twunk[1][0]`. Using subbraces lets you fill up the subarrays just partially:

```
static twenk[2][3] = { {5,6} ,  
                       {7,8} };
```

This initializes the first two elements of the array `twenk[0]` to 5 and 6 and the first two elements of the array `twenk[1]` to 7 and 8.

Using a Two-Dimensional Array

It's time for an example using a two-dimensional array. Earlier, the `checkpat.c` program used a one-dimensional array to construct a row of squares. A natural extension is to use a two-dimensional array to create more than one row, and that is what we will do. This time we have an array of an array of structures. Also, just as a for loop is often used to process a one-dimensional array, nested for loops are often used for two-dimensional arrays. That will be the case here. Also, we'll use our `wait()` function to slow down the display so you can visually check the order in which the for loops execute. Once again we'll assume that `wait()` is in a separate file that gets compiled along with the main program.


```

/* cboard.c -- makes a checkerboard-like pattern */
#define TOP 50
#define LEFT 56
#define SIZE 50          /* sides of individual boxes */
struct rect {short top,left,bottom,right;};
main()
{
    static struct rect screen = {0,0,512,342} ;
    struct rect boxes[4][8];          /* 4 rows, 8 columns */
    int row, col;
    void wait();

    erasect(&screen);                  /*clear screen */
    for( row = 0; row < 4; row++)      /* for each row */
        for (col = 0; col < 8; col++) /* and each column */
            setrect(&boxes[row][col], LEFT + col*SIZE,
                    TOP + row*SIZE, LEFT + (col + 1)*SIZE,
                    TOP + (row + 1)* SIZE );
    /* that sets bounds for all 32 rectangles */
    /* now draw them all */
    for( row = 0; row < 4; row++)
    {
        for ( col = 0; col < 8; col++)
        {
            if( (col+row) % 2 == 0)
                framect(&boxes[row][col]);
            else
                paintrect(&boxes[row][col]);
            wait(15);
        }
        wait(30);
    }
}

```

Be sure to provide adequate window room for the display. Running this program produces the splendid pattern shown in Figure 7.4

Note that **boxes[row][col]** is a structure name, so that **&boxes[row][col]** is a proper argument for the Quickdraw functions. The first nested for loop uses the values of row, col, and SIZE to describe boxes that are displaced progressively one box size to the right and down from the first box. The second nested loop produces framed boxes if the sum of row and col is even, and filled-in squares otherwise.

In drawing, the inner loop processes boxes in the same row. The next cycle of the outer loop lets the inner loop cycle through the next row. The

pauses in the program let you see this development visually. If you like, you can exchange the two for lines and see the pattern drawn by columns instead of by rows.

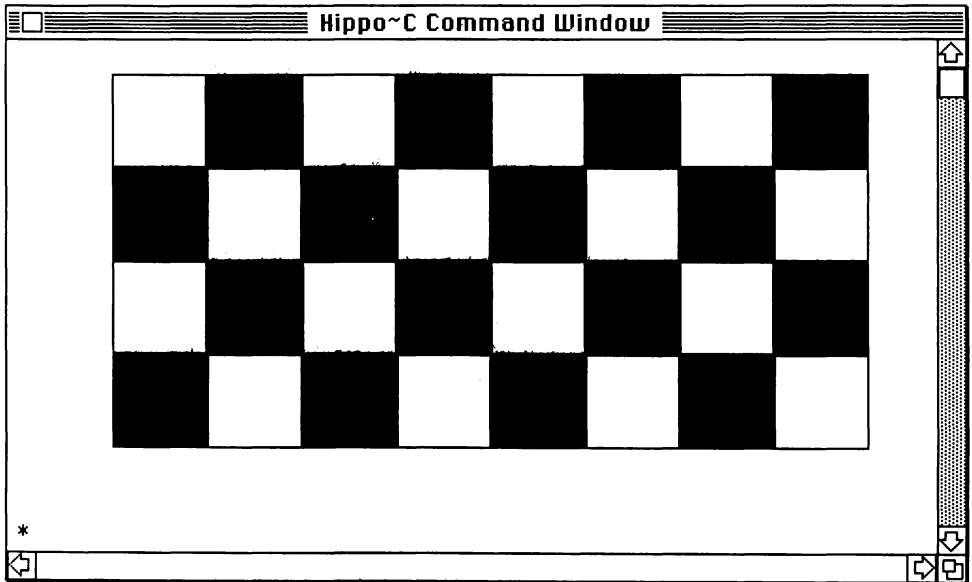


Figure 7.4 Output of cboard.c

Functions and Two-Dimensional Arrays

You've seen how to write a function that takes a one-dimensional array name as an argument. How would you go about writing one that takes a two-dimensional array name as an argument? The trickiest point is declaring the argument. For instance, suppose we add a function to the preceding program, one called `shrinkboxes()` that serves to shrink all the boxes down. The function call would look like this:

```
shrinkboxes (boxes, dh, dv);
```

The first argument (`boxes`), being an array name, is a pointer-to-array-of-eight-rect-structures. How do we declare that mess? Let's start by calling the formal argument `bp`. Then, to indicate it is a pointer, we would use the expression `*bp` in the declaration. It points to an array of eight things; that

expands the expression to `(*bp)[8]`. (The parentheses are used to counteract the higher precedence of the brackets.) The "thing" is a `rect` structure, so the entire declaration becomes this:

```
struct rect (*bp)[8];
```

The remaining arguments, which represent the size adjustments, are short integers and pose no problems.

Here is a revised version of the last program. It uses not only a `shrinkboxes()` function, but a `drawboxes()` and an `invertovals()` function; all use the same form of argument.

```
/* sqcir.c -- makes a pattern of squares and circles */
#define TOP 50
#define LEFT 56
#define SIZE 50          /* sides of individual boxes */
struct rect {short top,left,bottom,right;};
main()
{
    static struct rect screen = {0,0,512,342} ;
    struct rect boxes[4][8];          /* 4 rows, 8 columns */
    int row, col;
    void wait(), drawboxes(); shrinkboxes(),
    invertovals();

    eraserect(&screen);                /*clear screen */
    for( row = 0; row < 4; row++)      /* for each row */
        for (col = 0; col < 8; col++) /* and each column */
            setrect(&boxes[row][col], LEFT + col*SIZE,
                    TOP + row*SIZE, LEFT + (col + 1)*SIZE,
                    TOP + (row + 1)* SIZE );
    drawboxes(boxes);
    shrinkboxes(boxes, SIZE/4, SIZE/4);
    invertovals(boxes);
}

void drawboxes(bp)
struct rect (*bp)[8];
{
    int row,col;
```

```

    for( row = 0; row < 4; row++)
    {
        for ( col = 0; col < 8; col++)
        {
            if( (col+row) % 2 == 0)
                framerect(&bp[row][col]);
            else
                paintrect(&bp[row][col]);
            wait(15);
        }
        wait(30);
    }
void shrinkboxes(bp, dh, dv)
struct rect (*bp)[8];
short dh, dv;
{
    int row,col;

    for( row = 0; row < 4; row++)
        for ( col = 0; col < 8; col++)
            insetrect(&bp[row][col], dh, dv);
}

void invertovals(bp)
struct rect (*bp)[8];
{
    int row,col;

    for( row = 0; row < 4; row++)
        for ( col = 0; col < 8; col++)
            invertoval(&bp[row][col]);
}

```

Figure 7.5 presents the output.

As usual, there are some points to note. First, since the variable `bp` used in the functions is the same type as the array name `boxes` used in `main()`, it can be used in the same manner. Thus, `bp[row][col]` is, like `boxes[row][col]`, a particular structure in the array. Therefore, we can use the expression `&bp[row][col]` to pass the address of that particular structure to the Quickdraw routines. If you find this point a bit obscure, don't worry, we'll come back to it later.

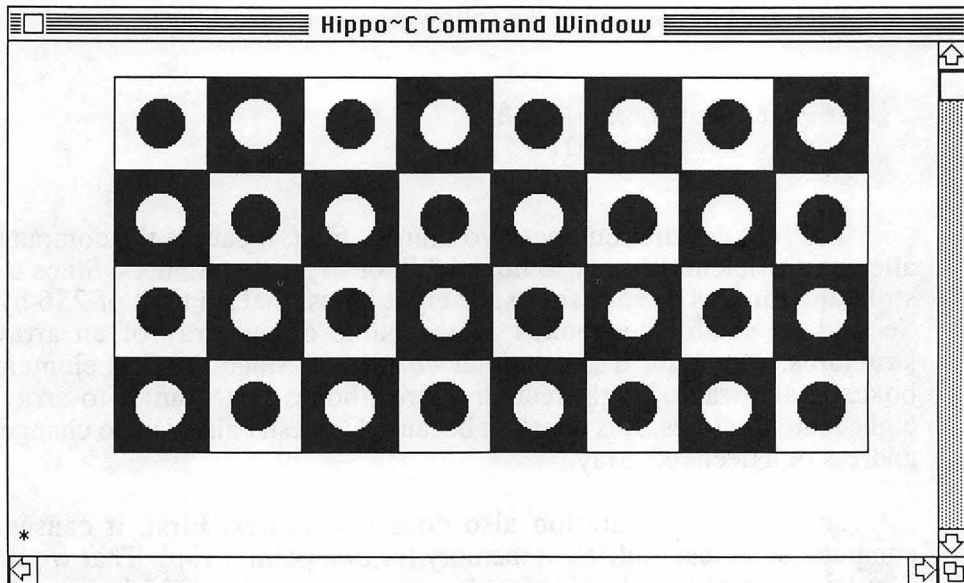


Figure 7.5 Output of `sqcir.c`

Second, all four functions in our program (including `main()`) make use of the `rect` structure definition. It is for cases like this that using an external template definition is worthwhile.

Third, the declaration

```
struct rect bp[][8];
```

is supposed to be equivalent to the one we used; however, it didn't work with the Hippo C version we used.

Fourth, note that the Quickdraw `invertoval()` function can be used without prior drawing of an oval. The function inverts those points within the conceptual oval bounded by the current values in the indicated `rect` structure.

Pointer and Array

In `shrinkboxes()`, `bp` is used in the same manner as `boxes` is in `main()`, yet the declarations appear (and are) different. It all works out for reasons we've discussed already, but it won't hurt to review the relevant facts.

Let's start by looking at the following two declarations:

```
struct rect boxes[4][8];  
struct rect *bp[8];
```

The first declaration does two things. First, it causes the computer to allocate sufficient memory to hold $4 * 8$, or 32, rect structures. Since a rect structure consists of four shorts, or eight bytes, that's a total of 256 bytes. Second, it establishes boxes as the name of an array of an array of structures. As such, it is a pointer *constant*. Since the first element of boxes is an array of eight rect structures, boxes is a pointer-to-array-of-eight-rect-structures. It is constant because C doesn't allow us to change the address of a declared array.

The second declaration also does two things. First, it causes the computer to allocate sufficient memory for *one* pointer (bp). That would be 4 bytes. Secondly, it establishes bp as a pointer *variable* that is of type pointer-to-array-of-eight-rect-structures.

Comparing types for bp and boxes, we see they are exactly the same. That is why they can be used in the same manner. One difference is that boxes is associated with an actual array, while bp starts out as a pointer without an associated array. But bp is a variable, and the function call shrinkboxes(boxes) serves to assign boxes' value (the beginning of the array) to bp, so that bp now becomes associated with the boxes array. Thus, bp[2][4] would designate the same structure as boxes[2][4]. The first uses a pointer variable to indicate which array, and the second uses a pointer constant (of the same value) to indicate the same array.

Complex Declarations and typedef

We have seen a few less than simple types, so perhaps it is a good time to look at the topic of making declarations. The aim of a declaration is to associate a name (more technically, an identifier) with a particular type of data form. Sometimes all that is needed is a fundamental data type and an identifier, as in the next declaration:

```
char cuterie;
```

To this basic form can be added various modifiers, such as unsigned, storage class keywords, the indirection operator (*) to indicate a pointer, brackets to indicate an array, and parentheses to indicate a function. We'll look at the last three now.

The indirection operator can be used alone or repeatedly. Consider these declarations:

```
char *frip;  
char **fnip;
```

The first, as you well know by now, says frip is a pointer-to-char. The second declares that fnip is a pointer-to-pointer-to-char. The pointer-to-pointer form is common in Macintosh application programming. In that context, it even has its own name, to wit, the "handle."

Because the handle is important to the Macintosh, let's take a quick look at using a handle. Suppose we have this statement:

```
**fnip = 'H';
```

We can paraphrase the statement this way. "Go to the memory location called fnip. In there, you will find an address stored. Go, then, to that address in memory. There you will find another address stored. Go to that address, and stuff an 'H' into it." This is an example of "double indirection". It may seem like a bunch of unnecessary trouble, but it turns out to be needed to work effectively with the Macintosh "heap" system of memory management. We'll say more about that in Chapter 10.

The bracket pair, too, can be used once or several times. You've experienced one- and two-dimensional arrays, and the process can be extended to three-dimensional arrays and beyond, should you need them.

Mixed Modifiers

Declarations become trickier when you use more than one type of modifier. Fortunately, the rules of precedence let us unravel the meanings. There are two points to keep in mind in interpreting declarations. First, modifiers closest to the identifier are applied first. Second, brackets and

parentheses have a higher precedence than the indirect value operator. Let's see how this works out with a few examples.

Consider these declarations:

```
char *fez[4];           /* array of 4 pointers to char */
char (*fuz)[4];         /* a pointer to array of 4 char */
```

In the first case, the `*` and the `[4]` are equidistant from `fez`, so the precedence order says to apply the bracket modifier first. Thus, `fez` is an array of 4 somethings. Next, we apply the `*` operator, learning that `fez` is an array of 4 pointers. Finally, `char` tells us that `fez` is an array of 4 pointers to `char`.

In the second example, grouping parentheses tell us to apply the `*` modifier first. Thus `fuz` is a pointer to something. Next, we go to the brackets, and then to `char` to get the rest of the declaration. Notice that the parentheses make a significant difference. The first declaration creates *four* pointers in *one* array, while the second creates *one* pointer and *no* array.

Similarly, we have the following two examples:

```
char *bear();           /* function returning pointer-to-char */
char (*boar)();          /* pointer to function returning char */
```

Our example of function return values up to now have been basic types, but functions can also return pointers to various types. Here, `bear()` is a function returning a pointer-to-`char`.

The second example introduces the concept of a pointer to a function. A pointer to a function can be used as an argument to another function to tell that function what function it can use. We'll supply a brief example later.

So far operator precedence was all that we needed to interpret a declaration. Let's go beyond that now. Look at these examples:

```
char *groucho[2][4];    /* array of pointers to
                           array of char */
```



```
char (*harpo)[2][4]; /* pointer to array of
                      arrays of char */
char *(chico[2][4]); /* array of arrays of
                      pointers to char */
```

The first example illustrates the natural order of modifiers. First, comes [2], then *. They tie for proximity, but brackets have a higher precedence. Then comes [4], because it is more distant from groucho than either of the other two. Applying the modifiers in sequence, we get that groucho is an array of two pointers to an array of four chars. It's a several-step process, but it is logical. The other two examples use grouping parentheses to alter the order of applying the modifiers.

In short, observing the rules of proximity and precedence should allow you to construct and interpret the declarations of C.

typedef

Although the rules let us interpret declarations unambiguously, the meaning of a declaration is not always quickly obvious. Then, too, if you have to declare a complex type several times, you increase the odds of making an error. To simplify and clarify declarations, C offers the *typedef* facility. It allows you to create a convenient abbreviation for a complex type.

Here is how it works. Suppose you want the term "string" to mean pointer-to-char. Then you declare string as if it were a variable of type pointer-to-char, but you precede the declaration with the keyword *typedef*:

```
typedef char *string;
```

The presence of *typedef* instructs the compiler that string identifies a *type*, not a *variable*. You can then use string to declare variables:

```
string catname, bossman;
```

This states that `catname` and `bossman` are both type pointer-to-char. In other words, this declaration has the same effect as the following:

```
char *catname, *bossman;
```

Similarly, if you wanted to create a type identifier for a pointer to an array of 10 `rect` structures, you could do this:

```
typedef struct rect  (*pointrects) [10];
```

Then the declaration

```
pointrects someboxes;
```

would mean that `someboxes` is a pointer to an array of 10 `rect` structures.

In short, to create a type identifier, declare it as if it were a variable identifier, and precede the declaration with `typedef`.

The scope of a `typedef` is the range of a program over which the `typedef` definition is recognized. The scope rules are the same as for variables. A `typedef` set up within a function is local to the function, and one set up external to any function is global.

Often `typedef` definitions are placed in a header file which then is brought into a program via the `#include` directive. For example, we did not have to define the `rect` structure in our Hippo C programs. Instead, we could have begun our programs with this directive:

```
#include "data.h"
```

This vast file includes the following two definitions:

```
typedef short integer;
typedef struct
{
    integer top, left, bottom right;
} rect;
```

The first definition allows you to use the word `integer` instead of `short` to declare short ints. The reason for this is that Quickdraw is based on Macintosh Pascal, in which the basic Pascal integer type is a 16-bit integer. (To provide a 32-bit integer, Macintosh Pascal uses a nonstandard type called `longint`, which corresponds to the Hippo C `int`.)

The second definition lets `rect` denote the structure type we used. Thus, instead of making declarations like

```
struct rect box;
```

we could have used `#include data.h` and declarations like this:

```
rect box;
```

This, too, makes the declarations look more like the Pascal equivalents.

Function Pointers

Aside from this section, this book doesn't use function pointers, but because we have shown you how to declare a pointer to a function, it is only fair that we show you how to use one. In C, a function can be passed a function pointer telling the first function which function to use. This may sound odd, but it is not too different from passing an array pointer to tell a function which array to use.

Let's look at the mechanics. First, the name of a function serves as a pointer to the function, just as the name of an array serves as a pointer to the array. Normally, this is the form of function pointer that would be used as the actual argument in a function call. Second, a function whose name is used as a function call argument must be declared. Third, the function pointer used as the formal argument must be declared as a pointer-to-function of the proper type.

An example should make these points clearer. Here is one in which the `diff()` function takes three arguments: a function pointer, and two ints. It evaluates the pointed-to function for the two integers and returns the difference:

```
/* ptfun.c -- uses a function pointer */
main()
{
    int answer;
    int square(); /* declare pointed-to function */

    answer = diff(square, 2, 10);
    printf("The answer to the Great Question is %d\n",
        answer);
}

int square(n) /* define a squaring function */
int n;
{
    return n * n;
}

/* here comes the function that uses
   a function pointer */
int diff(f, x, y)
int (*f)(); /* f a pointer to an int-returning
            function */
int x, y;
{
    return (*f)(y) - (*f)(x);
}
```

Here's what happens when `diff()` is called. It is passed the identifier `square`, which serves as a pointer to the `square()` function. This pointer is assigned to `f`, which is a pointer to a function that returns type `int`. Because `f` is a pointer to the function, the value of `f` is the function, so the expression `(*f)(x)` is interpreted to be `square(x)`. Since `x` is 2, the expression evaluates

to 4. Similarly, `(*f)(y)` means `square(y)`, which evaluates to 100. Thus the `diff()` function returns 96 in this case.

We had to declare `square()` even though it is of type `int`. Normally, C uses the following parentheses pair to tell that a name is a function name, but the parentheses are omitted when the name is used as a pointer. Explicitly declaring `square()` lets the compiler know that `square` is a function pointer, and not, say, an undeclared variable name.

Within `diff()`, `(*f)()` serves as a function. Some within the C community don't like this usage, arguing that `f()` should be used instead. This would make the usage more like that for arrays. Some compilers accept both forms.

Note that the actual argument (`square`) and the formal argument (`f`) describe functions that must agree in the type of return value and in the number and types of arguments.

The value of a function like `diff()` is that it can be used with a variety of functions. For example, suppose we needed to know differences between the values of several functions. Then `diff()` could be called several times, using a different function name in the argument list each time.

Summary

C allows you to construct data forms of arbitrary complexity. You can define **arrays of structures, structures of arrays, structures of structures, and arrays of arrays**, to give just a few possibilities.

An array of structures is declared like this:

```
struct rect boxes[5];
```

This allots storage for 5 structures of the `rect` type. The array name, `boxes`, is a pointer to the first of the 5 structures. The array elements are structures called `boxes[0]`, `boxes[1]`, and so on. You can use the membership operator with these structure names to obtain individual members. Thus, `boxes[0].top` would be the top member of the first structure.

Here is a set of declarations that creates a structure containing another structure and an array:

```

struct date {
    int month, day, year;
};      /* a structure template */
struct report {
    struct date duedate;
    int monthsales[12];
} defargo;

```

To access members of a structure within a structure, use the membership operator twice. Thus, `defargo.duedate.year` would be the year member of the structure `defargo.duedata`, which, in turn, is a member of the structure `defargo`.

To access array elements, use expressions like `defargo.monthsales[4]`; here `defargo.monthsales` serves as an array name, so adding the brackets and subscript yields an array element.

An array of arrays, or a two-dimensional array, can be created by using two sets of brackets. Thus, the declaration

```
int sales[10][12];
```

creates an array with 10 elements, each element of which is an array of 12 ints. In this construction, `sales` is a pointer to the subarray `sales[0]`, which is the first array of 12 ints. The identifier `sales[0]`, in turn, is a pointer to the element `sales[0][0]`. Individual elements are accessed using two subscript values. Thus, the fifth element in the third array is `sales[2][4]`.

The C **typedef** facility simplifies making repeated complicated declarations by establishing a one-word identifier to represent any combination of type identifiers and modifiers. To use this facility, declare the identifier as if it were a variable of the desired type, and precede the declaration with the keyword **typedef**. For example, the following statement establishes `rectpoint` as an identifier for the type pointer-to-struct `rect`:

```
typedef struct rect *rectpoint;
```

This definition enables you to use declarations such as

```
rectpoint pr1, pr2;
```

instead of the following:

```
struct rect *pr1, *pr2;
```

This is particularly convenient when the declaration form has to be used several times.

8

Character Strings

In this chapter you will learn about:

- The C string format
 - String constants
 - String variables
 - Character arrays and pointers
 - String I/O
 - String functions
 - The Macintosh Pascal string format
 - Quickdraw string functions
-

Many programs deal in part or in entirety with data in the form of words and phrases. For instance, an interactive program might ask you for your name so that it can address you by name later. C handles this kind of data with a data form called a "character string," or, for short, just "string." The Macintosh Toolbox also uses character strings for data of this kind. As we mentioned long ago in a distant chapter, the Toolbox character string is different from the C character string, but simple functions allow us to convert one form to the other.

In this chapter we will begin by studying C strings and examining some standard C library functions that deal with strings. Then, sensibly enough, we'll move on to the Toolbox treatment of strings.

Character Strings

As with other data types, there are both constant and variable forms of C character strings. Both constants and variables are stored the same way, so let's begin with that.

A **character string** is a series, or string, of characters stored sequentially in memory. Different strings require different amounts of memory, so a program needs some way to keep track of how long each

string is. The method used in C to do this is to mark the end of a string with a special character. The special character chosen for this role is the "null" character. This is the character whose ASCII code number is 0. In character notation, it is written `'\0'`. Don't confuse it with the digit 0 *character*, which is 48 (decimal) in ASCII code.

String Constants

Character string **constants** in C are written as a string of characters enclosed in quotes. We've often used them as arguments for `printf()`. Here are two examples:

```
printf("Hi, guy!\n");  
printf("x = %d\n", x);
```

The first string is an ordinary message. The second string contains formatting instructions for `printf()`; but it, too, is a string, for it consists of a bunch of characters enclosed in double quotes.

When writing a string using double quotes, you don't include the null character. The compiler uses the closing double quote to detect the end of the string, and it then inserts the null character when storing it. For example, Figure 8.1 shows how the first `printf()` argument above is stored.

`"Hi, guy!\n"`

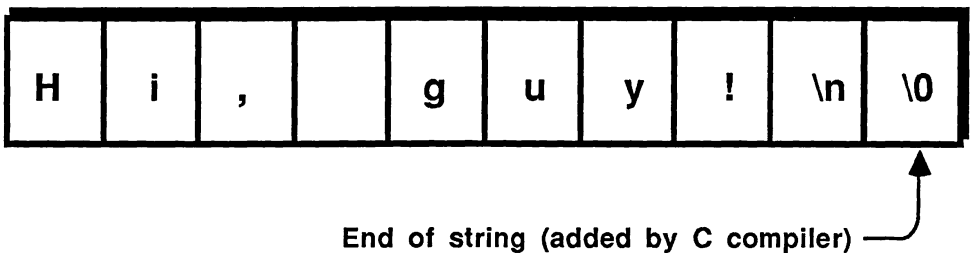


Figure 8.1 Storing a string constant

String Variables and Character Arrays

When you read about how a string is stored, you probably were reminded of arrays. Good, for a *character array* is one way to represent a string. By now you know quite a bit about arrays, so we don't need to spend much more time on the topic. There are a couple of points to examine, however.

First, how do you assign a particular string to an array of char? One method, which we will return to later, is to use input functions to read a string in from the keyboard. A second method is to initialize the array when declaring it. This can be done, recall, only if the array is of a static or external storage class. Here is an example:

```
static char msg[6] = {'N', 'o', ' ', 'g', 'o', '\0'};
```

This initializes the array to the string "No go". Each character is enclosed in single quotes to identify it as being a character constant and not, say, the name of a variable.

All the quotes and commas make this tough to type. Therefore C offers a simpler initialization form just for strings. It goes like this:

```
static char msg[6] = "No go";
```

When you use this form, the null character is inserted automatically.

Because of the null character, the array must have at least one more element than the number of characters in the quoted string. It's okay to have more elements than that, for the extra array elements just get set to ASCII zero, the null character.

If you don't feel like counting the number of characters in the string, you can make the declaration this way:

```
char words[] = "Fools rush in where wise men fear to  
tread.";
```

The compiler will count up the number of characters, add one for the null character, and allot the appropriate amount of memory. (Because we did not use the keyword `static`, this declaration would have to be made externally.)

Another point to note is that not all char arrays are strings. Consider this example:

```
static char name[4] = {'f','i','d','o'};
```

This may be a string of characters, but it is not a character string, for it does not contain the null character. A program may use this form when it needs to store several unrelated characters in an array.

Finally, the initialization forms we've shown can only be used in a declaration. Consider the following:

```
static char words[10]; /* this part is fine */

words[5] = '!';        /* so is this */
words = "Try it";      /* but this is no good */
words[] = "Oh yeah!";  /* and neither is this */
```

The first assignment statement assigns a char constant ('!') to a char variable (`words[5]`) and is perfectly valid. The second assignment statement tries to assign a string to `words`. But `words`, being an array name, is a pointer constant, and cannot be assigned any kind of value. In the final statement, `words[]` is meaningless in C except in a declaration statement.

String Variables and Pointers

Values *can* be assigned to pointer variables, and that offers another way to handle strings. Consider this program fragment:

```
char *pc;      /* pc is a pointer to char */
static char words[] = "This works."; /* words is array
                                         of 12 char */

pc = words;
pc = "This is a valid operation.";
```

The first declaration creates a pointer `pc` that be can assigned the address of any `char` value. The second declaration creates an array and initializes it to the indicated string. The next statement assigns the address of the first element of the `words` array to `pc`. This is valid because `words`, too, is a pointer-to-`char`. (Yes, once again the name of an array is the address of its first member.) The final statement is also valid; it assigns the address of the beginning to the string "This is a valid operation." to `pc`.

The last example uses the fact that in C a quoted string is a pointer. Just as `words` pointed to its first element, so "This is a valid operation." points to its first element. The whole quoted string acts like an array name. To show that we are not joshing you, here is an example that prints addresses:

```
main()
{
    static char ho[] = "Ho ho ho!";

    printf("array: %u; string %u\n", ho, "Ho ho ho!");
}
```

And here is the output:

```
array: 36136; string 36168
```

Note that the two strings are stored in different locations, even though they have the same contents.

A pointer-to-`char` can also be initialized to a quoted string; in that case storage is created for the pointer, and then the address of the quoted string is assigned to it. The next section contains an example.

String I/O

Now that we have seen how to set up strings within a program, let's turn to *input* and *output*. C offers several functions for string input and output. The output functions are slightly simpler to use, so let's start with them.

String Output: printf()

We've used `printf()` many a time to print a string constant, but it can also be used to print string variables. The method is to make use of the `%s` format specifier in the control string. Just as `%d` is a place holder for an integer, `%s` is a place holder for a string. When the compiler finds a `%s` in the control string, it expects to find a *pointer* to a string in the argument list. Since arrays of char names, declared pointers-to-char, and quoted string constants all are pointers to strings, they all can be used as function arguments for `printf()`. The next little program demonstrates these points:

```
main()
{
    static char m1[] = "I am";    /* initialized array */
    char *m2 = "what I";         /* initialized pointer */

    printf("%s %s %s be.\n", m1, m2, "must");
}
```

Here is the output:

```
I am what I must be.
```

In the `printf()` argument list, `m1`, `m2`, and `"must"` all are string pointers. Note that we used spaces within the control string to provide spaces between the separate strings. Also, we initialized the pointer `m2` when we declared it.

The `%s` format is executed in an interesting fashion. The actual argument is a pointer to the beginning of the string. The computer goes to that location, prints the character it finds there, goes on to the next location and prints it, and keeps on with this process until it finds a null character. If the argument points to a character array that is not a string (i.e., to one lacking a null character), the computer will keep on going, printing byte after byte until it finds a null character somewhere in your program. That might be the very next byte, or it might be a long way.

You can use a field width specifier and the left-justification specifier (the - sign) with the %s format, just as we did with the other formats in Chapter 3. Here are some examples:

```
main()
{
    char *msg = "Tennis, anyone?";

    printf("\"%s\"\n", msg);
    printf("\"%20s\"\n", msg);
    printf("\"%-20s\"\n", msg);
}
```

The quotes inside the control string (the \"s) aren't necessary; we inserted them to show the size of the printing field. Here is the output:

```
"Tennis, anyone?"
"      Tennis, anyone?"
"Tennis, anyone?   "
```

String Output: fputs()

The printf() function is quite versatile. It allows you to print strings, characters, and numbers in a specified format. Because it does so much, it is not that efficient for any one task. Therefore C offers another output function, fputs() that only prints strings. It is much more limited than printf(), but it is more efficient at the one thing it does.

Like printf(), fputs() uses a string pointer as an argument to tell it what string to print. Unlike printf(), fputs() can handle only one string at a time. Also, fputs() takes a second argument that tells it in *which* file to put the string. We haven't discussed files yet, but all we need to know at this point is that stdio.h defines **stdout** to be the standard output, which, by default, is the screen. Here is an example showing its use:

```
#include "stdio.h"
main()
{
    char *m = "Is this clear?";
```

```

    fputs(m, stdout); /* send string m to the screen */
    fputs(m, stdout);
}

```

And here is the output:

```
Is this clear?Is this clear?
```

Note that `fputs()` does not start a new line unless you tell it to by placing a newline character in the string.

If you just need to print strings, `fputs()` is more efficient than `printf()`. But if you need to format a string or combine a string with other output, you probably will need to use `printf()`.

String Output: puts()

Many C compilers (but not Hippo C) offer a function called **puts()** that differs from `fputs()` in two respects. First, it doesn't take a file argument, for it always uses the standard output. Second, it automatically inserts a newline character at the end of the string when printing it. Hippo C's `stdio.h` contains this macro:

```
#define puts(s) fputs(s, stdout)
```

This version of `puts()` doesn't require a file argument, but it doesn't put in the newline character.

It's not that difficult to write a function that behaves like the standard `puts()`, and doing so would give us more experience with strings and pointers. So let's do it.

First, let's think about what's needed. (This important step is neglected surprisingly often.)

1. The function should take a pointer-to-char as an argument (call it `pc`).
2. It should print the pointed-to character (that would be `*pc`).

3. Then it should advance the pointer to the next character (`pc++`).
4. It should continue printing characters as long as the pointer doesn't point to a null character (`while (*pc != '\0')`).
5. Then it should print a newline and end.

We've practically written the program. Here is a first draft:

```
puts(pc)
char *pc;
{
    while ( *pc != '\0')
    {
        putchar(*pc);
        pc++;
    }
    putchar('\n');
}
```

The draft is fine, but we can make some C-like shortcuts. First, the line

```
while ( *pc != '\0')
```

can be replaced by this:

```
while ( *pc )
```

Remember, a while loop continues as long as the test expression is nonzero. Since `*pc` is the ASCII value of the pointed-to character, it is nonzero unless it is the null character. Thus the loop stops when the null character is reached. This construction is common in string-processing functions.

Second, we can combine the two loop statements (printing and incrementing) into one. The next test program incorporates these changes.


```

main()
{
    char *m = "I am a rock.";

    puts(m);
    *(m + 7) = 's'; /* value of pointer to 8th element */
    puts(m);
}

puts(pc)
char *pc;
{
    while (*pc)
        putchar(*pc++);
    putchar('\n');
}

```

To add a tad of variety, we changed one of the string characters between puts() calls. Here is the output:

```

I am a rock.
I am a sock.

```

Note that puts() succeeded in giving each string its own line.

Note that we have used the compact pointer notation favored by C programmers. For comparison, here is the more pedestrian array version:

```

puts(s)                                /* array version */
char s[];
{
    int index = 0;                      /* array subscript */

    while ( s[index] )                  /* while s[index] not a null
        putchar( s[index++] );          character */
    putchar('\n');                      /* print and advance index */
}

```

It has to use one more variable, the array subscript. Instead of moving the pointer `s` from one character to the next, it keeps `s` fixed and varies the subscript. Aside from that, `s[i]` plays the same role that `*s` did in the preceding version.

String Input: scanf()

The multipurpose `scanf()` function uses the `%s` format to read strings. Reading a string is trickier than printing one. When the computer prints a string, it has the null character to tell it when to stop. But, for input, there is no exact equivalent to the null character. Therefore, the input function design has to use some other sign to indicate the end of a string. `scanf()` uses "whitespace" (a space, tab, or newline) to indicate the boundaries of a string. Thus, `scanf()` begins reading a string at the first nonwhitespace character and continues reading until the next whitespace character. The argument matching the `%s` format, as for the other formats, should be a pointer to the target storage area. `scanf()` places the input into the target storage area and appends a null character, making it into a C string. Here is a sample usage:

```
main()
{
    char name[10];    /* target array */

    fputs("Enter a name:\n");
    while ( scanf("%s", name) == 1 )
    {
        printf("Hello, %s, and good luck.\n", name);
        fputs("Enter another name:\n");
    }
}
```

Recall that `scanf()` *always* takes a pointer argument to indicate where the input should be placed. Because `name` is an array name and a pointer, we don't need to use the address operator in the `scanf()` argument; `name` already is an address. Here is a sample run:

```
Enter a name:
Fritzi[RETURN]
Hello, Fritzi, and good luck.
Enter another name:
```

```
Cleo[SPACE>Hello, Cleo, and good luck.  
Enter another name:  
[OPTION]-d
```

Notice that the moment we struck a whitespace key, the program terminated reading the string and went on to the next statement. Remember, Hippo C uses unbuffered input. A buffered system would still read in Cleo but wouldn't get around to processing it until a [RETURN] was struck.

There are some important facts about `scanf()` you should know. One, which should be obvious from the example, is that the `%s` format just reads in a single word. It can't read a string containing spaces because it stops at the first space.

Second, space must be allotted to hold the input string. Do not emulate the following example:

```
char *pc;  
  
scanf("%s", pc); /* big trouble */
```

There are two problems here. First, the declaration creates storage for `pc`, a pointer. Thus it sets aside one 4-byte memory unit suitable for storing an address. It doesn't create storage space to hold a string. Second, although it creates storage for `pc`, it doesn't assign a value to `pc`. Thus, when the program executes the `scanf()` statement, it will attempt to place the input string at whatever address happens to be lying in the `pc` memory location. Chances are the program will crash quickly.

Not only should there be storage, there should be enough storage. What happens if, say, you respond with the name Theophilocrates? If you do, `scanf()` will read in the whole word. When it runs out of array space, it will keep going into the memory cells following the array. This may wipe out other data or even overwrite program code, depending on the storage class and the compiler.

There is a way to prevent this disaster. You can use a field width modifier in the %s format. Then, scanf() will read to the end of the field or to the first whitespace character, whichever comes first. The field width should be one less than the array size in order to leave room for the terminating null character. Here is a modified version of the last program:

```
main()
{
    char name[10];    /* target array */

    fputs("Enter a name:\n");
    while ( scanf("%9s", name) == 1 )
    {
        printf("Hello, %s, and good luck.\n", name);
        fputs("Enter another name:\n");
    }
}
```

And here is sample output:

```
Enter a name:
TheophilocHello, Theophilo, and good luck.
Enter another name:
rates[SPACE]
Hello, crates, and good luck.
Enter another name:
[OPTION]-d
```

Once the "c" of "Theophilocrates" was typed, scanf() knew it had enough. It processed the first nine letters, and after the rest of the name was typed, the program processed the rest of the name. Note that the second call of scanf() resumed where the first left off, at the letter c.

String Input: fgets()

The scanf() function reads one word and converts it to a C string. The fgets() function reads a whole line and converts it to a C string. The signal it uses to mark the end of an input string is the newline character generated by the [RETURN] key.

The `fgets()` function takes three arguments. The first is a pointer-to-char to identify where the input should be placed. The second is an int that specifies the maximum allowable size of the input string. The function will read up to one less than the maximum size (leaving space for the null character to be added) or up through the first newline, whichever comes first. The third argument indicates the source file. All we need to know at this point is that the file `stdio.h` defines `stdin` as the standard input, which, by default is the keyboard. Here is a sample use:

```
#include "stdio.h"    /* needed for stdin, stdout
                        definitions */
#define MAXSIZE 70
main()
{
    char phrase[MAXSIZE];

    fputs("Enter a phrase:\n", stdout);
    fgets(phrase, MAXSIZE, stdin);
    fputs(phrase, stdout);
    fputs(phrase, stdout);
}
```

Here is the output from a run:

```
Enter a phrase:
A glitch in Time saves nine. [RETURN]
A glitch in Time saves nine.
A glitch in Time saves nine.
```

Note that the phrase was reprinted on separate lines. That's because `fgets()` includes the newline character as part of the input string.

We didn't use it in the last example, but `fgets()` has a return value. The function returns its first argument, unless it encounters end-of-file, in which case it returns something called `NULL`. What is `NULL`? First, note that the function `fgets()` must be of type pointer-to-char, if it is to return its first argument. (We got away without declaring the function type because we didn't use the return value.) Thus, `NULL` is a pointer, also. It is, in fact, a constant pointer whose value is 0, that is, it is a pointer to memory address 0. The definition of `NULL` is in `stdio.h`. C is set up to recognize `NULL` as an invalid value for a decent pointer, so it is often used as a negative signal for functions that return pointers. Do not confuse the `NULL` pointer with the null character. Although each has the numerical value zero, the former is

the null character. Although each has the numerical value zero, the former is a memory location, while the latter is a character value.

Here is an example making use of the return value as part of a test condition for a loop. Because we use the return value, we need to declare the function type.

```
#include "stdio.h"    /* needed for definitions */
#define MAXSIZE 70
main()
{
    char phrase[MAXSIZE];
    char *fgets();      /* declare function type */
    fputs("Enter a phrase:\n", stdout);
    while ( fgets(phrase,MAXSIZE,stdin) != NULL);
    {
        fputs(phrase,stdout);
        fputs("More, please.\n",stdout);
    }
}
```

This program will repeat phrases you type until you hit [OPTION]-d to generate the end-of-file signal.

You now know a few ways of getting strings into and out of programs. C offers several library functions to work on strings within a program. We'll look at them next.

String Functions

The Hippo C library contains several useful *string-oriented* functions. They perform such feats as comparing two strings to see if they are the same, copying a string into an array, adding a string to an existing string, and determining the length of a string. Here is a list and brief description of these functions; in it *s* and *t* represent string pointers.

strcat(s,t)	Appends string <i>t</i> to string <i>s</i>
strcmp(s,t)	Compares string <i>s</i> to <i>t</i> , returns 0 if the same
strcpy(s,t)	Copies string <i>t</i> into <i>s</i>
strlen(s)	Returns the length of <i>s</i> , not counting the null character
strncpy(s,t,n)	Copies <i>t</i> to <i>s</i> , maximum of <i>n</i> characters

The functions `strcat()`, `strcpy()`, and `strncpy()` return their first arguments; hence they must be declared type pointer-to-char *if* their return values are used.

The `strcmp()` function returns 0 if the two strings are identical. The comparison is made out to the null characters, so two strings can be the same even if stored in different sized arrays. If the strings are not the same, the difference in ASCII value for the first differing characters is returned. A positive value means the first string comes after the second string in ASCII ordering, and a negative value indicates the opposite order. This fact can be used, for example, in writing a function to put strings in alphabetical order.

The `strcmp()` function is necessary, for the relational operators (`==` and so on) work with single-valued variables, not with strings and other arrays.

Because `strncpy()` may copy only part of the string, it does not necessarily copy the terminating null character. We'll see an example later in this chapter.

The `isalpha()` family of functions we discussed in Chapter 4 are often useful for string handling. Two other useful functions are `tolower()` and `toupper()`. The first converts uppercase to lowercase, and the second does the opposite. In Hippo C, `tolower()` only executes the conversion if its argument is, in fact, uppercase. Not all implementations of this function check first, however, so it is safest to use this function along with the `isupper()` function. A similar situation holds for `tolower()`.

We can use some of these functions to write a simple guessing game program. It prints a question, reads the user's response and uses `strcmp()` to compare the response to the true answer. To facilitate comparison, a `lower()` function uses `tolower()` to convert the whole response string to lowercase. Also, the question and answer strings are expressed as `#defined` constants, so they can be altered easily. Here is the program:

```
#include "stdio.h" /* needed for stdout definition */
#define MAXSIZE 30
#define QUESTION "What talking pig lived on the Bean Farm?\n"
#define HINT "He was a poet, editor, banker, and detective.\n"
#define ANSWER freddy /* A Walter Brooks creation */
main()
```

```

{
    char response[MAXSIZE];
    int len;
    void lower(); /* converts string to lower case */

    fputs(QUESTION, stdout);
    fputs(HINT, stdout);
    scanf("%29s", response);
    lower(response);
    while ( strcmp(ANSWER, response) )
    {
        if ( (len = strlen(response) - strlen(ANSWER) ) > 0 )
            fputs("Your answer is too long.\n", stdout);
        else if ( len < 0 )
            fputs("Your answer is too short.\n", stdout);
        else
            fputs("That's the right length!\n", stdout);
        fputs("Now make another guess:\n", stdout);
        scanf("%29s", response);
        lower(response);
    }
    fputs("Congratulations, that's it!\n", stdout);
}

void lower(s) /* converts string to lower case */
char *s;
{
    while ( *s ) /* cycle until the null character */
    {
        *s = isupper(*s) ? tolower(*s) : *s;
        s++; /* advance to next character */
    }
}

```

The while loop in main() will continue until strcmp() returns 0, which it will do when the correct answer is entered. The strlen() function compares the length of the response to the length of the answer so that the program can help out the user a bit.

The lower() function converts the response to lowercase. That way, even if the user types Freddy or FREDDY or freDDy, the string is converted to freddy before being compared to the answer. Note the construction while(*s). This is a common idiom in string processing functions. We used it earlier in puts(), and we will use it again. The variable s initially points to the first element of the string passed to the function.

Thus, `*s` is the first character of the string. Each cycle of the loop increases `s` by one, making `*s` assume the value of each character in turn. The loop continues as long as `*s` is nonzero. Eventually, `*s` becomes the null character, which has the numerical value of zero, and the loop ends. Within the loop, the conditional operator (Chapter 4) causes `*s` to be converted to lowercase if it is uppercase.

Here is a sample run:

```
What talking pig lived on the Bean Farm?
He was a poet, editor, banker, and detective.
Porky
Your answer is too short.
Now make another guess.
Practical
Your answer is too long.
Now make another guess.
Freddy
Congratulations, that's it!
```

We could have used `fgets()` instead of `scanf()`. If we did, however, we either would have to strip the newline from the string fetched by `fgets`, or else we would have to change the correct answer to `"freddy\n"`. Later, we will show how to strip the newline from an input string.

String Construction

The `strcpy()`, `strcat()`, and `strncat()` functions are often used to construct a single string from smaller pieces. Here is an example showing how they are used:

```
#include "stdio.h"
#define MAXSIZE 70
#define WORDSIZE 15
main()
{
    char sentence[MAXSIZE], word[WORDSIZE];

    strcpy(sentence, "The ");
    fputs("Give me an adjective:\n", stdout);
    scanf("%14s", word);
    strcat(sentence, word);
}
```

```

    strcat(sentence, " gazed sadly upon its priceless ");
    fputs("Give me a noun:\n", stdout);
    strcat(sentence, word);
    printf("%s.\n", sentence);
}

```

Here is a sample run:

```

Give me an adjective:
drunken[RETURN]
Give me a noun:
buffoon[RETURN]
The drunken swan gazed sadly upon its priceless buffoon.

```

First, `strcpy()` place the word `The` in the string `sentence`. Then, calls to `strcat()` added new characters to the string. Note that even though `word` is a 15-element array, only the characters up to the null character are appended. `Strcat()` also has to remove the null character at the end of the old string, for it is the first null character in an array that signals the end of the string.

The functions `strcat()`, `strcpy()`, and `strncpy()` do *not* check to see if the target array has enough room for the incoming string. It's your responsibility as a programmer to check for that.

There is one more function used in constructing strings from fragments. It's called **`sprintf()`**, and, as you might expect, it is closely related to `printf()`. The difference is that it writes its output in a target string instead of on the screen. Its general form is this:

`sprintf(targetstring,controlstring,argument(s))`

Here, `targetstring` is a pointer to where the string is to be placed, and the remaining arguments are as for `printf()`.

Here is an example illustrating the mechanics of using `sprintf()`; we've included a couple macros to help keep them fresh in your mind:

```

#include "stdio.h"
#define MAXSIZE 70
#define WORDSIZE 15
#define PUTS(X) fputs(X,stdout)
#define GETWD(X) scanf("%14s", X)

```

```

main()
{
    char target[MAXSIZE], fname[WORDSIZE],
    lname[WORDSIZE];
    int age;

    PUTS("Enter your first name:\n");
    GETWD(fname);
    PUTS("Enter your last name:\n");
    GETWD(lname);
    PUTS("Enter your age:\n");
    scanf("%d", &age);
    sprintf(target,"%s %s claims to be
    %d!\n",fname,lname,age);
    PUTS(target);
}

```

Here is some output:

```

Enter your first name:
Jack[RETURN]
Enter your last name:
Benny[RETURN]
Enter your age:
39[RETURN]
Jack Benny claims to be 39!

```

Note how `sprintf()` lets you combine character and numerical information in a string. For example, the numerical value for age was converted to the character sequence "39".

Writing C String Functions

You may need string functions beyond those provided by the library. The C string format makes writing string functions relatively simple. For example, a function does not need to know in advance how long a string is; it can process each character in turn until the null character shows up. Because the value of the null character (zero) is also the value that terminates a while loop, it is natural to use a while loop to control string processing, just as we did with the `lower()` function a few pages ago.

For another example of this technique, let's see how the `strlen()` function could be written. To avoid confusion with the library function, let's use the name `stringlen()`:

```
/* stringlen.c -- example of string processing */
int stringlen(s)
char *s;
{
    int length = 0;

    while ( *s++) /* go until null character */
        length++; /* count characters */
    return length;
}
```

Here, nothing within the loop used the value of the character (`*s`), so we could use the increment operator to advance the pointer after the value of the pointed-to character was tested. When `*s` becomes the null character, the loop terminates, and the function returns the value of `length`.

The `index()` Function

Many C libraries (but not Hippo C) contain a function called `index()` or, sometimes, `strchr()`. It tells you if a given character is in a string. More specifically, it takes a string pointer and a character as arguments and returns a pointer to the first occurrence of the character within the string. If the character is missing, the function returns a pointer to `NULL`. Since this is a useful function, let's write our own version.

The function should compare the character to each string element in turn until the character is found in the string, or until the null character shows up. We can use a compound condition in a while loop to accomplish that. The value of the pointed-to character when the loop ends will determine whether a pointer to the character or a `NULL` pointer is returned. Here is one possible implementation:

```
/* index.c -- finds if character ch is in string s */
char *index(s,ch) /* returns a pointer to a character */
char *s;          /* pointer to the string to be
                  scanned */
char ch;          /* desired character */
```

```

{
    while ( *s && *s != ch ) /* not null and not ch */
        s++; /* advance to next array element */
    return *s ? s : (char *) 0;
}

```

First look at the loop condition: `*s` is nonzero, or true, as long as `s` doesn't point to the null character. The second expression, `*s != ch`, is true as long as the pointed-to character isn't `ch`. The body of the loop (`s++`) advances the pointer to the next character in the string. This continues until one or the other condition becomes false. Then the function reaches the return statement.

When the function reaches the return statement, there are two possibilities for `s`. First, it may point to the first occurrence of `ch` in the string `s`. In this case, the value of `*s` is the designated character, hence nonzero, or true. This causes the conditional operator to choose the first value after the `?`; namely, `s`. Thus the function returns a pointer to the first occurrence of the character in the string, as specified.

Suppose, however, the loop went to the end of the string before stopping. In this case, `s` points to the null character, and the value of `*s` is zero, or false. This causes the conditional operator to return its second choice, which is the NULL pointer.

Now all we have to explain is why `(char *) 0` is the NULL pointer. A pointer, we've said, has an address as a value, and the value of NULL pointer is 0; it points to address location 0.

So all the function needs to return is the numerical value 0. The `(char *)` is added for logical consistency. It is a type cast saying that the 0 is the address of a char. Eliminating the type cast doesn't affect the program, but including it shows that the 0 is the same data type as `s`.

If we had included the `stdio.h` file, we could have used `NULL` instead of `(char *) 0`. However, the main program may have already included that file, and we didn't want to waste space by including it twice.

We can write the function using array notation at the cost of using one more variable to handle the subscript. Here is the array version; if you find the pointer version obscure, perhaps comparing the two versions will illuminate matters for you:

```
/* indexa.c -- array version of index.c */
char *indexa( s, ch )
char *s;
char ch;
{
    int i = 0; /* array subscript */

    while ( s[i] != '/0' && s[i] != ch )
        i++; /* next element */
    return s[i] ? (s + i) : (char *) 0;
}
```

Here `s[i]` plays the same role that `*s` did in the pointer version. Instead of increasing the pointer each loop cycle, this version keeps `s` constant and increases the index `i`. Thus, at the end, it returns `s + i` instead of `s`. Since we are trying to make this version very clear, we explicitly compared `s[i]` to the null character.

Using index()

We claimed that `index()` is a useful function, so let's look at some examples.

The `fgets()` function, we revealed, reads in a whole line of input into a string. It identifies the end of the line by the newline character produced when you hit the [RETURN] key, and that newline character is stored as part of the string. Suppose, as is often the case, that you don't want the newline character stored. Then you can use the `index()` function to locate the newline and replace it. Here is a simple, but flawed, approach; it assumes that `name` is a character array and that `pc` is a pointer-to-char:

```
fgets(name, 80, stdin);
pc = index(name, '\n'); /* find the address */
*pc = '\0'; /* change the value */
```

First, `fgets()` fetches the input. Recall that its second argument is a limit to the number of characters to be read. Next, `index()` returns a pointer to the location of the first newline character in `name`. Thus, `pc` is assigned the *address* of the newline in the string. The third line places the null character into that address, making it the new end of the string.

This sounds fine; what is the flaw? The trouble is that the code assumes that the string `name` contains a newline character. Normally it would, but if you enter a string longer than allowed by the second argument to `fgets()`, only the first part of the string gets assigned to the array. In Hippo C, `fgets()` does not begin processing until the [RETURN] key is hit and the newline character transmitted; `fgets()` quits placing characters into the array when it reaches the newline or the specified limit, whichever occurs first. See Figure 8.2. Thus, the program should check to see if the array contains a newline before trying to replace it. This is easily accomplished by using the return value from `index()`. If it is not the NULL pointer, a newline character was found, and the substitution can be made. Therefore we can use an if statement like the following:

```
if ( (pc = index( name, '\n' ) ) != NULL)
    /* found a newline */
    *pc = '\0'; /* substitute a null character for it */
```

Here we've used the common C idiom of combining an assignment statement with a comparison. It assigns a value to `pc`, and then the value is compared to NULL. The code assumes `stdio.h` is included to provide the NULL definition.

What if `pc` is NULL? Then no newline character was found. This implies that there are still some characters hanging around in the input queue. If the program reads any more input, these characters should be cleared away first. We can do this by adding an else clause to the code:

```
if ( (pc = index( name, '\n' ) ) != NULL)
    /* found a newline */
    *pc = '\0'; /* substitute a null character for it */
else /* no newline */
    while ( getchar() != '\n')
        ; /* skip through end of line */
```

The while loop reads and discards input characters until the newline character is reached. This clears the line, giving the next input statement a clean slate to work from.

Input queue before function call:

L	o	r	e	t	t	a		L	a	n	d	o	s	k	a	\n
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Function call takes 12 characters:

```
fgets(name,12,stdin);
```

Array following function call:

L	o	r	e	t	t	a		L	a	n	\0
0	1	2	3	4	5	6	7	8	9	10	11

Input Queue following function call:

d	o	s	k	a	\n
1	2	3	4	5	6

Figure 8.2 Effects of fgets()

Another Example

A common string-handling problem is to copy part of one string to another. Suppose, for instance, that an array contains someone's full name, and you want to copy just the first name into a second array. One approach is to use `index()` to find the first space in the array and to then use `strncpy()` to copy the part up to that space. There are some possible pitfalls to this approach, so let's develop it step-by-step.

We can start by assuming we have these declarations:

```
char full[20], first[20], *pc, *index();  
int n;
```

The first step is to find the space in full, the array holding the full name:

```
pc = index( full, ' ' );
```

This assumes that there is a space; we'll remove that assumption later.

To use the `strncpy()` function, we need to know how many elements to copy. We can figure that out using pointer arithmetic. The array name `full` points to the first element of the array (subscript 0), and `pc` points to the element with the space. The difference gives the number of characters up to, but not including, the space:

```
n = full - pc; /* find number of elements before space */
```

It's easy to make a one-off counting error in these situations, so you should convince yourself that this formula is correct.

Now we can use `strncpy()` to copy the first `n` elements of `full` into `first`:

```
strncpy( first, full, n );
```

Recall that the third argument for `strncpy()` tells how many characters to copy.

Are we done? No, because when `strncpy()` copies less than the whole string, the null character isn't copied. In this case, we know less than the whole string was copied, for everything from the space onward was omitted. This means that `first` lacks a terminating null character, and hence it is not a string. So we need to add on the null character. We copied `n` characters; because array subscripts start at 0, the last character copied was placed in `first[n-1]`. Therefore, we can do this:

```
first[n] = '\0';
```

This places a null character immediately after the last copied character, making first into a proper string.

Finally, we can use an if statement similar to the one in the last example to check to see if a space is found. If one is, procede with the steps we just gave. If not, full contains just one name.

We've put together a program incorporating these programming elements along with the newline example we gave earlier. It also includes a new ploy that we will explain later. The program assumes that index() is in a separate file that is compiled along with the one containing this code. Here is the program:

```
#include "stdio.h"    /* use NULL, stdin definitions */
#define MAX 20        /* use short maximum string for
                        testing */

main()
{
    char full[MAX], first[MAX], rest[MAX];
    int n;
    char *pc, *index();

    printf("Please enter your full name:\n");
    fgets(full, MAX, stdin);
    if ( (pc = index( full, '\n') ) != NULL) /* newline? */
        *pc = '\0';                          /* replace it */
    else
        while( getchar() != '\n')
            ; /* skip to end of line */
    printf("You say %s is your full name.\n", full);
    if ( (pc = index( full, ' ') ) != NULL)
    {
        n = full - pc;
        strncpy(first, full, n);
        first[n] = '\0';
        strcpy(rest, pc);      /* new stuff */
    }
    else                        /* just one name */
    {
        strcpy(first, last); /* copy whole name */
        rest[0] = '\0';      /* set rest to empty string */
    }
}
```

```

printf("Oh! Not %s \"Big Byte\"%s, the famous
programmer!\n",
        first, rest);
}

```

Here is a sample run:

```

Please enter your full name:
Gladys Pips
You say Gladys Pips is your full name.
Oh! Not Gladys "Big Byte" Pips, the famous programmer!

```

It works, and it also handles single name entries, such as Sophocles, and overly long entries, such as Jennifer Elisabeth Longwoman. It is not completely foolproof (it doesn't handle replies like Smith, Joe sensibly), but it is fool-resistant.

The new bit of programming is storing the rest of the full array in the array `rest`. The second argument to `strcpy()` is supposed to be a pointer to the string to be copied. In this case, we used `pc`, which points to the first space in `full`. Thus, the copying starts with the space character instead of the beginning of the array. The pointers used as arguments by `strcpy()` and `strncpy()` don't have to point to the *beginnings* of strings. In this case, we used a pointer to somewhere in the middle of the original string. Similarly, the first argument can be a pointer to the middle of the destination array. In that case, only the array elements at and after that location are affected.

This flexibility of arguments allow `strcpy()` and `strncpy()` to do more than copy one entire string to another. Suppose, for instance, you wish to copy the fifth through tenth characters of the string `s1` (subscripts 4 through 9) into the string `s2`, starting at its third element (subscript 2). You can do this:

```

strncpy( s2 + 2, s1 + 4, 6);

```

Because `s1` points to the first element of the `s1` array, the expression `s1 + 4` points to the fifth element. Thus, copying will start at that element. Similarly, the expression `s2 + 2` points to the third element of the array `s2`, so the fifth element of `s2` will be copied to that location. Finally, the fifth through tenth characters constitute six characters, so 6 is the third argument.

Perhaps it is time to turn to examples more specific to the Macintosh, so let's look at Macintosh Toolbox strings.

Macintosh Pascal Strings

In Chapter 2, we mentioned that the Toolbox used a string format different from the C format. We'll look more closely at that difference now. Standard Pascal uses something called a "packed array of char" to hold strings, but this mechanism has not proven to be all that useful. For example, a Pascal function or procedure designed to work with this form of string will work only for the particular array size chosen by the designer. As a result, many Pascal implementations use extensions to Pascal to handle strings more effectively. Macintosh Pascal is one such implementation; it uses a nonstandard Pascal type called **STRING**.

Suppose the string 'Eureka' (Pascal uses single quotes for strings) is stored in a **STRING** variable. Then, as in C, the characters are stored in adjacent bytes of memory. However, there is no special character to mark the end of the string. Instead, the byte preceding the first character is used to store the length of the string, 6 in this case. Figure 8.3 shows the string storage.

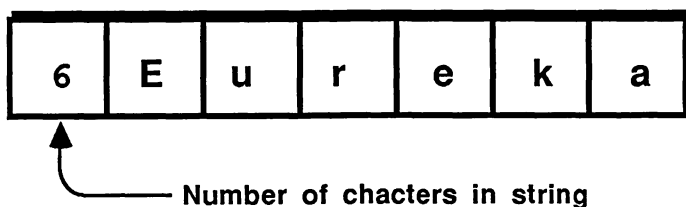


Figure 8.3 Storing a Pascal String

The largest number that can be stored in a byte is 255, so the largest possible Pascal string is 255 characters long. Toolbox functions dealing with strings assume strings are of the maximum size, but because they only process the number of characters indicated in the first byte, the actual strings can be shorter.

String Format Conversion

It's a simple matter to convert from the C format to Pascal and vice versa. For example, to convert from Pascal to C, note the number of characters, move that number of characters one position earlier in the array,

and put a null character at the end. Since each form is one element longer than the number of characters, there is no problem with storage size incompatibilities.

To make the conversion even easier, Hippo C provides two functions to do the actual work. The first is called `strctop()`. It takes a C string as an argument (an array name, a pointer-to-char, etc.), converts the string to the Pascal format, and returns a pointer to the string. Thus the function should be declared type pointer-to-char if the return value is used. The second is called `strptoc()`, and, of course, it converts a Pascal format to a C format in a similar fashion.

An important point about these functions is that both alter the original string. Once you make, say, the call `strctop(myname)`, the string `myname` remains in the Pascal format. If you need the C form later, you will have to use `strptoc()` to convert it back.

A second point is that both functions not only alter the original string, they return a pointer to the string. This gives us two ways in which to use the functions. First, we can call the function, then use the altered string:

```
strctop(myname);           /* convert string */
drawstring(myname);        /* use Pascal format */
```

Or we can use the return value directly to save a line of code:

```
drawstring( strctop(myname) ); /* use return value */
```

In either case, the string `myname` is permanently altered and can be used directly with `drawstring()` henceforth.

A program using the second form should, in principle, declare `strctop()` as type pointer-to-char. You may recall, however, that in Chapter 2 we snuck through an example that failed to do so. Obviously, we didn't want to frighten a beginner by throwing in pointer notation so soon, but why did the program work? Well, because we didn't declare the function, the program assumed the return value was of type `int`. Fortunately, in Hippo C both `int` and pointer types use 4 bytes of memory. The pointer types are unsigned, but as long as an address is less than the maximum signed value (2,147,483,647) a pointer can be successfully read as type `int`. Note: this is an explanation and not a recommendation.

Quickdraw Functions

Output to the screen can be handled by the Quickdraw part of the Toolbox. Here is a list of several of the text drawing functions, including those we mentioned earlier in Chapter 2.

textfont()	Sets font
textmode()	Sets "transfer" mode (overwrite, etc)
textface()	Sets style (BOLD, ITALIC, etc.)
textsize()	Sets type size
drawchar()	Draws a character
drawstring()	Draws a Pascal string
drawtext()	Draws portion of a string
charwidth()	Returns width of character in screen units
textwidth()	Returns width of section of text
getfontinfo()	Supplies information about font

We haven't the space to explain all of these functions in detail. Instead, we will present an example using some of them, and explain these more fully.

From Mac's Toolbox: New Routines

TextWidth	Returns width of a line of text
------------------	---------------------------------

It's been a while since we last used a Toolbox example, so let's mention the Toolbox functions we have used before: `eraserect()`, `setrect()`, `framerect()`, `invertrect()`, `moveto()`, `textsize()`, and `drawstring()`. The example also pulls in a global quantity called **theport**, which we will have to explain, too. Here's the program:

```
/* space_epic.c */
#include "data.h" /* holds many typedefs */
#include "stdio.h" /* defines theport */
#define TIMES 5
#define CYCLES 20
main()
{
    char *title = "WAR OF THE SPACE TWITS";
    int count, i;
    integer left, right, horiz, vert, size, width;
```

```

rect boxes[TIMES];
integer textwidth();

eraserect(&theport->portrect); /* clear window */
left = theport->portrect.left; /* window left */
right = theport->portrect.right
strctop(title); /* convert title format */
vert = 20; /* initialize vertical position */
for(count = 1; count <= TIMES; count++)
{
    size = 9 + count * count; /* font size */
    textsize(size);
    width = textwidth(title,1,title[0]);
    horiz = (left + right - width) / 2;
                                /* centering */
    vert += 3 * size / 2; /* increment vertical
                                position */
    moveto(horiz,vert); /* locate pen */
    drawstring(title); /* draw title */
    setrect(&boxes[count-1],horiz - 1, vert - size -
    2,
            horiz + width + 2, vert + 3);
    framerect(&boxes[count-1]); /* box title */
}
for ( count = 0; count < CYCLES; count++)
    for ( i = 0; i < TIMES; i++)
    {
        wait(20);
        invertrect(&boxes[i]);
    }
}

```

First, let's see what the program does. It prints the stirring title 4 times, each time in larger type. Program instructions center the title horizontally and box each printing. Figure 8.4 shows the appearance of the screen after the first for loop is finished. Then the final portion of the program inverts the boxes in sequence, producing a sense of epic drama. The inversion is repeated several times to impress upon the viewer a sense of ineffable grandeur. Figure 8.5 shows one of the intermediate stages.

Now let's look at some of the details:

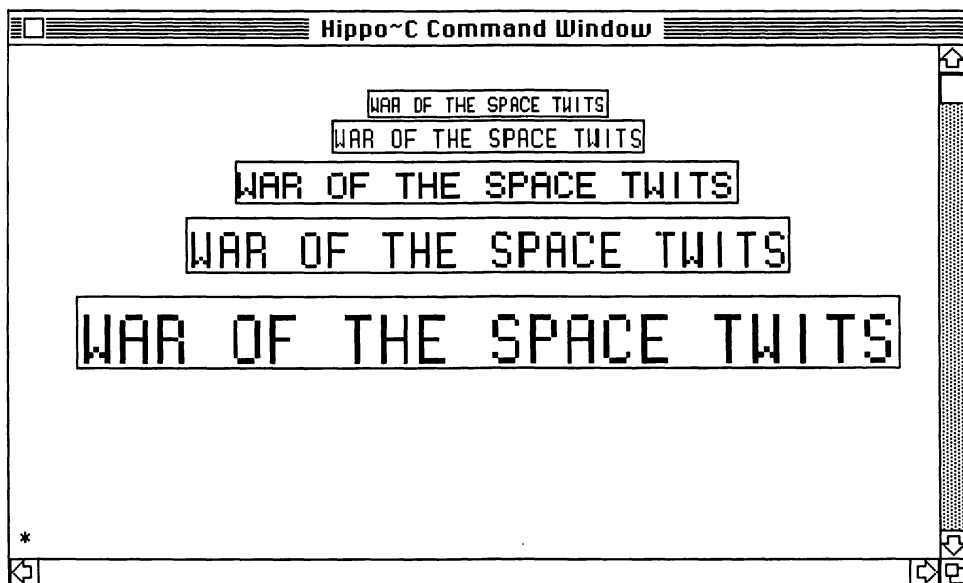


Figure 8.4 `space_epic.c` output

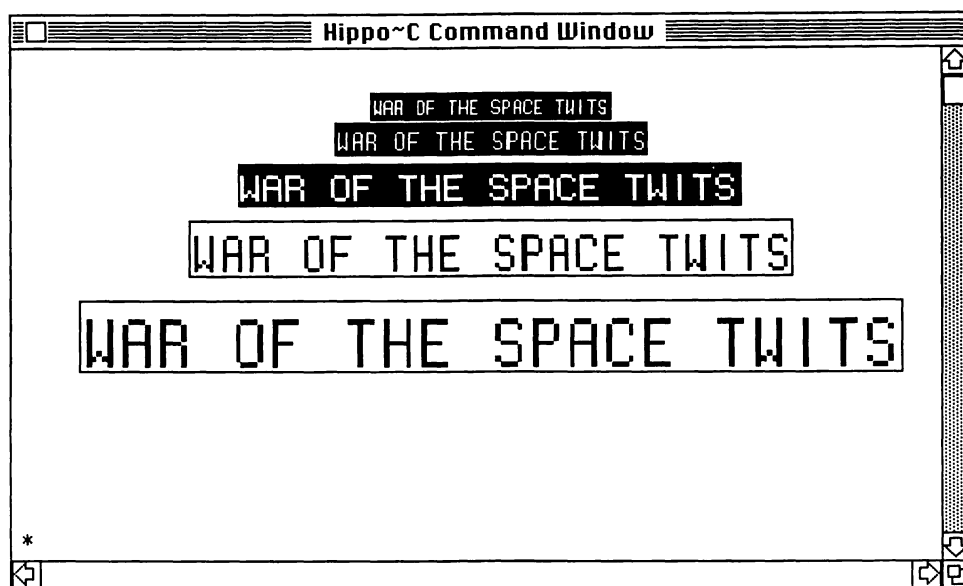


Figure 8.5 Further `space_epic.c` output

theport

Probably the most mysterious element of the program is the unannounced appearance of **theport**. It has this unappealing definition in `stdio.h`:

```
#define theport (*(grafptr *) jt_theport())
```

All we need to know at this point is that **theport** points to a structure that describes the graphics environment. In our case, this structure is set up by Hippo C when it creates the Hippo C and HOS environments. Our program just takes over the environment. In particular, one member of the structure is a `rect` structure called `portrect` that describes the dimensions of the current window. Thus the expression `theport->portrect` is the name of a `rect` structure. This structure contains the boundaries of the current window, so the function call

```
eraserect (&theport->portrect);
```

serves to clear the inside of the window. The indirect membership operator has higher precedence than the address operator, so no parentheses are needed to make the `&` apply to the whole expression and not just the **theport** part. Since the expression `theport->portrect` is a structure name, we can use the membership operator to obtain the left and right limits of the screen, which we did, with expressions like **theport->portrect.left**.

If you must know more about the **theport** definition, here's the lowdown. Clearly, `jt_theport()` is a function, making **theport** a representation of some modification of the return value of the function. What happens to the return value? First, it is subjected to a typecast: `(grafptr *)`. This means the return value is to be interpreted as a pointer-to-`grafptr`. Then the initial `*` yields the value stored at this pointer, meaning the final type (value of pointer-to-`grafptr`) is just `grafptr`. This type, in turn, is defined in the file `data.h` as being a pointer-to-type `grafport`, also defined in `data.h`. This type is a structure storing information about the graphics environment. We'll take up `grafport` further in Chapter 9. Meanwhile, the net result, as we indicated, is that **theport** is a pointer to a structure existing outside of our program. This structure was set up early when the Hippo C environment was created, and now our program takes over that environment.

Functions in the Program

The `strctop()` function makes an early appearance, converting the title to a string in the Pascal format.

We use `textsize()` as we did in Chapter 2, providing it with an integer value for the font size. The unit is the "point," a typesetting unit of approximately 1/72 inch. However, an argument of 0 means to use the system size (generally 12 points). The font size describes the vertical distance from the bottom of a line to the bottom of the next line. Any integer size is acceptable, but the results can look poor if the size does not correspond to a font size in stock or to a multiple thereof.

On the Macintosh, the point is taken to correspond to the "pixel" unit that describes screen *resolution*. Recall that the screen is 512 units across by 342 units down. A pixel has dimensions of 1 unit by 1 unit, and it is the smallest picture element on the screen. When fonts are scaled to various sizes, the results don't always come out in even pixels. But since the computer must use either a whole pixel or none at all, the fonts can have the actual proportions change a bit when shown on the screen. In the figures you can note small differences in how the titles fit in their boxes; the inexact rendition of font sizes is the reason.

To center a line, we need to know how wide it is. The width changes as the font size and font styles change, but the Toolbox provides functions to do the calculations. The `charwidth()` function takes a character as its argument and returns its width. We could have used that to make a function that returns the width of a whole string, but the Toolbox already furnishes two functions for that purpose. One, called `stringwidth()`, unfortunately is bug-ridden at the present. Therefore we used the more awkward one, `textwidth()`.

The `textwidth()` function takes three arguments. The first is type pointer-to-char and indicates which string is to be processed. The second argument is an integer indicating how far into the string the count should start. The final argument is the number of characters to be measured. Because this function lets you choose the starting and ending points, it can be used for either format of string. In our case, title is in the Pascal argument, so we used 1 as the second argument. That is, we instruct the function to start 1 element over from the beginning, i.e., at `title[1]`. For the third argument we used `title[0]` because, in the Pascal format, that contains the number of characters in the string. If title had been in C format, we would have used 0 and `strlen(title)` for the final two arguments.

Using the left and right borders and the `stringwidth()`, the program calculates the correct horizontal coordinate for starting the string. Next, an arbitrary formula is used to calculate the vertical position. The `moveto()` function moves the pen there, and the `drawstring()` function, which expects and gets a Pascal-format string, draws the title from there.

The `drawstring()` function does not recognize formatting instructions such as the newline character. Thus, you should use `moveto()` to position the text where you want it.

We did not use the `drawtext()` function, but it is much like `drawstring()`, except that it takes the same argument set as `textwidth()`. Thus, like `textwidth()`, it can be used with either string format, so long as you choose the arguments wisely.

The remaining functions concern rectangles, and you have met them before in Chapter 6.

Now that you know how it works, you can go back and fool around with the program, perhaps augmenting the screen with a

"STARRING _____" (fill in the blanks).

Perhaps the most interesting aspect of this example is that there exists a structure describing and controlling the graphics environment. We will take a more organized look at that topic the next chapter.

Programming With Pascal-Formatted Strings

We mentioned that `stringwidth()` seems to have a bug. (This is third-hand information.) It is simple enough to write a C function to take the place of `stringwidth()`. The Toolbox function `charwidth()` returns the width of a single character, so we can apply the function to each character in the string and keep a running total of the widths. Here is a version using array notation:

```
short stringwidth( s )
char s[]; /* s a pointer to beginning of a char array */
{
    short i, width, charwidth();
```

```

    for ( i = 1, width = 0; i < s[0]; i++ )
        width += charwidth(s[i]);
    return width;
}

```

This function makes use of the facts that, in the Pascal format, `s[0]` is the length of the string and `s[1]` is the first character in the string. Thus the for loop starts with `i` set to 1 instead of the usual 0. The `s[0]` value, in turn, serves to tell the loop how many elements to process.

Because the Pascal format provides the string length as the first byte, it is natural to use a for loop to process the string. The C format, with a null string at the end, makes the while loop more natural for that format. It's a matter of whether or not the length of the string is known before the loop starts.

Summary

A **character string** is a sequence of characters stored as a unit. In C, a string is identified by a pointer to the first character, and the end of the string is marked by the null character, `'\0'`, or ASCII zero. The pointer to the first character can be the name of the type char array storing the string, or it can be a declared pointer-to-char that has been assigned that address, or it can be a string constant, which is a sequence of characters in the program code enclosed in double quotes.

String output in C commonly is handled using the `printf()` and `fputs()` functions. The `%s` specifier allows `printf()` to format the string and to combine it with other output. The `fputs()` function prints just one string at a time.

String input is accomplished using `scanf()` and `fgets()`. The `scanf()` function uses the `%s` specifier to read single words, while `fgets()` reads entire lines.

The standard C library contains several *string processing* functions. They perform functions such as finding the length of a string, copying a string, combining strings, and comparing strings.

The Macintosh Pascal format for a string consists of one byte holding the string length followed by the characters of the string. The maximum length for such a string is 255 characters. Hippo C functions convert strings from one format to the other.

The Quickdraw functions **drawstring()** and **drawtext()** print strings. Formatting guides such as newlines are ignored, but the **moveto()** function can be used to control string placement. Other Quickdraw functions control the size and style of the type and provide information about strings.

9

C and the Macintosh Toolbox

In this chapter you will learn about:

- Toolbox subroutines as Pascal procedures and functions
 - Converting Pascal descriptions to C usage
 - Variable and value parameters
 - Pascal types
 - Pascal-C equivalents
 - Grafport, pattern, and cursor structures
-

The Macintosh Toolbox includes hundreds of subroutines that you can use as C functions. To describe them all would take a book of tremendous size. Fortunately, such a book exists; it is issued by Apple, and is called *Inside Macintosh*. (We'll suggest some alternatives at the end of Chapter 10.)

A problem (for C programmers, at least) with *Inside Macintosh* is that the subroutines are described in Pascal terms. As a C programmer, you will need to translate the Pascal descriptions of arguments and of function types to C equivalents. We will discuss how to do that.

Up to now we have been using Toolbox functions in a grab-bag fashion. They are, however, organized into coherent packages. We will take a more thorough look at our favorite package, Quickdraw.

Pascal Procedures and Functions

Pascal subroutines come in two varieties: procedures and functions. The procedure performs an action, but does not have a return value. The function does have a return value. Aside from that, the two are similar. Like C functions, Pascal procedures and functions take an argument list, termed a "parameter" list in Pascal. The procedure and function definitions declare the type of each parameter, and a function is typed according to the type of value it returns. This is all quite similar to C, but the form and some of the type names are different.

Let's look at a couple of excerpts from the manual. First, here is a procedure description:

PROCEDURE DrawChar (ch: CHAR);

DrawChar places the given character to the right of the pen location, with the left end of its baseline at the pen's location, and advances the pen accordingly. If the character is not in the font, the font's missing symbol is drawn.

The description is in clear English; we will concentrate on the procedure declaration. First, the reserved word "PROCEDURE" announces that DrawChar is a procedure. The parentheses contains the parameter list, which includes the type as well as the name of each parameter. Here, there is one parameter called ch, and it is of type CHAR. Note that in a Pascal declaration, the variable name comes first, followed by a colon, then the type. If more than one variable has the same type, the names can be listed together, separated by a comma, as in this hypothetical declaration:

```
PROCEDURE pchars( chfirst, chlast : CHAR);
```

If the procedure uses arguments of more than one type, a semicolon is used to separate declarations of different types:

```
PROCEDURE pnchar( ch : CHAR; n : INTEGER);
```

As in C, the actual parameters used in a procedure call must be in the same order as the formal parameters in the procedure definition.

FUNCTION CharWidth (ch : CHAR) : INTEGER;

CharWidth returns the value that will be added to the pen horizontal coordinate if the specified character is drawn. CharWidth includes the effects of the stylistic variations set with TextFace; if you change these after determining the character width but before actually drawing the character, the predetermined width may not be correct. If the character is a space, CharWidth also includes the effect of SpaceExtra.

A function is defined much like a procedure, except that a type declaration for the function itself comes at the end.

The function declaration tells us that CharWidth is a function name, that it takes one type CHAR argument, and that it returns a type INTEGER value.

Value Parameters and Variable Parameters

The procedure and function parameters we've shown so far are called "value parameters." They work just like regular C function arguments. That is, they are local variables that are created when the routine is called and are assigned the values passed on by the procedure or function call. Once again, we have parameter passing by value, hence the name.

However, in Pascal, just as in C, it is sometimes desirable to let a routine work with the original variables in the calling program. In C we did this by providing a pointer, or address, as an argument. In Pascal, it is done by declaring a parameter to be a "variable parameter" instead of a "value parameter." This means that the actual variable instead of its value is passed on to the routine. Thus, any changes made to the variable by the routine carry over to the calling program.

Here is a sample declaration using a variable parameter:

```
PROCEDURE InsetRect( VAR r: Rect; dh, dv : INTEGER);
```

The reserved word VAR identifies r as a variable parameter of type Rect, which is Pascal record corresponding to the rect structure you've used. The effect of a VAR extends only to the next semicolon; the variables dh and dv are ordinary value parameters. Here r had to be a variable parameter so that the procedure could alter the members in the original structure.

Pascal to C

As a C programmer, your concern is translating the Pascal routine description to instructions which will call the routine from a C program. Let's start with procedures.

First, look at the name of the procedure. In Hippo C, use the same name, but with lowercase letters only. (Some Macintosh C compilers retain the occasional uppercase letters used in the Pascal description.)

Next, look at the argument list. Use the same number of arguments in the same order, matching the type declarations. If the argument is a value parameter, you can use constants, variables, or other expressions of the same type in the procedure call. If the argument is a variable parameter, use the address of a variable as in the procedure call. Thus, the following is a valid Hippo C call to the `InsetRect` procedure described above:

```
insetrect(&box, 25, dh);
```

Here, `box` is of type `struct rect` and `dh` is a short variable.

Next, consider Pascal functions. The only point to add here is that the function should be declared according to its return value type. This step can be omitted if the return value is compatible with the C `int` type.

Some procedures and functions take no arguments. An example is the `HideCursor` procedure, which hides the cursor. In Pascal, such procedures and functions are called using the name alone without any parentheses. In C, however, you must use empty parentheses for argumentless functions. Thus, the Pascal call of `HideCursor` becomes `hidecursor()` in C.

What would rules be without exceptions? Let's look at some now. First, if the Pascal parameter is a structured type (a structure or an array), the C call usually should use the address of the variable regardless of whether the Pascal procedure uses a value parameter or a variable parameter. Thus, the manual describes `FrameRect` this way:

```
PROCEDURE FrameRect ( r: Rect);
```

Nonetheless, the proper C call is `framerect(&box)`; and not `framerect(box)`. The reason for this is that the computer really can't pass a whole structure or array by value anyway. When a Toolbox procedure calls for a value parameter for an ordinary variable, it expects a numerical value. When it calls for a value parameter for a structured variable, it actually expects an address. It then uses the address to copy the original structure into a new one.

As the "usually" in the last paragraph suggests, there is an exception to this exception. A structure that occupies 4 bytes or less *is* passed by value and not address when that is what the routine requests. For example, the function `PtInRect`, which determines if a point is in a rectangle, is defined this way:

```
FUNCTION PtInRect (pt: Point; r: Rect) : Boolean;
```

Here `Point` is a record of two `INTEGER` values, so it occupies 4 bytes. In Hippo C, the corresponding structure is typedefed as `point`. Now, if `position` is a point structure and `box` is a rect structure, the Hippo C call is this:

```
inside = ptinrect(position, &box);
```

One structure uses the address operator, and one doesn't, even though both are declared as value parameters in the Pascal description. On the other hand, the `getmouse()` procedure, which provides mouse coordinates to its point argument, does use the address operator:

```
getmouse(&position);
```

This case calls for a variable parameter. The moral here is to be wary. Fortunately, the point structure is the only Toolbox structured type that falls in the 4-byte category.

One problem with translating the Pascal descriptions to C versions is getting the types right. Let's look at that topic now.

Pascal Types

Like C, Pascal is a typed language. Pascal has 4 fundamental types, to which Macintosh Pascal adds one nonstandard type. We list them with the closest Hippo C equivalent; all uppercase indicates a standard Pascal type:

Pascal	Hippo C
INTEGER	short
CHAR	char
REAL	float
BOOLEAN	short
LongInt	int or long

The correspondence is not exact. For example, in Macintosh Pascal, each CHAR is actually allocated two bytes of memory, even though only the "high-order" byte is used to store the character. (If you visualize the bits in a two-byte unit as numbered from 15 to 0, reading from left to right, the high-order byte consists of bits 15 through 8, and the low-order byte consists of bits 7 through 0.) However, Pascal strings use just one byte per character.

The BOOLEAN type in Pascal refers to values that are either true or false. Internally, it is represented by 1 for true and by 0 for false, so it is compatible with C integer types. But this type uses only the high byte of a two-byte memory unit, so the 1 is placed in bit 8, and gets followed by eight 0s in the low byte. This makes the two-bit number have the value 256 when interpreted as a C short. This is no problem for functions returning a BOOLEAN value, for, in C, 256 is just as "true" as 1. (The Hippo C implementation converts Boolean values returned from Toolbox calls to 1 or 0, but this is not a universal practice.) However, if a Toolbox function expects a BOOLEAN argument, you must provide a short integer with at least some of the left-most bits set to 1. One way to do this is to provide a value of -1, which is represented internally by all 1s. Another is to do something like this:

```
#define PTRUE 256
```

Then you can use PTRUE as an argument.

Pascal offers one more BOOLEAN surprise. A BOOLEAN record member occupies just one byte, instead of two, so in that case the C equivalent is char.

Declaring Variables

Variables in Pascal programs are declared just as they are in parameter lists. For instance,

```
i, j : INTEGER;  
hairdye : BOOLEAN;
```

creates two INTEGER variables and one BOOLEAN variable when declared in the appropriate section of a Pascal program. The C equivalent, using typedefs from the Hippo C data.h file would be this:

```
integer i, j;  
boolean hairdye;
```

Strings

Strings in Macintosh Pascal are an extension to standard Pascal. The following declaration creates a string with room to hold 40 characters:

```
fullname : STRING[40];
```

The C equivalent would be this:

```
char fullname[40];
```

Keep in mind, however, that these two strings use different formats.

Arrays and Records

These types can be used to construct arrays and records. Pascal arrays are much like C arrays, and Pascal records are much like C structures.

Here is a sample Pascal array declaration:

```
stuff : ARRAY [0..15] OF INTEGER;
```

The 0..15 indicates the subscript range for the array. Since there are 16 elements, the corresponding C declaration is this:

```
short stuff[16];
```

Unlike C arrays, Pascal arrays are not constrained to have subscripts beginning at zero. What counts, however, is the size of the array, so a Pascal array with a subscript range of 2..6 would be represented by a C array of 5 elements.

Next, here is a sample Pascal record declaration:

```
fonty : RECORD
    ascent:  INTEGER;
    descent: INTEGER;
    widMax:  INTEGER;
    leading: INTEGER;
END;
```

The individual components (ascent, et al) are called fields of the record and correspond to C structure members. Thus, the equivalent C structure would be defined this way:

```
struct {
    short ascent;
    short descent;
    short widmax;
    short leading;
} fonty;
```

You will not, however, find array and structure definitions in the parameter lists of Pascal routines. Instead, Pascal uses TYPE definitions to create single-word identifiers for these and other types.

Pascal TYPE Definitions

Pascal programs have a TYPE section in which new types are defined. The format is the new type name followed by an equals sign followed by the type description. Here is a sample:

```
TYPE  Str255    = STRING[255];
      Bits16    = ARRAY [0..15] OF INTEGER;
      FontInfo  = RECORD
                          ascent:  INTEGER;
                          descent: INTEGER;
                          widMax:  INTEGER;
                          leading: INTEGER;
      END;
```

Once this is done, Bits16 and its fellows can be used to declare variables in the program and to declare parameter types for functions and procedures.

The Pascal TYPE feature is similar to the C typedef feature, but there is an important difference. The typedef feature creates new *names* for the basic types and their manifold modifications. The TYPE feature creates new *types*. This may seem to be a subtle difference, but there is a practical aspect. Suppose you say this in C:

```
typedef short integer;
```

This makes integer synonymous with short; the two are the same type, and the compiler won't complain if you assign a short to an integer.

Now suppose you say this in Pascal:

```
TYPE short : INTEGER;
```

This creates a *new* type called short. It is identical in properties to INTEGER, but if you try to pass a short variable to a procedure expecting an INTEGER variable, the compiler will tell you that you have a type clash.

Fortunately, we don't have to worry about Pascal's picky nature, for we are using a C compiler, not a Pascal compiler, even when we use the Toolbox. As long as we pass arguments that agree in nature, even if not in type name, we're all right.

Variant Records and Unions

In Pascal, you can set up a "variant record," which is one that can be used in more than one way. Here, for example, is the definition of the Toolbox Point type:

Toolbox Definition: Point type

```
Point      = RECORD CASE INTEGER OF
              0: (v: INTEGER;
                  h: INTEGER);
              1: (vh: ARRAY[VHSelect] OF INTEGER)
            END;
```

Earlier, VHSelect was defined to be the symbol v or h.

This definition means that a Point record can be either a record with two INTEGER fields or else a record with one field consisting of a two-element array. Suppose, for example, that position is a variable of type Point. Then, in a Pascal program, we could refer to the vertical component of the point as position.v (the v field) or else as position.vh[v] (the v element of the array position.vh).

How do we handle this bimodal form in C? One way, (the one used in Hippo C, Level 1) is to select just one variant. Thus, the Hippo C data.h file contains this definition:

```
typedef struct
{
    integer v,h;
} point;
```

Thus, in Hippo C programs, you would refer to the vertical component of point variable position as position.v.

The second method is to use a C data structure called a "union." A union definition is set up much like a structure definition, except the list of types within the braces is a list of choices rather than a list of members. For instance, consider this declaration:

```
union justone {
    char init;
    int  flop;
    char name[8];
} data;
```

This says the variable data is of type union justone, where justone is a tag, just as for structures. If this were a structure definition, the structure would hold one char, one int, and one array of char. But it is a union, and this means that data can hold one char *or* one int *or* one array of char. The union is made large enough to hold the largest single choice. Then that memory location can be used to hold several types of data, but not simultaneously. References use the membership operator. For example, to assign an integer to the union, do this:

```
data.flop = 94562345;
```

What happens if we follow up with this command?

```
data.init = 'R';
```

Then that portion of the union memory required to hold a char is overwritten with the code for 'R'.

Now we can apply this concept to defining the point type. We can make it the union of two members, one corresponding to the two-element record, and one corresponding to the array record:

```
typedef struct { integer v, h} point1;
typedef union  {
```



```

        integer vh[2];
        point1 st;
        } point;
#define V st.v
#define H st.h

```

Then, if position is of type point, we can refer to the vertical component as position.vh[0] or as position.st.v or as position.V. In this case, the two descriptions occupy the same amount of space, so that information stored in one form can be referred to by the other form.

Setting Up Types

The most convenient way to handle the C–Pascal interface for type matters is to set up a file containing appropriate typedefs. For instance, we can use the following definitions in order to work more easily with some of the Pascal types we've mentioned in this chapter:

```

typedef short boolean;
typedef short integer;
typedef int longint;
typedef struct
{
    integer ascent, descent, widmax, leading;
} fontinfo;

```

This is just what Hippo C has done. The file **data.h** contains these and many, many more definitions. By including this file with your program, you can use the same type names used in the Toolbox, making it much simpler to keep track of type matches in procedure and function calls.

If you find that you are using just a certain subset of the definitions, you may wish to extract them and place them in a smaller file. Several C compilers for the Macintosh already have divided the definitions among several files.

Constants

Many Toolbox routines use predefined constants. For example, Quickdraw uses `redColor` to stand for red. This concerns functions that await color hardware. The Hippo C file `defs.h` contains hundreds of such constants set up using the C `#define` facility. Again, this allows your programs to use the same nomenclature that the Apple manual does.

Back to Quickdraw

Now that we have a better understanding of the official Toolbox descriptions, let's return to Quickdraw. The Quickdraw package uses many specially defined types. Already, you have seen the `rect` and `point` types. Now we'll look at some other types used by the package. Not only will this give you practice interpreting data types, but it will also give you more insight into the workings of the package.

The Grafport Structure

The most important structure for the operation of Quickdraw is the Grafport structure. It contains a wealth of information describing the screen environment. We'll give its full definition now, although we will discuss only parts of it. Several of its members are of types we haven't mentioned yet, but we will eventually get to many of them. For the present, our main intent is to indicate the general nature of a Grafport structure. So here is its definition:

```
struct Grafport
{
    integer device;      /* identifies output device */
    bitmap portbits;     /* describes "bit image" */
    rect portrect;       /* writeable area of screen */
    rgnhandle visrgn,cliprgn;
    pattern bkpat,fillpat;
                        /* background, fill patterns */
    point pnloc, pnsz;   /* pen location, size */
    integer pnmode;
    pattern pnpat;       /* paint pattern */
    integer pnvis;
    integer txfont;      /* text font */
    char txface;         /* style */
    integer txmode,txsize,spextra;
                        /* text characteristics */
}
```

```

    int fgcolor,bkcolor;
    integer colrbit,patstrech;
    qdhandle picsave,rgnsave,polysave;
    qdprocsptr grafprocs;
};
typedef struct Grafport grafport, *grafptr;

```

Without going into detail, we can see that a Grafport structure holds information about the screen size, pen parameters, background and fill patterns, text characteristics, and other matters.

One important point to understand conceptually is what the **portbits** member is about. It contains the location of the screen "bit image" and information about its size. The bit image is a representation in memory of the screen. A full screen consists of a grid of 512 by 342 "pixels." A pixel is a small element of the screen, and it can either be on (white) or off (black). All together, there are 175,104 pixels, and each is represented by a bit in computer memory. If a pixel is white, then its corresponding bit is set to 0; black pixels have bits set to 1. If you change the value of one of the bits, then the corresponding pixel changes, too. The screen is said to be "bit mapped."

In general, before using Quickdraw, you need to initialize the graphics package and set up a grafport. In Hippo C, Level 1, this is done automatically, which is why we've been able to use Quickdraw routines without any special preparation. In particular, if you run a program from the Hippo C Command Window, the program inherits the grafport used by the window. Similarly, if you run a program from HOS, the program inherits the HOS grafport.

Once you have a grafport, you can examine and alter its characteristics by using the corresponding structure. We already did so when we used **theport** in Chapter 8. Before going further, let's discuss **theport**.

theport

In Macintosh programming, the name **theport** denotes a pointer to the current grafport. Thus, its type should be pointer-to-struct Grafport, or, using the typedef notations, pointer-to-grafport or, most simply, grafptr. Typically, **theport** is a global variable. In the Hippo C, Level 1, implementation, however, **theport** is the return value of a function. Perhaps you recall its definition from **stdio.h**:

```
#define theport (*(grafptr *) jt_theport() )
```

The return value from the Hippo C **jt_theport()** system function is type cast to "value of pointer-to-grafptr," which reduces to the required grafptr type. More specifically, the #define directive says, "Take the return value from jt_theport(), interpret it as the address of a pointer to a grafptr, find the value pointed to by that address, and assign it to theport." C is a concise language. We can use the value of theport to see where in memory the current grafport is kept, and we also can use theport to access the structure. We'll do that in the next example.

Creating a Grafport

The procedure **openport()** creates and initializes a grafport, but it needs a grafptr argument. It's easy to create a variable of the right type:

```
grafptr gp;
```

However, it is not enough to create gp, for it is just a pointer to where the grafport structure is to be stored. We must also allocate memory to hold the structure. Then we can assign the address of the first byte of the structure to gp, and only then are we ready to hand gp over to **openport()**.

Here's the obvious way to create memory for the grafport itself:

```
grafport aport;
```

Then we could say **gp = &aport;** and be on our way. But it would be a bumpy way, leading (with Hippo C, at least) to a system failure when the program ends. No, we can not use the obvious way.

If we can't use the obvious way, what's left? The answer is "dynamic memory allocation." We'll discuss this topic more fully in the next chapter, but here is the essence. The Macintosh uses three storage areas for memory. Static and external variables are assigned memory locations in a "global" area when the program is compiled and loaded into a.out. Automatic variables are assigned memory locations when the function containing them is called. They are stored in an area called the "stack." Dynamic memory allocation occurs when a program explicitly requests memory as it is run.

This time a memory area called the "heap" is used. The Macintosh is unusually heap-oriented, and grafports are one form expected to be stored there.

Memory Allocation. In C, we can use the malloc() (memory allocation) function to request memory. Its argument is the number of bytes desired. Its return value is a pointer to the first byte of the assigned memory. The malloc() function is defined to return a pointer-to-char; and then a typecast, if necessary, is used to convert it to a pointer of the proper type.

To obtain the number of bytes needed, we can use the C operator **sizeof**. This operator yields the size, in bytes, of a particular variable or data structure or of a data type. When finding the size of a type, enclose the type name in parentheses. Thus, sizeof (int) would be the number of bytes in the int type.

Here, then, is the combination of statements that assigns to gp the address of enough heap memory to hold a grafport:

```
grafptr gp;           /* declare pointer */
char *malloc();       /* declare function type */
gp = (grafptr) malloc ( sizeof(grafport) );
```

The malloc() function locates a sufficiently large hunk of memory and returns a pointer to it. The typecast converts the pointer to the grafptr type, suitable for assignment to gp.

Using a Grafport

With this background, we can now write a program that starts with the grafport it inherits, then creates a new grafport, and finally returns to the original grafport.

From Mac's Toolbox: New Routines

OpenPort	Opens a new grafport
TextFont	Sets type face
ClosePort	Closes a grafport
SetPort	Restores previously opened grafport

Here is the program:

```
/* firstport.c -- makes a new port */
#include "stdio.h"
#include "data.h"
main()
{
    grafptr gp, gpsave; /* two pointers to grafports */
    integer right;
    char *malloc();      /* memory allocation function */

    right = theport->portrect.right;
                        /* current screen right */
    printf("The right screen limit is %d.\n", right);
    gpsave = theport;   /* save old grafport address */
    printf("The grafport is at %u.\n", gpsave);
    printf("Hit the mouse button to continue.\n");
    while ( !button() ); /* wait for button*/
    gp = (grafptr) malloc( sizeof (grafport) );
                        /* create space */
    openport(gp);      /* open new grafport */
    eraserect(&gp->portrect); /* clear screen */
    framerect(&gp->portrect); /* outline screen */
    moveto(20,50);
    printf("gp and theport are %u and %u\n", gp,
           theport);
    printf("Now the right limit is %d\n",
           gp->portrect.right);
    printf("The font is the system font, ");
    textfont(1); /* alter grafport setting */
    textsize(10); /* alter grafport setting */
    printf("but we can change that.\n");
    printf("Hit the mouse button.\n");
    while ( !button() );
    eraserect(&gp->portrect);
    closeport(&gp);
    free(gp);
    setport(&gpsave); /* restore original port */
    framerect(&gpsave->portrect); /* frame it */
    moveto(20,50);
    printf("Well, here we are back at %d\n", theport);
}
```

Figures 9.1, 9.2, and 9.3 show the output of this program at different stages.

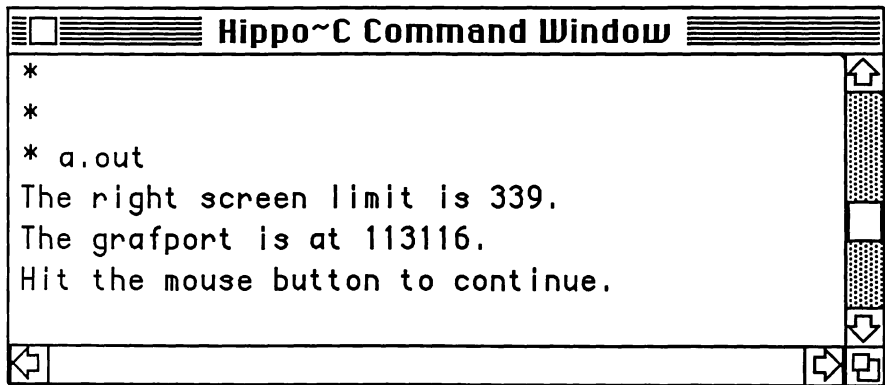


Figure 9.1 firstport.c output #1

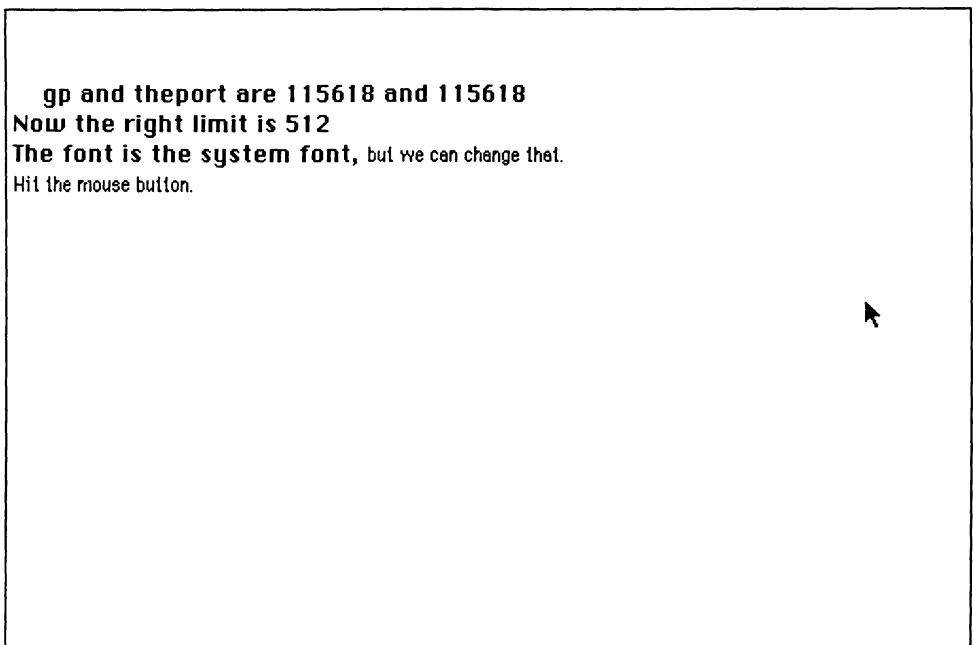


Figure 9.2 firstport.c output #2

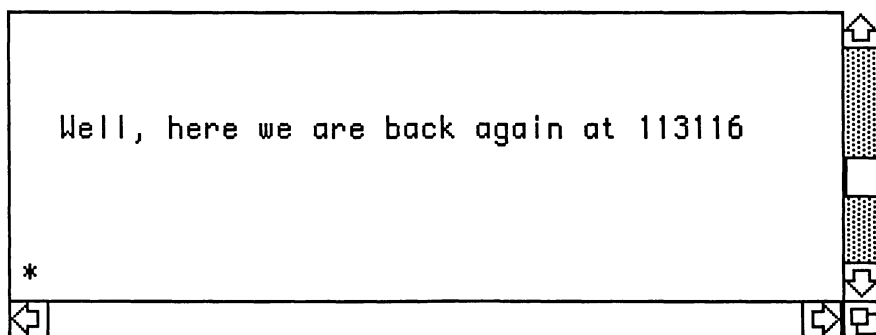


Figure 9.3 **firstport.c output #3**

As you can see in Figure 9.1, the program starts out using the Hippo C Command Window grafport. We adjusted the window before running the program to be narrower than usual. Figure 9.2 shows what happens when we create a new grafport. Note that theport, like gp, is the address of the new grafport. The openport() procedure initializes the grafport members to default values. In particular, the portrect member is set to cover the whole screen. Also, the font is set to the bold system font.

We used textfont() and textsize() to change the appearance of the font. These functions alter the contents of the txfont and the txsize members of the grafport. We could, instead, have done this:

```
gp->txfont = 1;  
gp->txsize = 10;
```

However, Apple recommends using the function calls, since they are specifically designed to avoid unexpected side effects. For textfont(), 0 selects the system font, and 1 selects the application program font.

The closeport() function frees some memory areas reserved by openport() for the use of the grafport. The C function free() returns the grafport memory area itself back to the memory pool. Then setport() restores the original port as active port. As Figure 9.3 shows, theport once again has its original value. If we had used openport() instead of setport() here, the grafport parameters would have been reinitialized to the same system values provided to gp's grafport.

Let's look at a few more aspects of a grafport.

Pen Parameters

Several grafport members refer to the "pen." The pen is the imaginary implement that leaves real lines behind as it is manipulated by the various drawing commands. The **pnloc** member holds the current location of the pen; this member is type **point**, meaning it is a structure with the vertical position and horizontal position as its two members. The **pnsiz** describes the track left by the pen. It, too, is type **point**. The vertical component describes the width of a vertical line, and the horizontal component gives the width of a horizontal line; the default values are both 1. The upper left-hand corner of the pen aligns with the coordinates given in drawing instructions. The **pnmode** member indicates the drawing mode. The mode determines, for example, what happens when the pen tries to draw across a black area. Does the pen mark remain black or turn white? There are several modes to cover different possibilities. The **pnpat** indicates what mark the pen makes when drawing. By default, it leaves a black mark, but that can be changed. We'll talk about patterns shortly. The **pnvis** member determines whether or not the pen draws as it moves. A negative value renders the pen invisible.

These members can be accessed directly, but the preferred practice is to use Quickdraw routines designed for the purpose. Here is an example illustrating pen control. It also introduces the **random()** function, which in Hippo C returns a number in the range 0 to 65535. Incidentally, due to the nature of how negative numbers are stored, (see Appendix C) **random()** will return numbers in the range -32768 to 32767 if we declare it to be type **short**. We use the modulus operator (%) to restrict the range to values needed in the program.

From Mac's Toolbox: New Routines

PenSize	Sets pen size
Random	Provides a random number
Line	Draws a line
PenNormal	Restores standard pen settings

```
/* rwalk.c -- a random square takes a random walk */
#include "stdio.h"
#include "data.h"
main()
{
    short hor, ver, dhor, dver, signh, signv;
    short topb, leftb, rightb, bottomb;
```

```

short psize, loop;

eraserect(theport->portrect);
topb = theport->portrect.top;
leftb = theport->portrect.left;
bottomb = theport->portrect.bottom;
rightb = theport->portrect.right; /* store bounds */
while ( !button() )
{
    psize = random() % 8 + 3; /* choose pen size */
    pensize(psize,psize); /* set pen size */
    hor = leftb + random() % (rightb -leftb);
    ver = topb + random() % (bottomb - topb);
    dhor = random() % ( (right - left) / 10 );
    dver = random() % ( (bottom - top) / 10 );
    moveto(hor,ver); /* move to random position */
    for ( loop = 0; loop < 500; loop ++ )
    {
        signh = (random() - 32767) > 0 ? 1 : -1;
                                                /* 1 or -1 */
        signv = (random() - 32767) > 0 ? 1 : -1;
        hor += signh * dhor; /* shift position */
        if (hor < left || hor > right )
                                                /* if too far */
            hor -= 2 * signh * dhor; /* back up */
        ver += signv * dver;
        if (ver < top || ver > bottom )
            ver -= 2 * signv * dver;
        moveto(hor,ver); /* move to new position */
        line(0,0); /* leave pen mark there */
    }
    eraserect(&theport-portrect);
    pennormal(); /* restore standard pen setting */
}

```

Figure 9.4 shows a sample output, although it doesn't capture the dynamic visual effect of the running program.

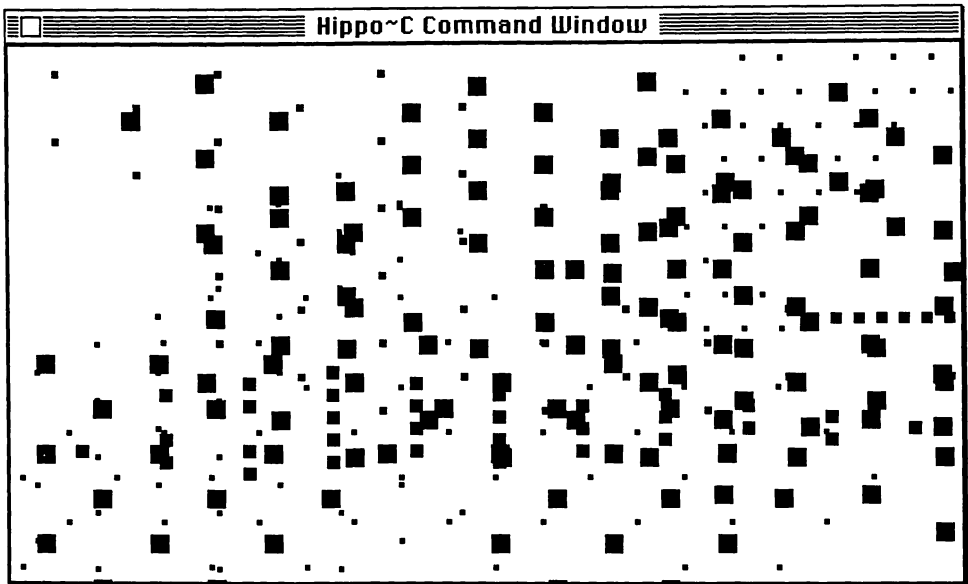


Figure 9.4 `rwalk.c` output

Now let's discuss some points about the program. The pen can be thought of as having a rectangular footprint. The `pensize()` function sets the width and height of the pen mark, and stores the information in the `pnsize` member of the `grafport`. The program chooses and sets `pensizes` randomly, but at the end it uses the `pennormal()` command to restore the standard pen values. Otherwise, the Hippo C Command Window will be left using the final pen size chosen by the program!

The `moveto()` command moves the pen from its current location (which is stored in the `pnloc` member of the `grafport`) to the indicated location. It makes no mark (`pnvis` gets set to a negative number), and `pnloc` is updated to the new position.

The `line()` function takes two arguments; the first tells it how far to move the pen horizontally from the current position, and the second details the vertical movement. The zero arguments used in the program mean the pen stays fixed. Nonetheless, a pen mark is made. Since the pen hasn't moved, the basic rectangular footprint of the pen is revealed. As mentioned earlier, the upper left corner is situated at the specified coordinates.

The program scales the pen movement to the size of the current window, using `theport` to obtain current values. If the position shift would

take the pen beyond the portrect boundaries, the program reverses the last motion in order to bring the pen back in. The `signh` and `signv` variables are used to change the direction of motion randomly, since there is a 50-50 chance that the random number returned by `random()` will be larger or smaller than 32767.

Patterns

Let's look at another data type used in grafports and other parts of Quickdraw: the pattern. In Hippo C, the pattern type is defined this way:

```
struct Pattern
{
    char bytes[8];
};
typedef struct Pattern pattern, *patptr;
```

Hence `pattern` means type `struct Pattern`, and `patptr` indicates a pointer to that type of structure.

Since a `char` type is 8 bits, a pattern is a 64-bit structure. Think of it as representing an 8 by 8 grid, with `bytes[0]` being the top row and `bytes[0][0]` being the first element in the top row. Each 1 in that grid represents a dark pixel on the screen, while a 0 in that grid represents a white pixel. By assigning the values to the grid members, we can create a pattern which then can be used in various places. The **`bkpat`** and **`fillpat`** members of the grafport are just such patterns. Certain Quickdraw commands use them, too. For instance, `eraserect()` fills the rectangle with the `bkpat` pattern, while `paintrect()` fills it with the `fillpat`. Rather than alter the grafport parameters, however, we will use `fillrect()` to show how patterns work. This function takes a pattern address as an argument and fills the rectangle using that pattern. Here is a short program showing how it works and looks:

From Mac's Toolbox: New Routines

FillRect

Fill rectangle with pattern

```
/* heart.c -- a head start for Valentine's Day */
#include "stdio.h"
```

```

#include "data.h"
main()
{
    rect box;
    static pattern newfill = { "Try me" };
                                /* haphazard try */
    static pattern heart =
        {0x44,0xEE,0xBA,0x92,0x44,0x6c,0x38,0x10};
                                /* planned patternhood */

    eraserect(&theport->portrect);
    setrect(&box, 50, 50, 200, 200);
    fillrect(&box, &newfill);
    invertoval(&box);
    offsetrect(&box, 210, 0);
    fillrect(&box, &heart);
    invertoval(&box);
}

```

If you run this program from the Hippo C Command Window, first expand the window to accommodate the boxes. Figure 9.5 shows the output.

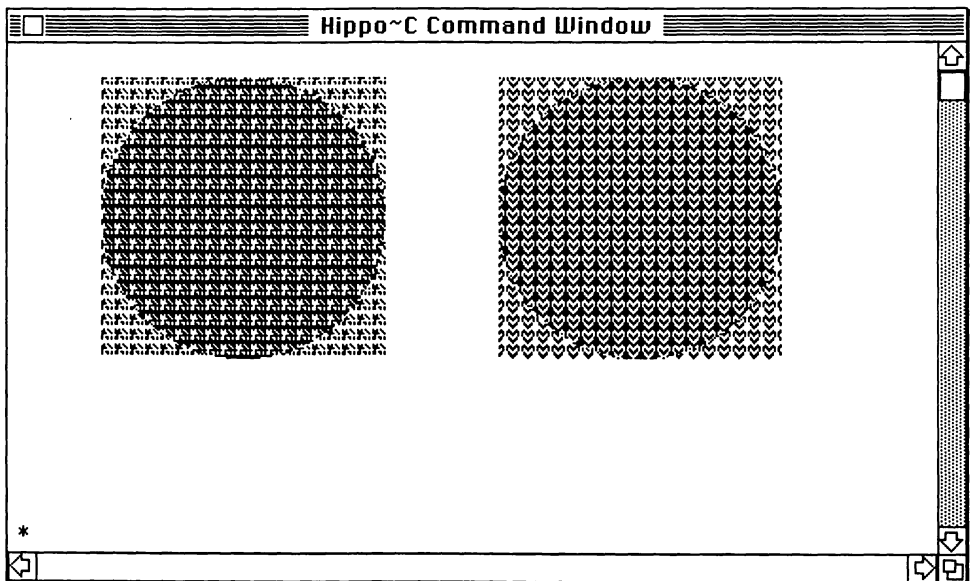


Figure 9.5 **heart.c** output

The trickiest part is initializing the patterns; we'll come back to that soon. First, however, let's note a few things about the Quickdraw functions that we haven't really emphasized before.

The `setrect()` function creates a mathematical representation of a rectangle. The various rectangle-related functions then act upon the interior of this rectangle. For example, `framerect()` draws a frame just inside the mathematical boundaries. The `fillrect()` function, which we use here, fills the interior with the specified pattern. The `offsetrect()` function moves the mathematical rectangle, but has no effect on what has been drawn on the screen. That is, it does *not* move the previously filled rectangle.

Note, too, how the `invertoval()` function works. Previously, we converted great white expanses to great black expanses, but Figure 9.5 shows how each pixel within the affected area is inverted individually. Thus, the dark hearts on white become white hearts on dark.

Pattern Construction

Now let's look at the patterns. A pattern is a structure containing one array of 8 bytes. A single character in C occupies one byte, so we used an 8-character object to initialize `newfill`:

```
static pattern newfill = { "Try me" };  
/* haphazard try */
```

Thus `newfill.bytes[0]` was assigned the character 'T'. If you only see seven characters, recall that the null character terminates a C string, so `newfill.bytes[7]` was assigned the null character—all 0s. As the comment suggests, this was a haphazard attempt. It is technically correct, for it assigns values to all the bits; but it does not produce a pattern that one might deliberately design.

The heart definition, however, is deliberate. It looked like this:

```
static pattern heart =  
    { 0x44, 0xEE, 0xBA, 0x92, 0x44, 0x6C, 0x38, 0x10 };  
/* planned patternhood */
```

Here, instead of using a character string, we have initialized each element individually with a hexadecimal number. How did we arrive at these values? We drew an 8-by-8 grid, darkened squares to represent the pattern we wanted, converted the pattern of dark and light to a pattern of 1s and 0s, then interpreted each row as a hexadecimal number. This is easier than it sounds, for each row is represented by a two-digit hexadecimal number. (Appendix C discusses hexadecimal numbers.) The first (left) digit represents the left four squares, and the right digit represents the right four squares of a row. Figure 9.6 illustrates the scheme.

	upper nibble				lower nibble				
	8	4	2	1	8	4	2	1	
bytes[0]	0	1	0	0	0	1	0	0	0x44
bytes[1]	1	1	1	0	1	1	1	0	0xEE 8 + 4 + 2 = 14 = E
bytes[2]									
bytes[3]									
bytes[4]									
bytes[5]									
bytes[6]									
bytes[7]									

Figure 9.6 Setting up a pattern

Note that in the figure we've placed 8 over the first column from the left, 4 over the second, 2 over the third, and 1 over the fourth. Then we repeat the process for the next four columns. To obtain the first hexadecimal digit, add up the heading values for those first four elements in a row containing a 1. If the answer is 10 or larger, convert it to the hexadecimal equivalent: A is 10, B is 11, and so on. For example, in the second row,

there are 1s in the columns headed 8, 4, and 2. Those numbers sum to 14, which is E in hexadecimal. Thus the first digit is E. In this case, the second digit also is E, so the complete number is 0xEE; recall that C uses the 0x prefix to denote hexadecimal notation.

Grafport Patterns and stuffhex()

Several members of the grafport structure are patterns. The bkpat member is the pattern used for the background and for the various erase commands. The openport() call initializes it to white. The fillpat member is used to store the pattern currently being used for a fill call (such as filloval()). The pnpat member holds the pattern made by the pen as it draws; openport() initializes it to black.

Suppose you wish to change one of the patterns. You can't use the static variable initialization technique of the last example, for the structure members aren't variables declared in the program. And it would be tiresome to set each byte separately. Fortunately, the Quickdraw stuffhex() function offers a simple way to assign values to a pattern.

As its name implies, stuffhex() requires using the same hexadecimal code we used before. The function takes two arguments. The first is the address of the data form to be stuffed, and the second is a Pascal-format string consisting of the hexadecimal digits. Here is how to set up the heart pattern using stuffhex():

```
pattern heart;           /* declare a pattern variable */
char *hearts = "44EEBA92446C3810";
                        /* create pointer to hex string */

strctop(hearts);        /* convert string to Pascal format */
stuffhex(&heart,hearts); /* stuff the pattern into
                        heart */
```

There are several points to note. First, because we are not initializing the pattern heart, it can be an automatic variable instead of static or external. Second, stuffhex() expects all the characters in the string to be hex digits, so it is not necessary to use the 0x prefix. Third, hearts doesn't have to be static or external to be initialized because it is a pointer, not an array. Fourth, the strctop() function converts the pointed-to string to the Pascal format. Finally, the stuffhex() arguments are the address of the pattern (&heart) and a pointer to the string (hearts).

This method requires more steps than the initialization method, but it has three advantages. First, the representation of the pattern is more compact, for no commas and no hex prefixes are needed. Second, it can be used at any time to give a value to a pattern structure; it's not confined to the declaration section. Third, because the representation is a string, it can easily be copied and altered.

Let's use `stuffhex()` to alter the background and pen patterns. We'll open a new port, set the patterns, and see what happens.

From Mac's Toolbox: New Routines

StuffHex

Puts hex-coded bits into a data structure

```
/* newpats.c -- alter grafport patterns */
#include "stdio.h"
#include "data.h"
main()
{
    grafptr gp;
    rect box; /* use for drawing */
    char *newpen = "99667ebdbd7e6699"; /* hex code for
                                         pen pat */

    char *back = "991818ffff181899";
    char *white = "0000000000000000";
    char *malloc();

    strtoc(newpen); /* convert the strings to
                     Pascal format */
    strtoc(back);
    strtoc(white);
    gp = (grafport) malloc( sizeof (grafport) );
    openport(gp);
    stuffhex(&gp->bkgpat, back); /* set grafport
                                bkgnd pat */
    stuffhex(&gp->penpat, newpen);
    erasect(&gp->portrect); /* draw in new
                             background */
    setrect(&box, 50, 50, 450, 300);
    paintoval(&box); /* uses new pen pattern */
    insetrect(&box, 64, 40); /* reduce box size */
    invertoval(&box); /* invert new pen pattern */
    insetrect(&box, 64, 40);
    eraseoval(&box); /* uses new background pattern */
}
```

```

while ( !button() ); /* wait for mouse button */
stuffhex(gp->bkpat, white); /* blank background */
eraserect(gp->portrect); /* clear screen */
closeport(gp); /* develop tidy habits */
}

```

Figure 9.7 shows the appearance on the screen before the mouse button is pushed.

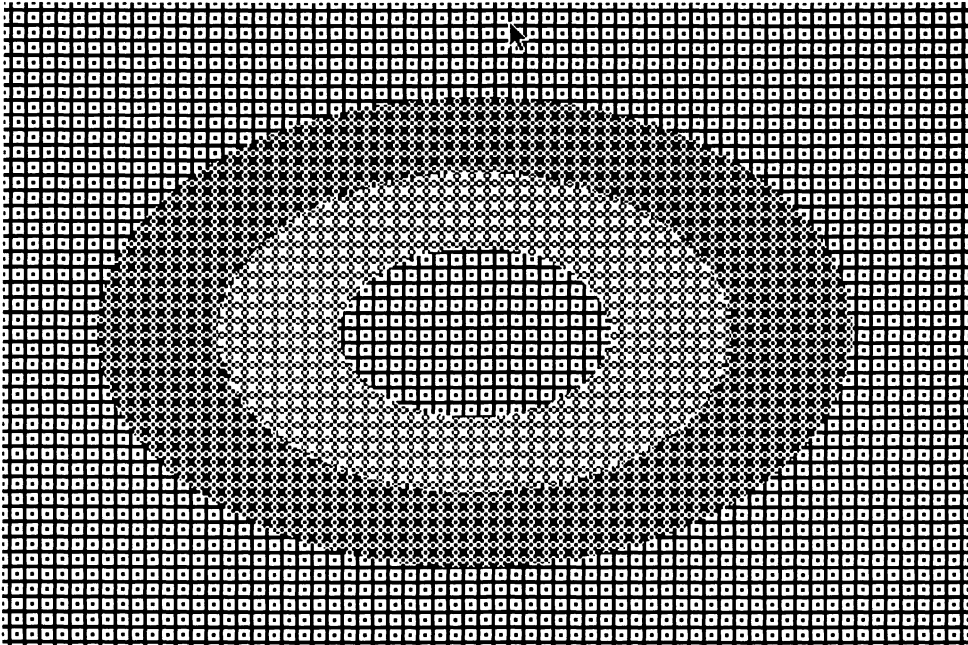


Figure 9.7 Output of `newpats.c`

Since `gp` is a pointer to the `grafport`, `gp->bkpat` is the `bkpat` member of the structure, and `&gp->bkpat` is the address of that member. The first call to `stuffhex()` sets the new background pattern. For variety, we used lowercase hex letter-digits; C recognizes either case. The screen does not change until the first `eraserect()` call. At that point, the screen is filled with the new background pattern. The `paintoval()` call paints in the oval using the new pen pattern. The `invertoval()` call, operating within a smaller rectangle, inverts the pen pattern. The `eraseoval()` call reminds us that this call doesn't really erase the affected area; it just paints it with the background pattern. Note that each pattern obliterates the one before it; the patterns are not superimposed.

Designing patterns is fun, but let's move on to a larger form of pattern, the cursor.

The Cursor

The cursor is the screen marker that represents mouse movement. You can design and set your own cursor if you like. Here is the Hippo C definition of the cursor type:

```
typedef char bits16[32];

struct Cursor
{
    bits16 data,mask;
    point hotspot;
};

typedef struct Cursor cursor, *cursptr;
```

The hotspot is the active part of the cursor, the part that has to be positioned over a location when you wish to click it. The data member is an array of 32 bytes, or 256 bits. It should be thought of as a 16-by-16 grid in which to set up an image of the cursor. Its the same idea as the pattern, only bigger. The mask member also is a 16-by-16 grid. Each element of the mask determines what happens to the corresponding data bit as it passes over light and dark backgrounds. We'll return to that, but first, let's design a cursor.

This time we draw a 16-by-16 grid and fill it in with light and dark squares. (Graph paper is handy for this purpose.) We divide each row into 4 parts, each with an 8-4-2-1 heading like the one we used for pattern design. Again, each 4 element section is translated into one hexadecimal digit. Each two-digit combination then is assigned to one byte.

What about the mask? The mask, too, is a 16-by-16 grid. The value of the mask bit determines what happens when the corresponding cursor bit passes over different backgrounds. Here is the scheme:

data bit	mask bit	Resulting pixel on screen
0	1	white
1	1	black
0	0	same as pixel under cursor
1	0	inverse of pixel under cursor

A mask of 1 means that the black part of a cursor will become invisible if the cursor passes over a black field. A mask of 0 makes the cursor visible, for the 1 bits always appear the opposite of the background, while the 0 bits always blend in.

The next program shows the outcome of this process of creating a design and a mask. In it, we initialize a cursor variable `newcur`. The cursor type, we saw, is a structure of three structures, so we initialize each structure. The first structure (`newcur.data`) is initialized to the first clump of data; the numbers were generated by the graphic process we just described. Then we initialize the mask section. We arbitrarily set the upper half of the mask to all 0s and the lower half to all 1s. Then we set the hotpoint to 7,7. The `setcursor()` function uses the address of `newcur` to set the cursor for that pattern.

In the program we introduce a very useful Quickdraw function called `ptinrect()`. As you might guess, it determines whether or not a point lies within a rectangle. Its arguments are the name of a point variable, and the address of a rect variable. We use it in conjunction with the `getmouse()` function, which determines the mouse position. This function takes as an argument the address of the point variable used to store the information. By using these functions along with `button()`, we can use mouse clicks to control the progress of the program.

We also introduce the `initcursor()` function, which sets the cursor to the standard arrow. Otherwise, as you perhaps have noted, the program gets stuck with whatever the last cursor was, usually the clock.

From Mac's Toolbox: New Routines

PtInRect	Determines if a point is in a rectangle
InitCursor	Set the arrow cursor
SetCursor	Set cursor to indicated form
GetMouse	Obtain the mouse location

Here is the program; try it out and see what happens to the cursors as they pass over light and dark areas; that will show how the mask works. If you are running the program from the Hippo C Command Window, expand the window to nearly full screen before running the program.

```
#include "data.h"
#include "stdio.h"
main()
{
    static cursor newcur = { /* first, define pattern */
        { 0x00,0x00,0x00,0x00,0x21,0x08,0x13,0x90,
          0x0E,0xE0,0x0C,0x60,0x19,0x30,0x33,0x98,
          0x19,0x30,0x0C,0x60,0x0E,0xE0,0x13,0x90,
          0x21,0x08,0x00,0x00,0x00,0x00,0x00,0x00 },
        /* now comes the mask */
        { 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
          0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
          0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
          0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF},
        /* and now the hotspot */
        { 7, 7}
    };
    rect msgbox, box1, box2;
    point mouse; /* mouse position */

    erasereact(&theport->portrect);
    framereact(&theport->portrect);
    setrect(&msgbox,20,50,300,100);
    setrect(&box1, 320, 50, 400, 100 );
    setrect(&box2, 320, 150,400, 200 );
    framereact(&msgbox);
    framereact(&box1);
    framereact(&box2);
    textsize(18);
    moveto(340,80);
    drawchar('1');          /* label box1 */
    moveto(340,180);
    drawchar('2');          /* label box2 */
    textsize(12);
    moveto(30,65);
    drawstring(strctop("You now see the leftover
cursor."));
    moveto(30,80);
    drawstring(strctop("Click box 1 to initialize
cursor."));
    getmouse(&mouse);
    while ( !button() || !ptinrect(mouse,&box1) )
        getmouse(&mouse);
```

```

    initcursor();          /* activate standard cursor */
    eraserect(&mesgbox);
    framerect(&mesgbox);
    moveto(30,65);
    drawstring(strctop("This is the standard cursor." ));
    moveto(30,80);
    drawstring(strctop("Click box 2 to see a new cursor." )
    ));
    invertrect(&mesgbox);
    while ( !button() || !ptinrect(mouse,&box2) )
        getmouse(&mouse); /* continue until click in box */
    eraserect(&mesgbox);    /* erase old message */
    framerect(&mesgbox);
    setcursor(&newcur);    /* activate new cursor pattern */
    moveto(30,65);
    drawstring(strctop("Great! Click this box to stop." )
    ));
    invertrect(&mesgbox);
    while ( !button() || !ptinrect(mouse,&mesgbox) )
        getmouse(&mouse) ;
    eraserect(&theport->portrect);
}

```

One More Example

Now that we know more about Quickdraw, let's do one more example. This program produces a grid (we're bullish on rectangles) in which you can cause circles to appear by clicking the mouse in the chosen box. The circles come out alternately light and dark, making the screen look a bit like a game board for the ancient game of Go. To stop the program, click the mouse outside of the game board. Note that the program contains two new functions of our own: `makeboxes()` and `mouseinbox()`.

From Mac's Toolbox: New Routines

UnionRect	Make smallest rectangle containing 2 others
------------------	---

Here is the program listing (once again, the `wait()` function is compiled separately):

```

#include "data.h"
#include "stdio.h"
#define ROWS 4
main()
{
    grafptr gp;
    rect boxes[ROWS][8];
    static point upperleft = { 50, 56};
    static pattern heart =
        {0x44,0xEE,0xBA,0x92,0x44,0x6C,0x38,0x10 };

    char *malloc();
    void makeboxes(), mouseinbox();

    initcursor();
    gp = (grafptr) malloc(sizeof(grafport));
    openport(gp);
    fillrect(&gp->portrect,&heart); /* fill screen with
                                     hearts */

    pensize(4,4);
    makeboxes(boxes,upperleft,50,ROWS);/* create boxes */
    mouseinbox(boxes,ROWS);           /* mouse work */
    erasereact(&gp->portrect);
    closeport(gp);
    free(gp);
}

void makeboxes(bp,ul,size,rows)
rect (*bp)[8];           /* pointer to row of 8 boxes */
point ul;                /* upper left corner of grid */
integer size;            /* box size */
integer rows;            /* number of rows */

{
    int row, col;

    moveto(ul.h,ul.v);
    for ( row = 0; row < rows; row++) /* each row */
        for (col = 0; col < 8; col++) /* each column */
        {
            setrect(&bp[row][col],ul.h + col*size,ul.v +
                row*size, ul.h + (col + 1)*size, ul.v + (row +
                1)*size);
            erasereact(&bp[row][col]); /* erase hearts */
            framereact(&bp[row][col]);
        }
}

```

```

void mouseinbox(bp,rows) rect (*bp)[8];
integer rows;
{
    integer row, col;
    point mouse;                      /* mouse location */
    rect stop;
    int pebble = 0;

    unionrect(&bp[0][0],&bp[rows-1][7], &stop);
    invertrect(&stop);
    insetrect(&stop, -10, -10 );
    invertrect(&stop);
    framerect(&stop);
    do
    {
        getmouse(&mouse);
        for (row = 0; row < rows; row++)
            for (col = 0; col < 8; col++)
                if ( ptinrect( mouse, &bp[row][col]) && button())
                {
                    if ( ++pebble % 2 == 1)
                    {
                        eraseoval(&bp[row][col] );
                        frameoval(&bp[row][col] );
                    }
                    else
                        paintoval(&bp[row][col] );
                    wait(40);
                }
    } while ( !button() || ptinrect(mouse, &stop));
}

```

Figure 9.8 shows the screen appearance.

We've used one new Quickdraw routine. It's called `unionrect()`, and it finds the smallest rectangle enclosing two rectangles. The three arguments are pointers to the two rectangles and the address of the rectangle to include them. We took a box to enclose the upperleft and lowerright boxes. We did this so that we only have to check if the mouse is outside this one box rather than check each of the smaller boxes individually.

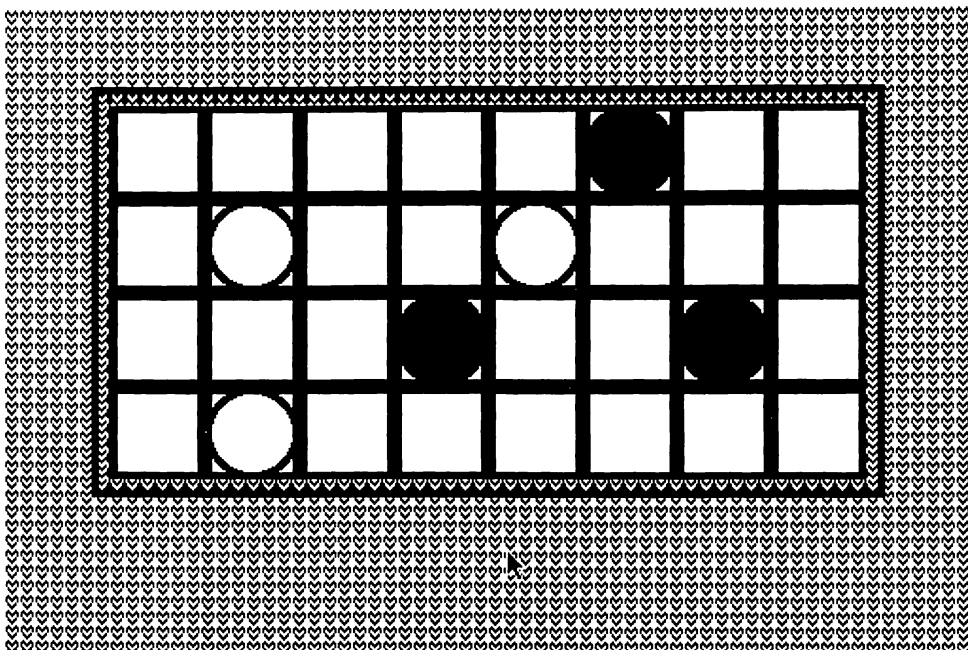


Figure 9.8 Screen appearance for `go.c`

The rest of the program uses familiar techniques, including passing a two-dimensional array of structures as a function argument. One point of interest is the use of the `wait()` function in `mouseinbox()`. Without it, the inner loop will cycle several times before you can release the mouse button, leading to unpredictable results. This is a bit of a kludge. In Chapter 10 we'll show a better way to handle this problem.

In `mouseinbox()`, the nested for loops check each box in turn to see if the mouse is in the box and if the mouse button is being pushed. If so, a circle is drawn. The variable `pebble` acts as counter; depending on whether it is odd or even, a white or black circle is drawn.

The stop rectangle is set up to be slightly larger than the board grid. Note the condition for the loop control variable for the outer `do...while` loop:

```
!button() || ptinrect(mouse,&stop);
```

It remains true as long as either the mouse button is up or the mouse is in the rectangle stop. Thus, to halt the program, you must press the button while the mouse is out of the large rectangle.

Summary

The **Toolbox** is described in Pascal terms, which the C programmer must convert to the appropriate C idioms. Pascal procedures correspond to C functions without return values, while Pascal functions correspond to C functions with return values. A C function call must match the Pascal description of arguments in both type and number; also, the type of a C function should match the return value of a Pascal function. Pascal value parameters correspond to regular C arguments, while Pascal variable parameters correspond to passing an address in C. An exception occurs for arrays and structures, in which case the C call should pass an array or structure address when either a variable or value parameter is called for. The **Point** structure is an exception to the exception and follows the same rules as ordinary variables. The **Toolbox** utilizes a host of Pascal Type definitions. These can be converted to C equivalents using `#defines` and `typedefs`. Hippo C provides a header file **data.h**, which contains such definitions.

Using **data.h** makes using **Quickdraw** easier. This package makes heavy use of predefined structures and pointers to structures. One can use the included definitions to facilitate using routines to create and modify grafports, to create new patterns and cursors, to control the pen appearance, and to monitor the location of the mouse, among other possibilities.

10

A Mac Miscellany

In this chapter you will learn about:

- **Macintosh managers**
 - **Memory management**
 - **Stacks and heaps**
 - **Relocatable and nonrelocatable blocks**
 - **Pointers and handles**
 - **Quickdraw regions**
 - **The event manager**
 - **Files**
 - **The sound driver**
-

Programming in C for the Macintosh takes a lot of knowledge. You have to know C, of course, and you have to know about the Macintosh. We have concentrated on presenting C, but in this chapter we will look more at Macintosh matters. The topic is too vast for a book of this size, let alone a chapter, so this chapter will give a quick overview, then concentrate on concepts that lay the groundwork for future development. In particular, we will look at memory management, events, and files. Once more, we'll draw upon Quickdraw for examples.

The Macintosh Software System

The Macintosh has a large selection of built-in software, with nearly 500 routines stored in its ROM (read only memory). The routines can be subdivided into two classes: the operating system and the Toolbox. These classes are divided further into modules called "managers." Here are the major managers.

Toolbox Managers

- Quickdraw
- Toolbox Utilities
- Font Manager
- Event Manager
- Resource Manager
- Window Manager
- Control Manager
- Dialog Manager
- Menu Manager
- Desk Manager
- Text Edit
- Scrap Manager
- Package Manager

Operating System Managers

- Event Manager
- Operating System Utilities
- Memory Manager
- File Manager
- Segment Loader
- Vertical Retrace Manager

The division between the two groups is somewhat fuzzy, as the appearance of the Event Manager in each group testifies. Each package has an area of responsibility, although the areas sometimes overlap. The Quickdraw package, for example, essentially is the screen manager, and the Window Manager must draw upon Quickdraw to do its work of controlling windows.

Each package is described in a separate section of Apple's *Inside Macintosh* volumes. The discussions there explain the concepts behind each package and present a description of the included routines along the lines of the Quickdraw excerpts you saw in Chapter 9.

We'll study the Memory Manager and the Event Manager in more detail. Let's begin by looking into how the Macintosh handles memory.

Macintosh Memory Management

The Macintosh uses three techniques and three different memory areas for storing program data. We outlined them in Chapter 9, but it won't hurt to repeat them now.

Variables that are declared externally or that are declared to be storage class static are placed in a "static" memory area. The memory space they need is allocated when the program is compiled and loaded. Automatic variables have their memory needs allocated when the function using them is called; they are placed in an area called the stack. Finally, storage space explicitly requested during run time (for example, by the malloc() function)

is allocated from an area called the heap. Let's examine each type more carefully now.

Static Memory

The term static is used to indicate that these memory locations stay assigned as long as the program runs. In Hippo C, the static memory locations are stored in the same general area as the program code, which is in a memory section called the "application heap." Once a program is finished, all its memory locations are liberated, so the "static" variables of the program are not truly permanent. The Macintosh system, however, has its own set of global variables that are really static, for they persist even after your program quits. They are stored in the system static memory locations, which occupy a little over 2000 memory addresses at the very beginning of memory. For example, the tickcount (the number returned by the tickcount() function) is obtained through using location 362. (You don't need to know the address, for the function takes care of matters for you.) Figure 10.1 outlines the location of these memory locations and of others we will discuss.

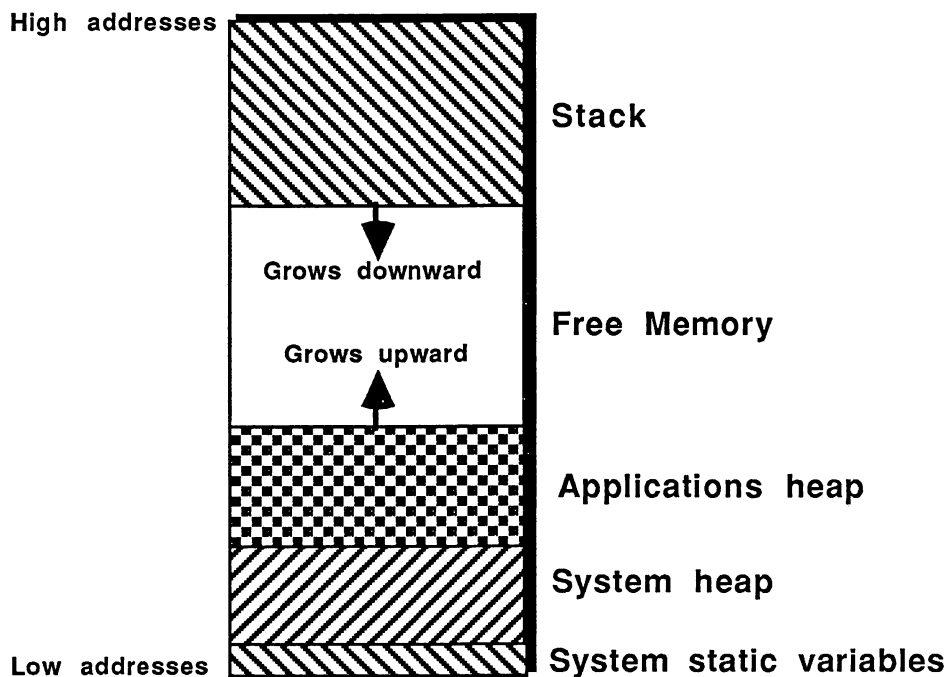


Figure 10.1 **Memory in the Macintosh**

The Stack

The stack, used for automatic variables, starts near the top (large addresses) of memory, just below the area holding the bit mapping for the screen. Each time a function is called from a program, its automatic variables are added to the stack, and when a function quits, its automatic variables are removed. The actual contents of a memory location aren't erased, but they are no longer accessed by the variable name. Also, the location is freed for the next variable needing it. On the Macintosh, the stack actually grows downward from large addresses to smaller ones. So here, the phrase "top of the stack" refers to the low-memory end of the stack. Figure 10.2 illustrates the process.

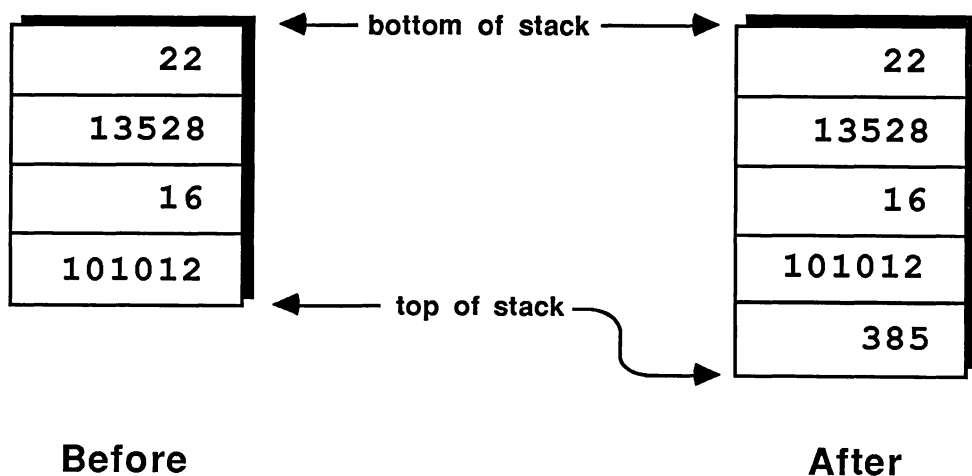


Figure 10.2 Adding a new value to the stack

To get a more concrete view of what's happening, let's look at a program that creates some static and some automatic variables, calls some functions with automatic variables, and prints out the addresses for all the variables. Here is the program.

```
/* addresses1.c -- shows addresses of variables */
char ext1, ext2; /* two external variables -
                  - static storage */

main()
{
    char auto1, auto2; /* two stack variables */
    void funct1(), funct2();
```

```

    printf("The address of ext1 is %u\n", &ext1);
    printf("The address of ext2 is %u\n", &ext2);
    printf("The address of auto1 is %u\n", &auto1);
    printf("The address of auto2 is %u\n", &auto2);
    auto1 = 22;
    funct1(auto1);
    funct2(auto2);
}
void funct1(a)
char a;      /* formal argument */
{
    char b;   /* local variable */

    printf("The address of a is %u\n", &a);
    printf("The address of b is %u\n", &b);
}
void funct2(c)
char c;
{
    printf("The address of c is %u\n", &c);
    funct1(c);
}

```

Here is the output; note how the static variables are in one area, while the automatic variables (including the formal function arguments) are in another. Note, too, how successive static variables come later in memory, while successive automatic variables come earlier in memory.

```

The address of ext1 is 36448
The address of ext2 is 36450
The address of auto1 is 106823
The address of auto2 is 106822
The address of a is 106819
The address of b is 106807
The address of c is 106816
The address of a is 106805
The address of b is 106793

```

The stack memory process is very orderly. Memory assignments operate on a "last in, first out" (or "lifo") basis. Note how funct2() uses a memory location in the midst of the memory area used by the first call of funct1(). Also, see how the second call to funct1() results in different locations being assigned to a and b, for now they are added to the stack on

top of `funct2()`'s assignments. Gaps between the addresses shown indicate that the stack is being used for matters in addition to the declared variables.

The orderly stack is well suited for handling automatic variables. Add memory when a function is called, and remove memory when the function finishes. Additions and subtractions are always made at the top of the stack, in the same order that functions are called and dismissed. A function that has called another function doesn't quit before the called function quits, so there is no necessity to free memory in the middle of a stack. However, not all memory demands are as well-ordered as those created by automatic variables. Sometimes we need the heap.

Heap Memory

Just as the name "stack" suggests an orderly arrangement, the name "heap" suggests a greater degree of disarray. In a heap, memory is assigned in blocks, and assigned blocks can be freed even if they are in the middle of the heap. So the heap can wind up having a jumbled mixture of assigned and unassigned memory. This is what some programming situations need.

For instance, consider the Macintosh window system. You can open several windows at once, shift back and forth between them, and dismiss them in any order you want. There is no last-in, first-out restriction here. Each window has its own memory requirements, for which the stack is unsuited, but for which the heap is ideal. The key advantage to the heap is that it lets a program allocate and deallocate memory in the order needed, not just on a lifo basis.

The disadvantage of the heap is that it can wind up in fragmented chunks. At some point a program might need more memory than is left in one chunk. To combat this possibility, the Memory Manager can move around remaining blocks of memory, consolidating them and creating larger blocks of unassigned memory. See Figure 10.3. The Memory Manager can only do this shuffling for a variety of heap memory called a "relocatable block." A second type of heap memory, called a "nonrelocatable block," stays put as long as your program runs. Of the two, the nonrelocatable block is the simpler, so we'll look at it first.

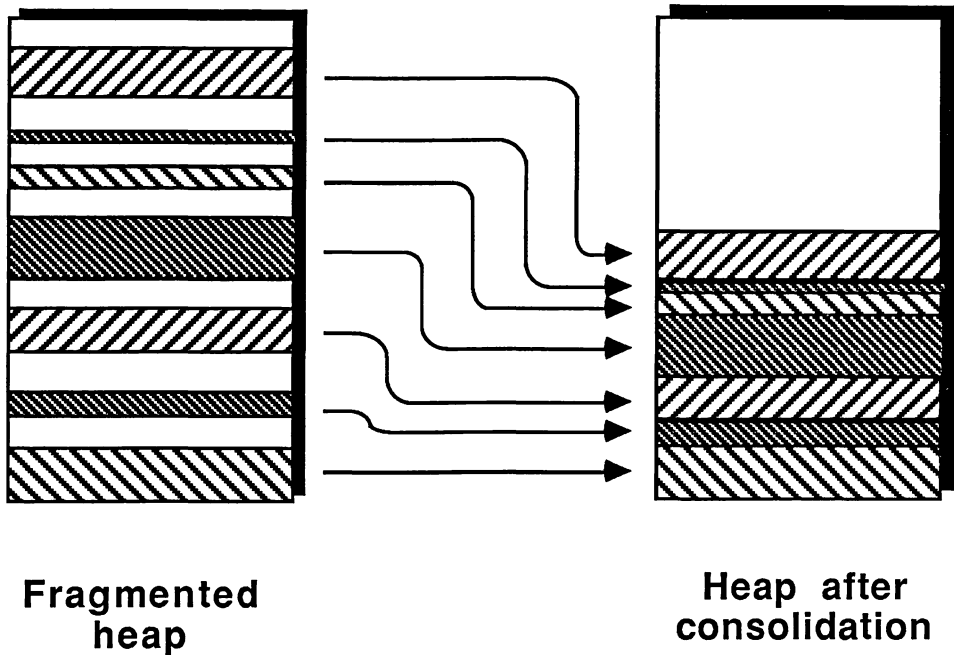


Figure 10.3 Consolidating relocatable blocks

Nonrelocatable Blocks

A nonrelocatable block of heap memory is described by a pointer to the beginning of the block. Since the block remains put, the pointer remains valid for the duration of the program or until the program frees that block. C's **malloc()** (memory allocation) function can be used to obtain a block of the desired size and to return the address of the block. We had an example in Chapter 8, when we used these statements:

```

grafptr  gp;    /* declares a pointer */
char *malloc(); /* gives function type */

gp = (grafptr) malloc( sizeof (grafport) );
/* allocates memory and returns address value */

```

The **malloc()** function takes one argument: the number of bytes of memory required. It returns the address of the first byte of the assigned block of heap memory. The function type is pointer-to-char, as that is the

most elemental pointer type, pointing to just one byte. If the pointer is supposed to describe a larger unit, then you should use a type cast to convert the return value to a pointer to the correct type. The numerical value of the pointer is unchanged, but the interpretation of the pointer is altered. For example, the typecast above makes `gp` a pointer to a `grafport` structure, thus permitting the use of constructions like `gp->portrect`.

The `sizeof` operator is very useful for memory allocation, for it yields the size, in bytes, of the following operand. The operand can be the name of a particular variable or the identifier for a type; if it is a type identifier, it should be enclosed in parentheses, as in the example above.

The Memory Manager has a function (unimplemented in Level 1 Hippo C) called `newptr()` that works just like `malloc()`.

It is important to realize that the program's only link to the allocated block of memory is the pointer. If you assign a new value to the pointer without saving the old value, the program will have no way to use what is in the block, even though the memory is still there.

Let's look at a simple example illustrating where heap stuff gets placed. Here is another program that prints out addresses:

```
char ext;
main()
{
    char aut;
    char *p1, *p2;           /* two pointers to char */
    char *malloc();

    p1 = malloc( sizeof (char) ); /* allocate memory */
    p2 = malloc( sizeof (char) );
    *p1 = *p2 = 'Q';
    printf("Addresses for ext,aut,p1,p2 = %u %u %u %u\n",
           &ext,&aut,&p1,&p2);
    printf("Addresses assigned to p1,p2 = %u %u\n",
           p1,p2);
    printf("Stored values = %c %c\n", *p1,*p2);
}
```

Note the differences between `&p1`, `p1`, and `*p1`. The first is the address of the `p1` variable. Since `p1` is declared at the program head, it is an automatic variable, just like `aut`. But `p1` is the *value* stored at that location,

and it is the address of the newly allocated memory. Finally, *p1 is the value stored in that newly allocated location. Now check out the printout:

```
Addresses for ext,aut,p1,p2 = 36252 106823 106818 106814
Addresses assigned to p1,p2 = 81476 81488
Stored values = Q Q
```

The heap area lies between that used for the external variable and the stack. Note that the heap addresses grow larger as new space is requested. The stack, recall, grows downward. See Figure 10.1. When the two meet, your program is out of memory.

The preceding example was run from the Hippo C Command Window. Note what happens if we run it from HOS (the Hippo Operating System) instead:

```
Addresses for ext,aut,p1,p2 = 36336 106727 106722 106718
Addresses assigned to p1,p2 = 36674 36686
Stored values = Q Q
```

The statically stored ext has the same address as before; the automatic variables have slightly different addresses, and the heap addresses are quite changed. Recall that the program itself is stored in the heap. When we run the program from HOS, the free part of the heap comes right after the end of the program, so the values returned by malloc() are only slightly bigger than the address of ext. When we run the program from the Hippo Command Window, however, a large section of heap is used up for programs and files used in that mode. In that case, the free area of the heap starts at a much larger memory location.

When your program is done with using allocated memory, it should free it. In C this is done using the free() function. Its argument is a pointer previously assigned a value through malloc(). (The Memory Manager equivalent is disposeptr().) Thus, to free the memory assigned in the preceding example, we would use these statements:

```
free(p1);
free(p2);
```

Then later calls to `malloc()` could reuse those memory locations. The locations are said to be returned to the memory pool.

Once again, the problem with nonrelocatable blocks of heap memory is that they can lead to a fragmented heap structure as memory is allocated and freed. So let's look at relocatable memory in the heap.

Relocatable Blocks and Handles. The Macintosh uses a clever scheme to keep track of relocatable memory blocks in the heap. At the same time the relocatable block is allotted, a "master pointer" is also allocated in the heap. The master pointer contains the address of the block, but it (the pointer) is not relocatable. Finally, the address of the master pointer is assigned to a "handle" in the program requesting the memory. A "handle" is just a pointer to a pointer.

What happens if the memory manager decides to move the relocatable block of memory? It moves the block, and places the new address in the master pointer. Note that both the master pointer and the memory block, like other memory locations in the heap, are nameless. The only way to access the block is to use the master pointer, and the only way to access the master pointer is to use the handle. By now, you probably have three questions on your mind. How do you declare a handle in C? How do you access the memory block using a handle? How do you request memory allocation using a handle? Whether or not those questions are actually on your mind, we will answer them now.

First, how do you declare a handle? Suppose, for example, that we have allocated memory for a Pattern structure using relocatable memory. Then the handle would be type pointer-to-pointer-to-struct Pattern. We could declare the handle this way:

```
struct Pattern **handlepat; /* pointer-to-pointer-to-  
                             structure */
```

Or we could make use of Hippo C's `data.h` file, which includes this typedef:

```
typedef struct Pattern pattern, *patptr, **patternhandle;
```

This sets up `pattern` to represent the `struct Pattern` type, `patptr` to represent the type pointer-to-struct Pattern, and `patternhandle` to represent the type

pointer-to-pointer-to-struct Pattern. Then we could declare the handle this way:

```
patternhandle handlepat;
```

In both approaches, the double asterisk (**) ultimately is used to identify a pointer-to-pointer, or handle.

That's how to define a handle in C, but how do we use it? First, suppose we have a handle for a simple type:

```
int **intheandle;
```

If intheandle has been made a handle to some particular integer value (i.e., if intheandle points to a pointer to an int), then we use double indirection to get the value. That is, we can have statements like this:

```
finalscore = oldscore + **intheandle;
```

Thus, **intheandle is the stored value. Also, *intheandle would be the address stored in the master pointer, intheandle would be the address of the master pointer, and &intheandle would be the address where intheandle itself is stored. If the block is moved, then the value *intheandle changes. Figure 10.4 shows the effect on these values of relocating the block.

More typically, the Macintosh system uses handles to structures. Suppose we have this declaration:

```
pathhandle spider;
```

This makes spider a handle to type pattern. Next, as discussed below, spider should be assigned a relocatable block. Now, since spider is a handle, *spider is syntactically the same as a pointer-to-pattern. We can use the indirect membership operator to obtain a structure element using a pointer, so we can refer to, say, the hotspot member of the pattern structure as follows:

```
(*spider)->hotpoint
```

The -> operator has higher priority than *, so we use parentheses to indicate the proper sequence of operators.

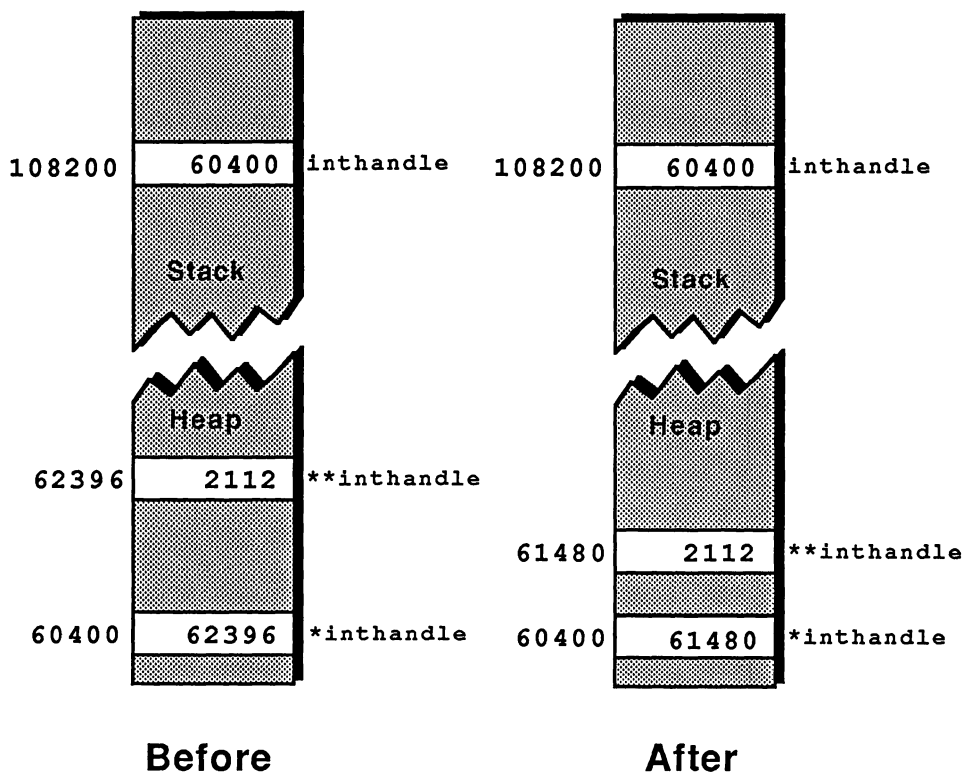


Figure 10.4 Handle, master pointer and block before and after relocation

Finally, how do you get a handle? You need to work through the Toolbox Memory Manager, either directly or indirectly. The direct method involves using the `newhandle()` procedure, which fetches a handle for an arbitrary type. The indirect method is to use Toolbox functions that fetch handles for particular types. For instance, the `newrgn()` function returns a handle to a particular structure type called a region. Because Level 1 of Hippo C does not access `newhandle()`, we will illustrate handles using the second approach. For a context, we will turn once again to Quickdraw.

First, however, let's summarize the methods of accessing heap memory. The first method is to request a pointer to a memory block. In C this is done with `malloc()`. This function allocates a nonrelocatable block of memory in the heap, and returns the beginning address of the block to the calling program; this address is assigned to the pointer. The second method is to request a handle to a memory block. Several Toolbox functions serve this purpose. Each allocates a relocatable block of memory in the heap and a master pointer in the heap. The address of the master pointer is returned to the calling program, where it is assigned to the handle. If the memory block gets moved, the master pointer gets updated.

Handles *sound* complicated because so much has to be done. But most of the work is done by the Memory Manager, and programming using handles turns out to be quite simple. A good example is provided by the Quickdraw treatment of regions. Let's talk about them next.

Regions

The region is a more talented cousin to the rectangle. Recall that in Quickdraw, a rectangle structure is used to describe a rectangular area on the screen. Two kinds of functions are used with the `rect` structure. One set controls the parameters, or attributes, of the rectangle. For instance, `setrect()` sets the coordinates, `offsetrect()` shifts the coordinates, and `insetrect()` modifies the size of the rectangle. A second set of functions causes things to happen on the screen within the confines of the defined rectangle. For instance, `framerect()` draws a boarder, `paintrect()` paints in the rectangle, and `eraserect()` erases the interior of the rectangle.

The region is a similar concept, except that a region is not limited to a rectangular shape. You can create whatever shape you like, within reason, for a region, and then you can manipulate it much as a rectangle is manipulated. That is, a region can be moved, altered in size, framed, painted, erased, and so on.

Because a rectangle is a simple figure, the definition of the rect structure was simple:

```
typedef struct
{
    integer top, left, bottom, right;
} rect;
```

But how can you define a structure to describe an as yet unspecified shape? On the Macintosh, it is done this way in Hippo C:

```
struct Region
{
    integer rgnsz;
    rect rgnbox;
    /* more data if not rectangular */
};
typedef struct Region *rgnptr, **rgnhandle, region;
```

The first structure member, `rgnsz`, is the size of the structure in bytes. The second member is a `rect` structure that encloses the region. Then comes the tricky part, "more data if not rectangular." The Quickdraw package has a method of describing a nonrectangular region with a series of numbers. The `Region` structure gets expanded (or shrunk) as necessary to include that series of numbers. This changing structure size is why the first member of the structure provides the size explicitly. It's much like the Pascal string format, in which the first byte tells how long the whole string is. Here the first member tells how long the whole structure is.

Special Quickdraw procedures generate the numbers to describe a particular region as a program is run. Thus the amount of memory needed for a `Region` structure is determined and can be changed during run time. This is exactly the type of situation for which relocatable heap memory and handles were intended. Consequently, the Quickdraw region functions are written using handles, not regions or pointers to regions. Let's look at some of these functions now.

Creating Regions

To create a region, you need to go through three steps. First, you need to declare a region handle variable. In Hippo C you use the typedef type `rgnhandle`. Next, you allocate and assign a relocatable block of heap memory to hold the region. This is done using the `newrgn()` function. It takes no argument, allocates the memory space initially needed for the region, and returns a region handle value. Third, you create a description for the region. A selection of Quickdraw routines makes this simpler than what you might expect. Let's look at an example that illustrates some of the simpler possibilities. The following program creates a region having the shape of a sideways T. It then performs such operations as painting, offsetting, shrinking, and inverting the region. Naturally, it uses several new Toolbox routines.

From Mac's Toolbox: New Routines

NewRgn	Creates region and returns handle to it
SetRectRgn	Set rectangle bounding a region
FrameRgn	Frame a region
PtInRgn	Returns true if mouse is in region
UnionRgn	Combine two regions
DisposeRgn	Dispose of a region
PaintRgn	Paint a region
InsetRgn	Shrink a region
InvertRgn	Invert contents of a region

Here's the program:

```
/* regions.c -- make some simple regions */
#include "data.h"
#include "stdio.h"
main()
{
    rgnhandle first,second,third; /* declare 3 handles */
    point mouse;
    rect mesg;
    rgnhandle newrgn();           /* declare type for
                                   newrgn() function */

    initcursor();                 /* set arrow cursor */
```

```

eraserect(&theport->portrect); /* clear screen */
setrect(&mesg,15,180,215,205); /* message box */
first = newrgn(); /* allocate first region structure */
second = newrgn();
third = newrgn();
setrectrgn(first,50,50,100,150); /* set first to a
                                   rectangle */
framergn(first); /* show region */
setrectrgn(second,100,80,180,120); /* also set to a
                                   rectangle */
framergn(second); /* show it */
moveto(20,200);
drawstring(strctop("Click the left box"));
getmouse(&mouse);
while ( !button() || !ptinrgn(mouse,first))
    getmouse(&mouse); /* wait for mouse click
                       in region */
unionrgn(first,second,third); /* third combines
                               2 rgns */
disposergn(first);
disposergn(second); /* free memory */
paintrgn(third); /* show new region */
eraserect(&mesg);
framereact(&mesg);
moveto(20,200);
drawstring(strctop("Click this box") );
getmouse(&mouse);
while ( !button() || !ptinrect(mouse,&mesg) )
    getmouse(&mouse);
offsetrgn(third,150,0); /* shift region to right */
paintrgn(third); /* show it */
insetrgn(third,20,5); /* modify region */
invertrgn(third); /* show it */
eraserect(&mesg);
moveto(20,200);
printf("Structure third occupied %d bytes\n",
      (*third)->rgnsize);
}

```

Figures 10.5, 10.6, and 10.7 show three stages of output from the program. Before running the program, we expanded the Hippo C Command Window to accommodate the expected output.

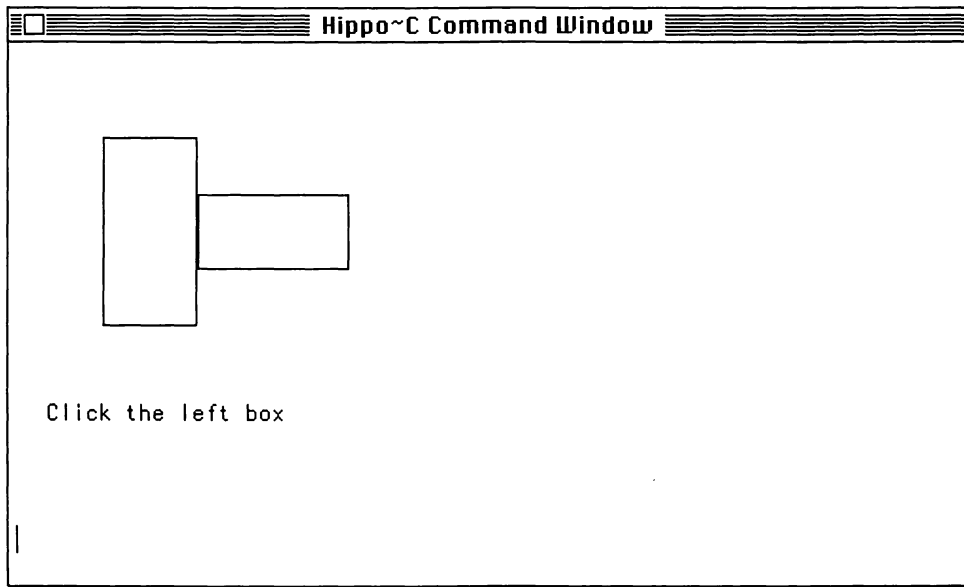


Figure 10.5 Output from regions.c

Let's discuss the new functions we have used. First, as promised, the **newrgn()** function returns a region handle and allocates space for the region structure. Note that we declared the function to be type **rgnhandle**, which is the typedef equivalent of pointer-to-pointer-to-struct **Region**.

Next, the **setrectrgn()** function sets the named region, represented by its handle, to the rectangle described by the next four arguments. If the region were previously defined, the previous definition is wiped out. The rectangle arguments come in the same order as used in the **setrect()** function: left, top, right, bottom. The program sets up the regions first and second to be two adjacent rectangles. (Technically, first is not the name of the region—it is, in fact, nameless—but the handle to the region. But it is much simpler to say "region first" than it is to say "the region whose handle is first." We'll take the simpler way and trust you to understand what we really mean.) The **rgnsize** member of each structure would have the value 10. The **rgnsize** member is one integer itself, and the **rect** structure is another 4 integers. Each integer is 2 bytes, making a total of 10.

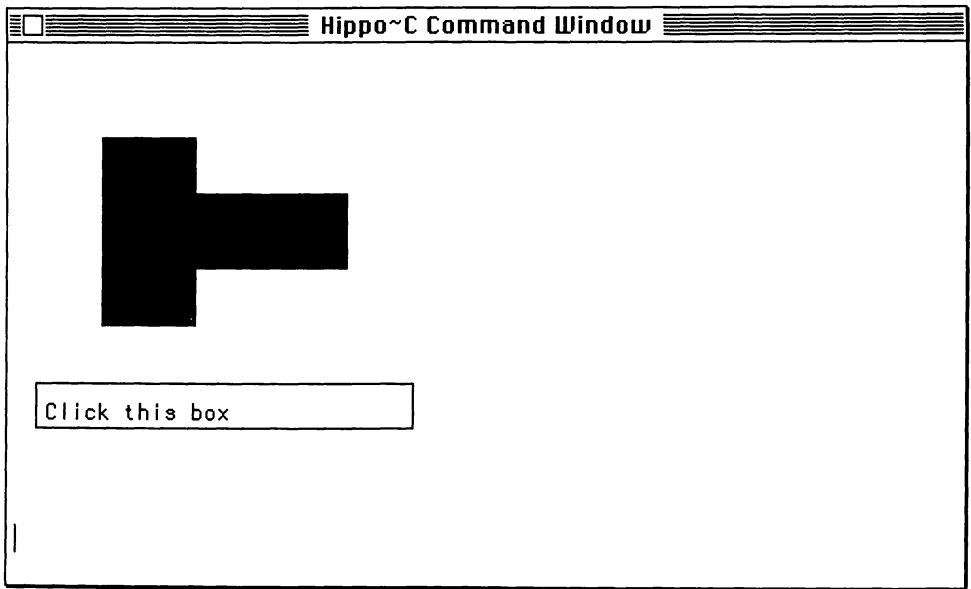


Figure 10.6 Output from regions.c

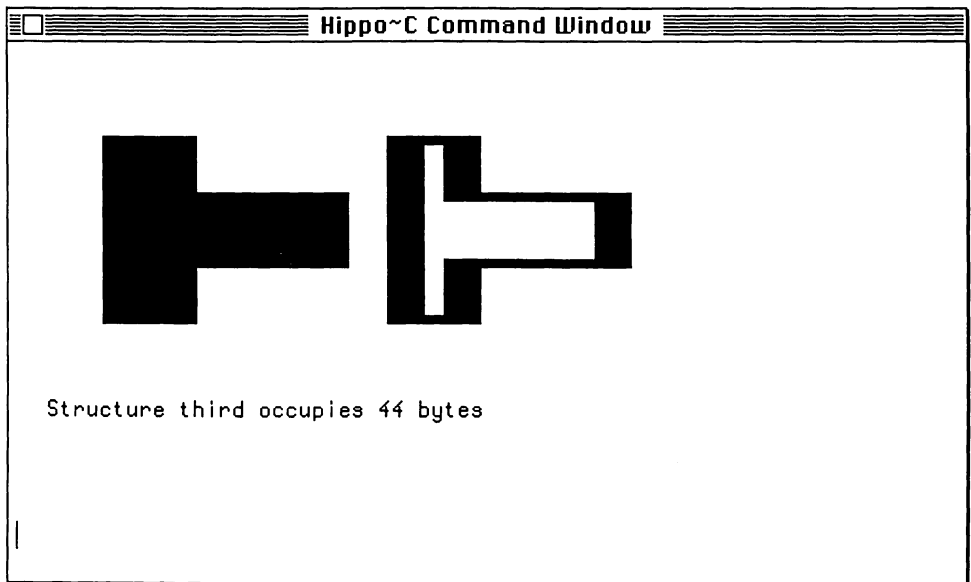


Figure 10.7 Output from regions.c

The **framergn()** function takes a region handle as argument, and draws a border just inside the region boundary. Figure 10.5 shows the program at this stage.

The **ptinrgn()** function works much like the **ptinrect()** function, except that its second argument is a region handle instead of a pointer-to-rect. The function returns a "true" (nonzero) value if the first argument (type point) is in the region, and "false" otherwise. We use it as part of the loop to freeze the screen until the user is ready to advance to the next stage of the program.

Next, the **unionrgn()** function makes third into a region that is the union of first and second. The function takes three arguments. The first two are handles to the regions to be combined, and the third argument is a handle to resulting region. Note that this function does not create storage for the new region; it requires that we had used **newrgn()** earlier for that purpose. Also, the function does not alter first and second. They were used as a source of information for putting together the Region structure for third.

This is our first nonrectangular region and it has the shape of a sideways T. The **paintrgn()** function, which takes a region handle as an argument, fills in the region so that you can see it on the screen. Figure 10.6 captures this stage.

The new region third can be manipulated. For example, the **offsetrgn()** function is used to move it 150 units to the right. This function takes three arguments: a region handle, a horizontal offset, and a vertical offset.

The program uses **paintrgn()** to show the new region location. Then **insetrgn()** is used to change the size of the region. It takes three arguments: a region handle, the horizontal inset, and the vertical inset. All points on the region boundary are moved inward vertically by the vertical inset and horizontally by the horizontal inset. Negative numbers indicate expansion.

Then the program uses **invertrgn()** to invert the new region, making it show up as white against the black background of the preceding version of the region. Figure 10.7 shows the program at this stage.

The figure also exhibits the output of the **printf()** statement, revealing that the structure referred to by third occupies 44 bytes. Indeed, the structure has grown beyond the minimum of 10 bytes needed for a simple rectangular region. The Quickdraw system and the Memory Manager have done all the work for us.

Finally, the **disposergn()** function releases the allocated blocks of memory back to the memory pool. The memory is released anyway when the program ends, but it is a good idea to dispose of a region when you are done with it. That way, your program can use the memory for other needs later.

The **setrectrgn()** and **unionrgn()** functions let you compose regions from rectangles. The arguments to **unionrgn()** need not be limited to contiguous rectangles. For instance, a region could consist of two nontouching rectangles. Or a third could be used as one of the initial arguments to **unionrgn()** and be combined with another region.

Other routines that combine regions are **sectrgn()**, **diffrgn()**, and **xorrgn()**. All, like **unionrgn()**, take three arguments: handles to two "source" regions and a handle to a "destination" region. The **sectrgn()** routine provides a region that is the intersection (or overlap) of the first two regions. The **diffrgn()** function produces the region that results from subtracting the second source region from the first. The **xorrgn()** procedure produces the region consisting of everything that is in one source region or the other, but not in both. All of these procedures require that **newrgn()** be used first to provide memory space for the destination region.

Another useful function is **copyrgn()**. It takes two region handles as arguments and copies the structure of the first into the second. Again, **newrgn()** should be used prior to this function to provide memory space.

These various functions give you many ways to manipulate and modify regions. The most interesting approach, however, is provided by the **openrgn()** and **closergn()** functions.

Designing Regions with openrgn() and closergn()

The **openrgn()** function, which takes no arguments, announces your intention to create a region definition. You then can use **line()**, **lineto()**, and the various framing functions, such as **framerect()**, **frameoval()**, and **framergn()** to create one or more closed loops. These lines and shapes are saved, not drawn on the screen (unless you invoke **pendown()**). Then, a call to **closergn()** organizes the collection of lines and shapes into a region definition. The resulting region is saved in the region indicated by the region handle used as the argument to **closergn()**.

The next example uses this approach.

From Mac's Toolbox: New Routines

OpenRgn
CloseRgn

Prepare to define a region
Use prior instructions to define region

```
/* puff.c uses openrgn() to define a disjoint region */
#include "data.h"
#include "stdio.h"
main()
{
    rgnhandle sample;
    rect box;
    point mouse;
    rgnhandle newrgn();

    initcursor();
    eraserect(&theport->portrect);
    sample = newrgn();
    openrgn(); /* prepare to define region */
    setrect(&box, 40, 50, 100, 90);
    frameoval(&box);
    offsetrect(&box, 0, 50);
    frameoval(&box);
    moveto(20, 200);
    lineto(70, 150);
    lineto(120, 200);
    lineto(20, 200);
    closergn(sample); /* set up region, assign to
                        sample */
    paintrgn(sample); /* see what it looks like */
    offsetrgn(sample); /* move region */
    paintrgn(sample);
    getmouse(&mouse);
    while ( !button() || !ptinrgn(mouse, sample) )
        getmouse(&mouse);
    insetrgn(sample, -10, 5); /* alter shape */
    offsetrgn(sample, 120, 5);
    paintrgn(sample);
}
```

Figure 10.8 shows the appearance of the screen after running this program.

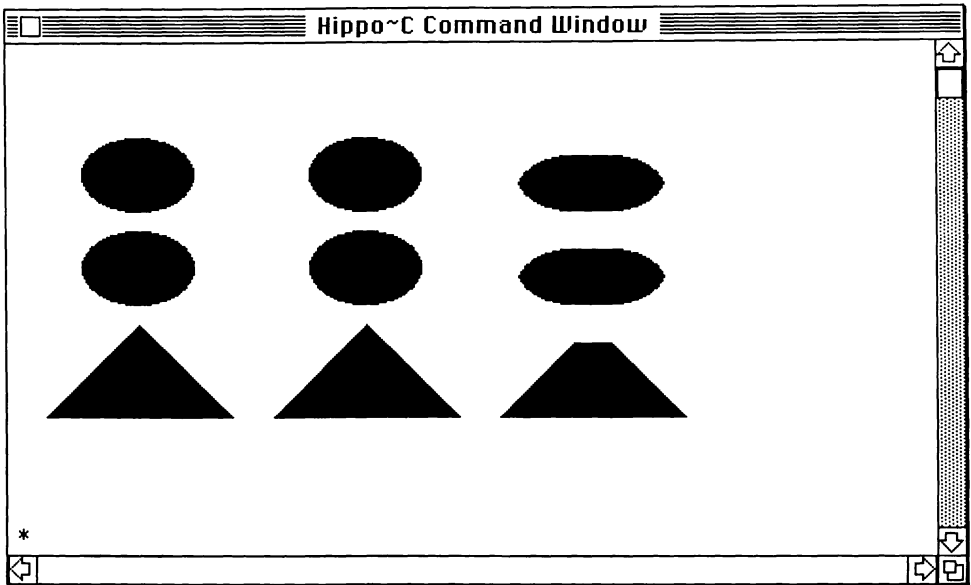


Figure 10.8 **puff.c output**

We indented the portion of the program between `openrgn()` and `closergn()` to make the limits of the defining section obvious. Note that all three parts of the region are affected by the `offsetrgn()` and `insetrgn()` functions. Also, the `ptinrgn()` function will work if the mouse is in any one of the three parts. If you run the program, you'll note that the mouse has to be in one of the parts; simply being inside the boundary rectangle is not enough.

The `insetrgn()` can alter the shape of a region. For example, insetting all boundary points vertically by 5 units while moving each point horizontally by 10 units results in the point of the triangle being truncated.

The main point to notice is how simple it is to use the handles.

Regions are very important to Macintosh programming. For instance, the structure region (include window and frame) of a window is a region, as is the content region, in which the text and graphics appear. If you understand regions, you will find it simple to use the Quickdraw picture and polygon structures, for they are developed along similar lines.

Describing a Region

To use regions you don't really need to know how they are described internally, for the Quickdraw routines take care of the messy details. Still, most of us feel better when we have some idea of what is going on, so we'll describe the method now.

A region is described in terms of its corners. Ultimately, each screen figure is made up of straight line segments. With a curved line, these segments might be only 1 or 2 pixels long, but they are still straight lines.

The descriptive scheme goes like this. Start from the top of the region frame and move down row by row of pixels until you find a row containing one or more corners. Store the vertical coordinate of that row and the horizontal coordinates of all the corners on that row. Then store a 32767 to indicate no more corners in that row. Proceed downward until the next row containing one or more corners, and repeat the process. Continue until you reach the end of the region shape, and store one additional 32767 to mark the end of all the data.

Let's see how this scheme applies to the sideways T we created for region third in `regions.c`. The first row with corners is row 50. The horizontal coordinates of the two corners are 50 and 100. Thus the region structure is expanded to hold the sequence 50 50 100 32767. Going down, the next row with corners has a vertical coordinate of 80. The horizontal corner coordinates are 100 and 180, so the sequence 80 100 180 32767 is stored. Next comes the sequence 120 100 180 32767, and finally the end of the region is represented by 150 50 100 32767 32767. Altogether, that makes 17 integers, or 34 bytes. Add that to the basic 10 bytes any region structure has, and you get 44 bytes, exactly the number reported back by the `regions.c` program when we ran it. Oh yes, the `rgnbox` member of the structure was set by the `unionrgn()` command to the smallest rectangle containing the actual region.

Events

In these last two programs, as well as others, we've used a while loop based on the mouse position to control the stopping and starting of program action. That approach has weaknesses that we've tried to conceal. For example, in `regions.c`, we used two separate areas in which to click the mouse. If, instead, we had used the left box for both click attempts, the program would run so fast that by the time you took your finger off the mouse button, the program would have passed through both click loops. As

a result, the program would go through to the end, and you would miss the intermediate halt. This particular programming can be solved by a more elaborate test that makes sure the button has to be up between while loops, but the Macintosh provides a better approach to controlling the program. That approach is to use the Event Manager.

The "event" is one of the basic concepts in programming for the Macintosh. An event, according to *Inside Macintosh*, is a "notification to an application program of some occurrence that the program must respond to." The occurrences that get reported are happenings such as the mouse button being pressed, the mouse button being released, a key being depressed, a key being released, a window being activated or deactivated, a disk being inserted, an abort message, and a few others.

The event itself is not the occurrence, but a Pascal record (or C structure) created to describe the particular occurrence. Each time you press a key or the mouse button, an event record is created. These are placed in the "event queue," which is the Event Manager's list of pending events waiting to be processed. Each event record contains several items of information, including the time and the type of event. A program can take events off the queue and process the ones relevant to the program.

One motivation for developing the event queue system is the fact that the Macintosh takes input from both the keyboard and from the mouse. The event queue system lets a program deal with both sources of input in a unified fashion.

The key Event Manager function for our purposes is `getnextevent()`. It procures the next event from the system. As each event is processed (or passed over, if unwanted), it is discarded from the queue. If `getnextevent()` fails to find an event when called, it returns a 0 ("false") value; otherwise it returns a true value. But before we can understand its use, we need to understand something about its arguments. Its first argument is type integer and represents an "event mask." That is, it describes the particular events the program wishes to process. The second argument is a pointer to an event record. When an event of a specified type shows up, information about it is transferred to the pointed-to record. We'll look at these two topics (event masks and event records) next.

The Event Mask

The event mask argument is a single number of Pascal type `INTEGER`, equivalent to Hippo C short or to integer as typedefed in the `data.h` file. Since this type is a 2-byte number, we can think of it as 16 1-bit

numbers. That, in fact, is what `getnextevent()` does. Each bit is a "flag" for a particular event; if a particular flag is set to 1, `getnextevent()` will look for the corresponding type of event. Figure 10.9 shows a two-byte integer considered as a mask. It follows the usual computer tradition of numbering the bits 0 to 15, from right to left. Note, for example, that bit 1 is the flag for detecting a "mouse down" event, while bit 3 is the flag for a "key down" event. These will be the only two we will work with.

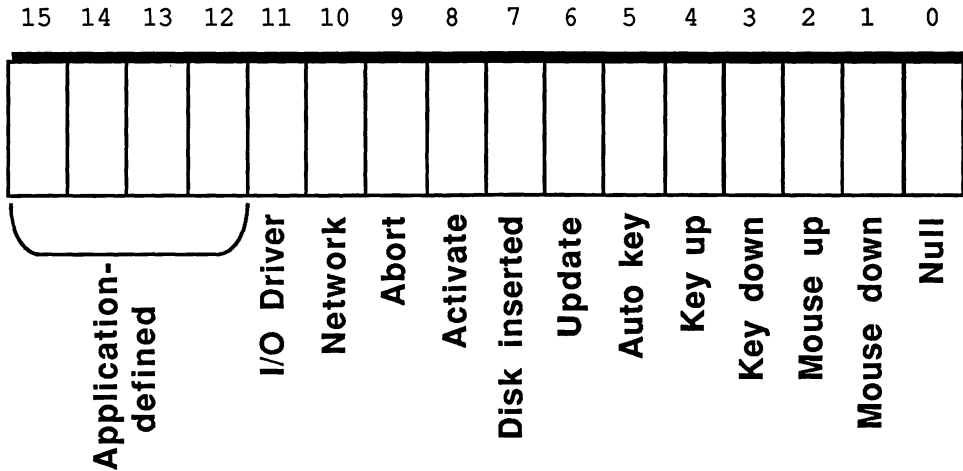


Figure 10.9 The event mask

Next, let's see how to set a flag. When we look at the bit pattern as a binary number, each bit corresponds to a power of two. For example, bit 1 corresponds to 2 to the first power (2), bit 2 corresponds to 2 to the second power (4), bit 3 corresponds to 2 to the third power (8), and so on. So, if `eventmask` is an integer, we can turn on the mouse-down flag by saying

```
eventmask = 2;
```

and we can use

```
eventmask = 8;
```

to turn the key-down flag on. But that also resets bit 1 to 0, turning the mouse-down flag back off. To turn on two or more flags simultaneously,

add their codes together. Thus, to turn on both the mouse-down and the key-down flags, do this:

```
eventmask = 10;
```

This works because 10 in binary (see Appendix C) is 1010, which sets bits 3 and 1 to 1.

To make the system a bit more mnemonic, the defs.h file supplies a list of definitions to use for the different flag settings. Here is a partial list from that file:

```
#define NULLMASK 1
#define MDOWNMASK 2
#define MUPMASK 4
#define KEYDOWNMASK 8
#define KEYUPMASK 16
```

Then, to detect both key-down and mouse-down events, we could use this statement:

```
eventmask = MDOWNMASK + KEYDOWNMASK;
```

The eventrecord Structure

Next, let's look at the structure that `getnextevent()` fills. Here is its Hippo C definition:

```
typedef struct
{
    integer what;
    longint message, when;
    point where;
    integer modifiers;
} eventrecord;
```

The longint type is a data.h equivalent for int.

The first member of the structure is the `what` member; it describes the type of event. Again a numerical code is used to indicate the type of event, and again, the Hippo C `defs.h` file provides mnemonic equivalents. Here are the first few:

```
#define NULLEVENT 0
#define MOUSEDOWN 1
#define MOUSEUP 2
#define KEYDOWN 3
#define KEYUP 4
```

Note that the event code is *not* the same as the event mask code. One reason for this is that the event code is supposed to indicate just one type of event, while the event mask code must be able to indicate several types of events simultaneously. There is a close connection between the two codes, however. The event code for a particular event is just the bit number of the corresponding event mask flag.

Next in the structure comes the message member. It contains information that depends on the type of event. For window-related events, it contains a pointer to the window. For disk-drive events it specifies the drive. For keyboard events, it indicates which key is involved. The low-order byte (bits 0 through 7) contains the character code for the key or key combination. The next byte (bits 8 through 15) contains a key code as defined by Apple. The remaining two bytes are empty. The message member is not used for mouse events.

The `when` member contains the time in ticks (one-sixtieth second) since system startup.

The `where` member contains the position of the mouse when the event occurred. The position is in "global" coordinates rather than in the "local" coordinates used in Quickdraw. Without going into the differences, we will note that `getmouse()`, which is also part of the Event Manager, supplies local coordinates. Hence it is better to use `getmouse()` than the `where` member when you check to see if the mouse is in a rectangle or in a region.

The `modifiers` member indicates the state of the mouse button and of the modifier keys ([`OPTION`], [`CAPSLOCK`], [`SHIFT`], and [`COMMAND`]) when the event is recorded. This member also uses the bit flag approach, with a 1 indicating the key or button that is down. Here is the mapping:

Bit	Key
11	[OPTION]
10	[CAPSLOCK]
9	[SHIFT]
8	[COMMAND]
7	Mouse Button

As you can see, the event record contains a wealth of information. Usually, however, your program will need to use only some of the information.

Using getnextevent()

Let's make a very elementary use of `getnextevent()`. Suppose we want a program to wait for the mouse button to be pushed down. Then we can ask `getnextevent()` to look only for mouse down events. For example, we can use the following elements:

```
#include "data.h" /* definition of eventrecord */
#include "defs.h" /* event and event mask definitions */
...
eventrecord event;
...
while ( !getnextevent(MDOWNMASK, &event) )
    ; /* wait for next mousedown */
```

The `getnextevent()` function will keep examining and discarding events until a mouse-down event reaches the head of the queue. When the next event is not a mouse-down event or when the event queue is empty (the null event), the function returns a "false" value, and the loop continues to cycle. The only use this makes of event is to provide `getnextevent()` with the proper arguments. Instead, the return value of the function is used to control the program flow.

Let's try out this construction.

From Mac's Toolbox: New Routines

GetNextEvent Screens event queue for selected events

```
#include "data.h"
#include "defs.h"
main()
{
    rect box;
    eventrecord event;
    int loop;

    initcursor();
    erasect(&theport->portrect);
    setrect(&box,20,20,490,300);
    framerect(&box);
    for ( loop = 0; loop < 10; loop++)
    {
        insetrect(&box,20,10);
        while ( !getnextevent(MDOWNMASK, event) )
            ; /* wait for mouse down */
        framerect(&box);
    }
}
```

This program draws a sequence of nested rectangles; you should expand the Hippo C Command Window to full screen to accommodate the largest rectangle. The point of the exercise is that after drawing each rectangle, the program stops and doesn't continue until you push the mouse button. Thus, each push of the mouse button produces a new rectangle until the program ends.

Suppose we had used our old standby instead:

```
while ( !button() )
    ;
```

The first time you push the button, all the remaining rectangles get drawn. The reason, as we said earlier, is that the Macintosh is much faster than your fingers. Before you release the mouse button, the Mac manages to go through the entire for loop. Every time it reaches the while loop, the button is still down.

With the `getnextevent()` approach, we avoid that problem. As long as you hold the button down, no more mouse events are created. Mouse events only occur when you press the button and when you release the button. Thus the next mouse-down event is generated only after you release the button and press it again. One way of expressing the difference is that `getnextevent()` reports *events*, or changes of state, while `button()` reports a *condition*.

Using the what Member

One important method of utilizing `getnextevent()` is to use the `what` member of the event structure to guide the program flow. For instance, the program can take one course of action if the mouse button is pushed, and a separate course if a key is pushed. Here is a template for this approach:

```
#include "data.h"
#include "defs.h"
...
eventrecord event;
integer eventmask = MDOWNMASK + KEYDOWNMASK;
...
while ( !getnextevent(eventmask, &event) )
{
    switch (event.what)
    {
        case MOUSEDOWN:    ...
                           break;
        case KEYDOWN:      ...
                           break;
        default:           ...
    }
}
```

When an event selected by the event mask shows up, the switch statement selects a course of action based on the value of the `what` member. There should be a case selection for each kind of event specified in the event

mask. The default case shouldn't show up, but it is good practice to program defensively.

The next program uses this device. It places a framed circle in the middle of the screen. When you click the mouse button, the program paints in the circle, draws a new framed circle at the location of the mouse, and connects the two with a straight line. This continues until you strike a key. It's a simple-minded concept, but it illustrates how to blend mouse and keyboard input to control a program. The program creates its own grafport so that it will have the whole screen to work with. Here is the listing:

```
/* balldraw.c -- chase the cursor */
#include "data.h"
#include "defs.h"
#include "stdio.h"
#define HALFH 10      /* half width of ball */
#define HALFV 10      /* half height of ball */
#define PENW 4         /* pen size */
#define FALSE 0
#define TRUE 1
main()
{
    integer hor,ver;          /* coordinates */
    integer eventmask = MDOWNMASK + KEYDOWNMASK;
    eventrecord event;
    rect box;
    grafptr gp, gpsave;
    point mouse;
    integer done = FALSE;    /* is the program done? */
    char *malloc();
    gpsave = theport;        /* save Hippo C's grafport */
    gp = (grafptr) malloc ( sizeof (grafport) );
    openport(gp);            /* set up own grafport */
    initcursor();
    pensize(PENW,PENW);
    eraserect(&gp->portrect);
    framerect(&gp->portrect);
    hor = (gp->portrect.left + gp->portrect.right) / 2 ;
    ver = (gp->portrect.top + gp->portrect.bottom) / 2 ;
    setrect(&box,hor-HALFH,ver-
HALFV,hor+HALFH,ver+HALFV);
    moveto(hor-PENW/2,ver-PENW/2); /* offset for pen
                                   width */
    frameoval(&box); /*draw initial circle at screen
                    center */
    while ( !done )
    {
```

```

while ( !getnextevent(eventmask, &event) )
;    /* wait for an event */
switch (event.what)
{
case MOUSEDOWN : getmouse(&mouse);
hor = mouse.h;
ver = mouse.v;
lineto(hor-PENW/2,ver-PENW/2);
paintoval(&box); /*fill in old ball */
setrect(&box,hor-HALFH,ver-HALFV,
hor+HALFH,ver+HALFV);
frameoval(&box);
break;    /* break from switch */
case KEYDOWN : done = TRUE; /* sign to quit */
erasetrect(&gp->portrect);
break;
case default : done = TRUE;
erasetrect(&gp->portrect);
moveto(20,50);
drawstring(
    strtoc("Something's wrong!") );
break;
}
}
closeport(gp);
setport(gpsave);
}

```

Figure 10.10 shows some sample output from the program.

The program uses elements that we have discussed before, so there is not much that needs to be said. One point that may seem obscure is the `PENW/2` offset we used for the `moveto()` and `lineto()` statements. The reason for this usage is that the upper left corner of the pen is aligned with the coordinates in a `lineto()` or `moveto()` call. The offset is used to bring the center rather than the corner of the pen pattern near the center of the circle.

In discussing the Event Manager, we have concentrated on `getnextevent()`. We have, however, used other functions from that manager: `button()`, `getmouse()`, and `tickcount()`. This is not a complete list; but, as we have confessed before, we just can't hope to cover the whole Toolbox.

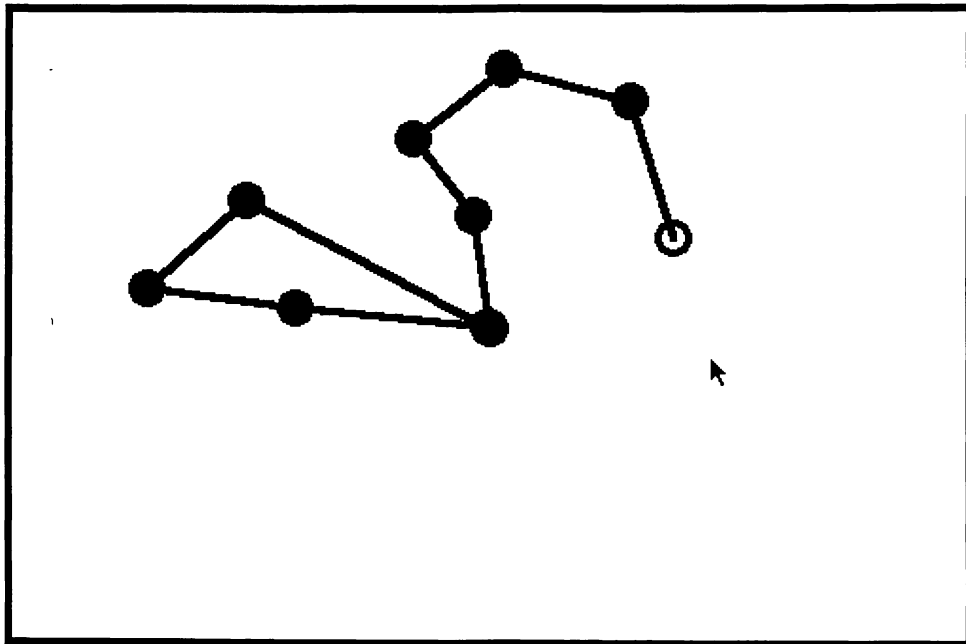


Figure 10.10 Output from `balldraw.c`

Files

Files are another topic we can't discuss in entirety. Macintosh files store many varieties of material, including programs, text documents, and graphics documents. Apple chose to structure each file into two parts: a "data fork" and a "resource fork." Although both parts are always present, one or both can be empty.

The data fork contains the text in a text file, data for a spreadsheet program, and the like. The resource fork contains programming, if any, in the 68000 machine language used by the Macintosh. It may contain other resources, such as font information or window data, to be used by a program.

The File Manager handles the creation and manipulation of files, and it does so to a degree of detail that we don't wish to go into. Instead, we will look at the basics of opening and using a file in C, and not worry about the elaborations required to make a program work like a standard Macintosh application.

Our simple-minded goal is to write a program that requests the user to supply a filename and then counts the number of bytes in that file. To do that, we need to learn about C's file-handling system.

C and Files

Most C implementations provide a package of file-related functions called the standard I/O package; the standard file `stdio.h` supports this package with shared definitions used by the functions. A key definition is of type `FILE`. In Hippo C, it is defined this way:

```
typedef struct
{
    int *file;
    char flag, type;
} FILE;
```

The key point is that `FILE` is a structure containing information about a file. For instance, the members `file` and `type` identify which file and what kind of file it is. Many C I/O functions, when dealing with a file, identify it not by name, but by a pointer to a `FILE` structure. Fortunately, we don't have to specify the structure contents ourselves; that's taken care of by the function that opens the file.

Let's get more specific. Suppose you want your program to open the data fork of a file called `myfile` and read what is in it. Then you use the `fopen()` function. It takes two arguments. The first is a pointer to the name of the file to be opened, and the second is a string indicating what is to be done with the file. Here are the choices for the second argument:

Argument	Meaning
"r"	read only (no modifications allowed)
"r+"	read/write
"w"	delete existing file and open write only
"w+"	delete existing file and open read/write
"a+"	append in read/write mode

The `fopen()` function has a return value, a pointer-to-a-FILE structure set up to describe the requested file. If `fopen()` fails to open a file, it returns a pointer-to-NULL.

The basic call to `fopen()`, then, would have these elements:

```
FILE *fp, *fopen(); /* declare pointer, function type */
...
fp = fopen("myfile", "r");
```

Usually, the value of `fp` will be compared with `NULL` to see if the file was opened successfully.

Once a file is opened for reading, it can be read by file input functions. The input functions we've used before all have file-reading equivalents. The names are changed slightly, and an argument (a pointer-to-FILE) is added to specify the file to be read. For example, in Hippo C we could use any of these statements for the "myfile" file:

```
ch = fgetc(fp); /* getchar() analog */
fscanf(fp, "%d", &number); /* scanf() analog */
fgets(line, 81, fp); /* we've seen this one already */
```

We've used `fgets()` before, using `stdin` to specify that it read the standard input. It turns out that `stdin` and `stdout` are predefined (in `stdio.h`) identifiers of type pointer-to-FILE. Thus, all these functions can be used with standard input, too, just by using `stdin` as the file identifier.

Incidentally, many implementations use the name `getc()` instead of `fgetc()` for the character-reading function. For compatibility, Hippo C defines the two names as being equivalent in `stdio.h`.

After processing a file, use `fclose()` to close the file. Again, the argument should be the pointer provided by `fopen()`.

Here is a short program illustrating these fundamental points.

```
/* bytecount.c -- counts bytes in a file */
#include "stdio.h" /* absolutely essential */
main()
```

```

{
    int bytes = 0;
    char filename[81];
    FILE *fp;
    FILE *fopen();

    fputs("Enter name of file whose bytes are to be
        counted:\n", stdout);
    scanf("%80s", filename);
    if ( (fp = fopen(filename, "r")) == NULL )
    {
        printf("Can't open the file %s\n", filename);
        exit();      /* quit if in trouble */
    }
    while ( fgetc(fp) != EOF )
        bytes++; /* count bytes till EOF */
    close(fp);
    printf("File %s contains %d bytes\n",
filename, bytes);
}

```

In interpreting the if condition, recall that the value of an assignment expression is the value of the left-hand member. Thus, the if condition first assigns a value to the pointer `fp` and then compares that value to `NULL`. If it is `NULL`, then `fopen()` failed. In that case, the program prints a message and exits. (The `exit()` function causes a program to terminate in a tidy fashion. Any opened files get closed.)

Since the program doesn't do anything with the bytes it reads aside from checking for EOF, we didn't assign `fgetc()`'s return value to a variable.

Here is a sample run:

```

* a.out
Enter name of file whose bytes are to be counted:
data.h
File data.h contains 4731 bytes

```

Notice that we had to run the program from a keyboard-oriented environment. A truly Mac-like program would use windows and menus to arrange file selection and to report output, but the inner core of the program could be the same.

We sampled the file input functions, now let's list the output functions briefly, indicating the arguments; `fp` will indicate a pointer-to-FILE.

```
fprintf(fp,format,arguments) /* works like printf() */
fputs(line,fp)                /* a familiar function */
fflush(fp)                    /* flushes output buffer
                               to fp */
```

We discussed buffered output back in Chapter 3. Normally, the output functions send output to an intermediate buffer. The contents of the buffer are sent on to the file when the buffer fills or when a newline is transmitted. The `fflush()` function lets you force transmission at any time.

Binary I/O

The I/O functions we've discussed do text I/O. The character functions transmit a single character, and the other functions transmit strings of characters. Even when you use `printf()` or `scanf()` in the `%d` mode, they work with characters. For instance, when you enter the number 234, you separately type the characters "2", "3", and "4". The `scanf()` function reads them as a character string, then converts them to the binary number that finally gets stored. Similarly, the `printf()` function sends a series of characters to the screen.

When the file output functions we've discussed are used to write to files, they create *text* files, files consisting of a sequence of characters. Similarly, the input functions we've discussed are designed to read text files. However, C also offers functions to let a program read and write *binary* files.

A binary file is one that stores data in the same form that it is stored in a program. For a character, there is no difference, since a character occupies one byte in either case. For numerical data, however, there is a difference. For example, all short integers on a Macintosh occupy two bytes of memory. If we store a short integer in a binary file, it will occupy two bytes of file space. If we store a short integer in a text file, however, it can occupy from one byte (as does the number "3") to six bytes (as does the number "-23224"), depending on the number of characters needed to represent the number.

The C binary output function is called `fwrite()`. It takes four arguments. The first argument is a pointer to the location in program memory from which it is to start taking data. The second argument is the size, in

bytes, of the chunk of memory it is to write. The third argument is the number of chunks, and the final argument is a pointer-to-FILE identifying the target file. Suppose, for instance, we wanted to store the contents of an array of ints. We could do something like this:

```
int nights[1001];
FILE *fp, *fopen();
...
fp = fopen("scheher", "w"); /* ignore error checking */
fwrite(nights, sizeof(int), 1001, fp);
```

First, `nights`, being the name of an array, is a pointer to the first element of the array. Next, `sizeof(int)` indicates the size of the unit to be read. Then 1001 indicates how many of these units, and `fp` specifies the file. If this fragment were run under an implementation that used a different size for `int`, it would still run because it explicitly checks for the `int` size. Incidentally, instead of using 1001 `int` chunks we could have used just one huge chunk:

```
fwrite(nights, 1001 * sizeof(int), 1, fp);
```

One advantage of using a binary file is that data can be recovered easily by using the same format for reading that was originally used to write the file. (This is not the case when you use `fprintf()` and `scanf()`.) The binary read function is called (surprise!) `fread()`. It, too, takes four arguments with more or less the same significance. The main difference is that the first argument is a pointer to the initial address of the block of program memory where the data is to be placed. For example, to recover data from the `scheher` file, we could do this:

```
int tales[1001];
FILE *pf, *fopen(); /* no law says you have to use fp */
...
pf = fopen("scheher", "r");
fread(tales, sizeof(int), 1001, pf);
```

Just be sure that you have allocated enough memory to hold what is read.

Sound Programming

An interesting aspect of the Macintosh is that it treats device drivers (programs that run devices) as files. For example, if we wish to run the speaker, we do so by opening a "file" called .sound. Once the file is open, we can use `fwrite()` to send instructions to sound the speaker. Before we rush to open that file, however, we should look into what kind of instructions we can send.

The sound driver expects a sequence of short (INTEGER) integers. These are normally arranged in a structure, but the form of the structure is variable. In all cases, however, the first member of the structure is a mode integer instructing the driver which sound mode to use.

Sound Modes. The sound driver has three modes: square wave, free form, and four tone. These are indicated by negative mode number, zero mode number, and positive mode number, respectively.

The square wave mode produces a buzzy sound quality that many associate with electronic noise makers. It is the simplest kind for the computer to produce. It just has to provide a constant voltage for fixed periods of time to the speaker.

The four tone mode produces a more musical tone. In fact, it produces four of them simultaneously, each with its own distinct characteristics. Since each tone requires the computer to provide rapidly varying voltages, this mode demands more time of the computer. It also requires more programming effort.

The free form mode lets you design the sound quality. It is intended to synthesize music and speech. It, too, is more demanding of the computer and of the programmer.

The Square Wave Structure. We'll follow the simplest path for computer and programmer and develop a square wave example. The appropriate structure begins with the mode integer, which should be negative; -1 will do fine. Next, the structure should contain one or more packets of three short integers. This can be handled by a substructure of the following form:

```
struct tone
{
    short count, amplitude, duration;
};
```

Here count determines the pitch of the note, amplitude determines the loudness, and duration determines the duration. We'll return to the scaling of these parameters in a moment. But first, here are two possible structure forms for transmitting to the sound driver:

```
struct squarewave
{
    short mode;
    struct tone shape;
};      /* sends one note to the sound driver */

struct squarewaves
{
    short mode;
    struct tone shapes[24];
};      /* sends 24 notes to the sound driver */
```

We'll use both forms soon. First, however, let's examine the three members of the tone structure.

First, the count member determines the pitch, or frequency of the note—that's the number of oscillations per second. Human hearing covers the range from 20 Hertz to 20,000 Hertz. (The Hertz is a frequency unit corresponding to what used to be called a cycle per second.) Voice sounds are usually in the range 200 Hertz to 800 Hertz. The count determines the speaker frequency according to this formula:

$$\text{frequency} = 783360 / \text{count}$$

However, the speaker can't reproduce some extreme frequencies that you may feed it.

Next, the amplitude member can range from 0 to 255, with the larger numbers being louder.

Finally, the duration member gives the note duration in ticks; remember that 1 tick is one-sixtieth of a second.

Using Square Waves. We've put together a program that makes naive use of the sound driver. Instead of carefully selecting count values to represent specific notes, it uses a loop to assign progressively increasing

values; this translates to progressively lower notes. The loop index is also used to vary the amplitude and duration members to add variety.

We've run the driver in two ways. First, we use the squarewave structure to describe a single note. A for loop revises the description and uses fwrite() to send a sound request each cycle of the loop. Meanwhile, the loop also stores up the separate descriptions in the array of tone members of the squarewaves structure. Then, after the loop is finished, all 24 notes are sent in one fwrite() call. Although the 24 fwrite() calls in the loop and the single fwrite() call after the loop send the same note information to the driver, the results sound different. The reason is that in the first case various program steps, including separate fwrite() calls, are executed between each note.

Here is the program; run it and hear it:

```
/* soundoff.c -- make some noises */
#include "stdio.h"
#define SWMODE (-1)    /* square wave */
#define TIMES 24
struct tone
{
    short count, amplitude, duration;
};
struct squarewave
{
    short mode;
    struct tone shape;
};
struct squarewaves
{
    short mode;
    struct tone shapes[TIMES];
};
main()
{
    short loop;
    struct squarewave wave;
    struct squarewaves waves;
    FILE *fp, *fopen();

    if ( (fp = fopen(".sound", "w") ) == NULL )
    {
        printf("Can't open sound driver\n");
        exit();
    }
}
```

```

waves.mode = wave.mode = SWMODE;
for ( loop = 1; loop <= TIMES; loop++)
{
    waves.shapes[loop - 1].amplitude =
        wave.shape.amplitude = 80 * (loop % 3 + 1);
    waves.shapes[loop - 1].duration =
        wave.shape.duration = loop;
    waves.shapes[loop - 1].count =
        wave.shape.count = 350 * loop;
    fwrite(&wave, sizeof(struct squarewave), 1, fp);
}
fwrite(&waves, sizeof(struct squarewaves), 1, fp);
fclose(fp);
}

```

Note that we made use of the fact that C allows us to use the assignment operator more than once in a statement. Note, too, that each call to `fwrite()` sends just one structure to the driver; the final call, however, sends a larger structure.

A Soundmouse Experiment

Let's tie some of the elements of this chapter together by using the mouse to control sounds. Here is the plan. Create a new grafport with a nice clean screen. Use `getnextevent()` to look for mouse-down and key-down events. If the mouse button is pressed down, find out where the mouse is and use its coordinates to control the loudness and pitch of the sound. Have the sound continue as long as the button is held down. If a key is pressed, have the program halt.

We have already developed nearly all the necessary elements for this approach. The one missing ingredient is the `waitmouseup()` function from the Event Manager. It is called after a mouse-down event and returns "true" if the mouse button is still down as a result of that particular mouse event. If the button is not down, the function removes the corresponding mouse-up event from the queue and returns "false."

From Mac's Toolbox: New Routines

WaitMouseUp True if button still down after last mouse event

Here is the program listing:

```
#include "data.h"
#include "defs.h"
#include "stdio.h"
#define SWMODE (-1)
#define QUARTERSECOND 15
#define FALSE 0
#define TRUE 1

struct tone
{
    short count, amplitude, duration;
};
struct squarewave
{
    short mode;
    struct tone shape;
};

main()
{
    integer eventmask = MDOWNMASK + KEYDOWNMASK;
    eventrecord event;
    grafptr gp, gpsave;
    point mouse;
    boolean done = FALSE;
    FILE *fp, *fopen();
    char *malloc();
    void soundmouse();

    gpsave = theport;
    gp = (grafptr) malloc( sizeof(grafport) );
    openport(gp);
    eraserect(&gp->portrect);
    initcursor();
    if ( (fp = fopen(".sound", "w") ) == NULL )
    {
        printf("Can't open sound driver\n");
        exit();
    }
    while ( !done)
    {
        while ( !getnextevent(eventmask, &event) )
            ; /* wait for mouse down or key down */
    }
}
```

```

switch(event.what)
{
    case MOUSEDOWN : while ( waitmouseup() )
                        {
                            getmouse(&mouse);
                            soundmouse(mouse.h,mouse.v,fp);
                        }
                        break;
    case KEYDOWN   :
    default        : done = TRUE;
                        break;
    } /* end of switch */
} /* end of while */
fclose(fp);
closeport(fp);
setport(gpsave);
}

void soundmouse(h,v,fp)
short h,v;
FILE *fp;
{
    struct squarewave wave;

    wave.mode = SWMODE;
    wave.shape.duration = QUARTERSEC;
    wave.shape.amplitude = 255 - v * 30 / 41;
    wave.shape.count = 400 + 7 * (512 - h);
    fwrite(&wave,sizeof (struct squarewave), 1, fp);
}

```

Note that we opened the sound driver in the main program rather than in `soundmouse()`. Thus we had to pass the `FILE` pointer to `soundmouse()` so that it would know where to write. Putting the file opening in `soundmouse()` would require opening and closing the file every function call, a rather inefficient way of doing things.

The formulas used to assign values to the amplitude and count members were devised to produce reasonable values for possible mouse positions. With this setup, low pitch comes from mouse to the left, and low volume comes from mouse to the bottom of the screen.

If you move the mouse while keeping the button pressed, the pitch will change, updating the mouse position (and hence note) before each call to `soundmouse()`. Or you can move with the button up, producing separate noises.

This program gives you plenty of opportunity for play. You can superimpose a pattern of keys on the screen and set the pitch according to which key the cursor is in. You can alter the cursor to a note or a finger tip. You can have a small circle indicate the current and previous note. You can develop your own possibilities.

The earlier versions of the Macintosh Toolbox did not include routines for the sound driver, but the later versions do. However, at the time this is written, Hippo C, Level 1 does not support those Toolbox routines. Nonetheless, the standard C `fwrite()` function lets you make music anyway.

Summary

The Macintosh software system is organized into several managers, each with particular areas of responsibilities. Three managers often drawn upon by the others are **Quickdraw** (the screen manager), the **Memory Manager**, and the **Event Manager**.

Macintosh programs use three kinds of memory: **static**, **stack**, and **heap**. Static memory is set at compilation time and is used for external and static variables. Stack memory is used for the automatic variables generated by functions as they are called. When a function dies, the stack memory it used is freed. Heap memory is used for memory allocated dynamically as the program runs. Nonrelocatable heap memory is referenced using a pointer value returned by C's `malloc()` function. Relocatable heap memory is referenced through a handle value returned by an appropriate Toolbox function. When heap memory gets too fragmented, the Memory Manager rearranges relocatable blocks to open up larger blocks of free memory. Thus relocatable blocks are the preferred form for many kinds of structures.

The Quickdraw region structure offers a good example of the use of **handles**. It is also of interest because it lets you define regions of arbitrary shape and perform operations upon them.

The Event Manager keeps track of various input events (mouse, keyboard, disk drive, and so on), places them in a queue and makes them available to a program in an organized, coordinated fashion. The `getnextevent()` function lets a program screen events, accepting only those that concern it.

Files can be open, read, and written to using the C standard I/O library. Device drivers, such as the sound driver, are treated as files.

Conclusion

You have come a long way, both in C and in Macintosh lore, since starting this book. You have seen most of the major features of the C language. The only major topic not covered is C's ability to operate directly on individual bits. Since Pascal lacks that ability, it is not required for the use of Toolbox routines. If you are interested in the subject, please read Appendix D, which covers C's bit operations.

What you have seen of the Macintosh, however, only scratches the surface of its wealth of routines. But the parts you have seen are perhaps the most essential ones, and understanding them greatly facilitates learning the rest of the system.

What comes next? You can study Apple's *Inside Macintosh* manual. Not only does it describe all the toolbox routines, it also discusses each of the event managers. Other books, such as S. Chernicoff's *Macintosh Revealed*, (volumes one and two) (Hayden, 1985) provide a condensed version of the manual. These books, however, do not go deeply into describing how to put the pieces together to program the Macintosh. For a guide to Macintosh programming, you can try Christopher L. Morgan's *Hidden Powers of the Macintosh* (New American Library, 1985). This book uses a small subset of Toolbox routines, but systematically develops the use of the various managers and shows how to put programs together.

These books are written from a Pascal viewpoint. But the experience you've gained in working with the Toolbox and the guidelines we've provided for making the transition from Pascal to C put you in a good position to continue your education. Good luck, and good programming!

Appendices

- **Addendix A : Keywords in C**
- **Appendix B : Operators**
- **Appendix C : Binary, Octal, and Hexadecimal Numbers**
- **Appendix D : Bit Fiddling**
- **Appendix E : Macintosh ASCII Table**

A

Keywords in C

The keywords of a language are the words used to express the actions of that language. The keywords of C are reserved; that is, you can't use them for other purposes, such as for the name of a variable.

Program Flow Keywords

Loops: `for` `while` `do`

Decision and Choice: `if` `else` `switch` `case` `default`

Jumps: `break` `continue` `goto`

Data Types

`char` `int` `short` `long` `unsigned`
`float` `double` `struct` `union` `typedef`

Storage Classes

`auto` `extern` `register` `static`

Miscellaneous

`return` `sizeof` `asm` `endasm`

B

Operators

C is rich in operators. Here we will present a list grouping them by class. Next, we will summarize the operators except for the bitwise operators, which are discussed in Appendix D. Finally, we present a table of operators, indicating the priority ranking of each and how each operator is associated.

The C Operators

Arithmetic Operators:	<code>+</code> <code>-</code> <code>*</code> <code>/</code> <code>%</code> <code>++</code> <code>--</code>
Assignment Operators:	<code>=</code> <code>+=</code> <code>-=</code> <code>/=</code> <code>*=</code> <code>%=</code>
Relational Operators:	<code><</code> <code><=</code> <code>==</code> <code>>=</code> <code>></code> <code>!=</code>
Logical Operators:	<code>&&</code> <code> </code> <code>!</code>
Pointer-related Operators:	<code>&</code> <code>*</code>
Structure and Union Operators:	<code>.</code> <code>-></code>
Bitwise Operators:	<code>&</code> <code> </code> <code>~</code> <code>^</code> <code>>></code> <code><<</code> <code> =</code> <code>&=</code> <code>^==</code> <code>>>=</code> <code><<=</code>
Grouping Operators:	<code>()</code> <code>{}</code>
Miscellaneous Operators:	<code>sizeof</code> <code>,</code> <code>(type)</code> <code>?:</code>

I. Arithmetic Operators

- +** Adds value at its right to the value at its left
- Subtracts value at its right from the value at its left
- As a unary operator, changes the sign of the value to its right
- *** Multiplies value at its right by the value at its left
- /** Divides value at its left by the value at its right. Answer is truncated if both operands are integers
- %** Yields the remainder when the value at its left is divided by the value to its right (integers only)
- ++** Adds 1 to the value of the variable to its right (prefix mode) or of the variable to its left (postfix mode)
- Like ++, but subtracts 1

II. Assignment Operators

- =** Assigns value at its right to the variable at its left

Each of the following operators updates the variable at its left by the value at its right, using the indicated operation. We use r-h for right-hand, l-h for left-hand.

- +=** adds the r-h quantity to the l-h variable
- =** subtracts the r-h quantity from the l-h variable
- *=** multiplies the l-h variable by the r-h quantity
- /=** divides the l-h variable by the r-h quantity
- %=** gives the remainder from dividing the l-h variable by the r-h quantity.

Here is an example:

`frogs *= 2;` is the same as `frogs = frogs * 2;`

III. Relational Operators:

Each of these operators compares the value at its left to the value at its right. The relational expression formed from an operator and its two operands has the value 1 if the expression is true and the value 0 if the expression is false.

<code><</code>	less than
<code><=</code>	less than or equal to
<code>==</code>	equal to
<code>>=</code>	greater than or equal to
<code>></code>	greater than
<code>!=</code>	unequal to

IV. Logical Operators

Logical operators normally take relational expressions as operands. The `!` operator takes one operand, and it is to the right. The rest take two: one to the left, one to the right.

<code>&&</code>	Logical AND: the combined expression is true if both operands are true, and it is false otherwise.
<code> </code>	Logical OR: the combined expression is true if one or both operands are true, and it is false otherwise.
<code>!</code>	Logical NOT: the expression is true if the operand is false, and vice versa.

VI. Pointer-Related Operators

- &** The address operator: when followed by a variable name, gives the address of that variable:

`&nanny` is the address of the variable `nanny`

- *** The indirection operator: when followed by a pointer, gives the value stored at the pointed-to address:

```
nanny = 22;
ptr = &nanny; /* pointer to nanny */
val = *ptr;
```

The net effect is to assign the value 22 to `val`.

VI. Structure and Union Operators

- .** The membership operator (the period) is used with a structure or union name to specify a member of that structure or union. If `name` is the name of a structure and `member` is a member specified by the structure template, then `name.member` identifies that member of the structure. The membership operator can also be used in the same fashion with unions.

Here is an example:

```
struct {
    int code;
    float cost;
} item;

item.code = 8472;
```

This assigns a value to the `code` member of the structure `item`.

- >** The indirect membership operator is used with a pointer to a structure or union to identify a member of that structure or union. Suppose `ptrstr` is a pointer to a structure and that

member is a member specified by the structure template. Then `ptrstr.member` identifies that member of the pointed-to structure. The indirect membership operator can be used in the same fashion with unions.

Here is an example:

```
struct {  
    int code;  
    float cost;  
} item, *ptrst;  
  
ptrst = &item;  
ptrst->code = 8281;
```

This assigns a value to the `code` member of `item`. The following three expressions are equivalent:

```
ptrst->code    item.code    (*ptrst).code
```

VII. Grouping Operators

- ()** Override precedence; expressions inside parentheses are evaluated first.
- { }** Block delimiters; statements within a brace pair constitute a "block", or compound statement, which is treated as a single statement.

VIII. Miscellaneous Operators

- sizeof** Yields the size, in bytes, of the operand to its right. The operand can be a type-specifier in parentheses, as in `sizeof (float)`, or it can be the name of a particular variable or array, etc., as in `sizeof foot`.
- (type)** Cast operator: converts following value to the type specified by the enclosed keyword(s). For example, `(float) 9` converts the integer 9 to the floating-point number 9.0.

, The comma operator links two expressions into one and guarantees that the left-most expression is evaluated first. A typical use is to include more information in a `for` loop control expression:

```
for ( step = 2, fargo = 0;
      fargo < 1000; step *= 2)
    fargo += step;
```

?: The conditional operator takes three operands, each of which is an expression. They are arranged this way:

expression1 ? expression2 : expression3

The value of the whole expression equals the value of *expression2* if *expression1* is true, and equals the value of *expression3* otherwise.

Here are some examples:

`(5 > 3) ? 1 : 2` has the value 1

`(3 > 5) ? 1 : 2` has the value 2

`(a > b) ? a : b` has the value of the larger of `a` or `b`

Table of Operators

OPERATORS (from high to low priority)	ASSOCIATIVITY
() {} ->	L-R
! ~ ++ -- - (type) * & sizeof (all unary)	R-L
* / %	L-R
+ -	L-R
<< >>	L-R
< <= > >=	L-R
== !=	L-R
&	L-R
^	L-R
	L-R
&&	L-R
	L-R
?:	L-R
= += -= *= /* %=	R-L
,	L-R

C

Binary, Octal, and Hexadecimal Numbers

Binary Numbers

The way we usually write numbers is based on the number 10. Perhaps you were once told that the number like 4652 has a 4 in the thousand's place, a 6 in the hundred's place, a 5 in the ten's place and a 2 in the one's place. This means we can think of 4652 as being

$$4 \times 1000 + 6 \times 100 + 5 \times 10 + 2 \times 1.$$

But 1000 is 10 cubed, 100 is 10 squared, 10 is 10 to the first power, and, by convention, 1 is 10 (or any positive number) to the zero power. So we also can write 4652 as

$$4 \times 10^3 + 6 \times 10^2 + 5 \times 10^1 + 2 \times 10^0.$$

Because our system of writing numbers is based on powers of ten, we say that 4652 is written in *base 10*.

Presumably, we developed this system because we have 10 fingers. A computer bit, in a sense, only has 2 fingers, for it can be set only to 0 or 1, off or on. This makes a *base 2* system natural for a computer. How does it work? It uses powers of 2 instead of powers of ten. For instance, a binary number such as 1101 would mean

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0.$$

In decimal numbers this becomes

$$1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 13.$$

The base 2 (or "binary") system lets one express any number (if you have enough bits) as a combination of 1s and 0s. This is very pleasing to a computer, especially since that is its only option. Let's see how this works for a 1-byte integer.

A byte contains 8 bits. We can think of these 8 bits as being numbered from 7 to 0, left to right. This "bit number" corresponds to an exponent of 2. Imagine the byte as looking like this:

bit number	7	6	5	4	3	2	1	0
value	128	64	32	16	8	4	2	1

Here 128 is 2 to the 7th power, and so on. The largest number this byte can hold is one with all bits set to 1: 11111111. The value of this binary number is

$$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255.$$

The smallest binary number would be 00000000, or a simple 0. A byte can store numbers from 0 to 255 for a total of 256 possible values. Or, if it is a signed byte, it can store the values -128 to 127.

Signed Integers

How does the computer represent a negative number? Perhaps the most obvious way would be to use the left-most bit to represent the sign, with 0 indicating a positive number, and 1 a negative number. This has been done, but the method is inconvenient in practice. For one thing, it produces two zero values: plus zero and minus zero. The Macintosh uses a different system, one called the "two's complement".

To see how the scheme works, let's work with a single byte example. With the two's complement approach, the numbers 0 through 127 represent themselves, while the numbers 128 through 255 represent the negative numbers -128 through -1. Note that in this scheme that a 1 in the left-most bit does indicate a negative number, for the numbers 128 to 255 all have that bit set to 1. So, if we have a signed byte, and if the left-most bit is a one, we subtract 256 from the stored number to get the actual value. Thus, if 255 is stored (all 1s), we subtract 256, getting a value of -1. Going the other direction, if you want to store a value of -30, the computer will subtract 30 from 256 and store 226. In general, the absolute value is subtracted from one plus the maximum unsigned number. Thus, if -30 were to be stored in a short integer, the actual value stored would be 65536 - 30, or 65506.

One consequence of this approach is that the same bit pattern could mean -30 or 65506, depending on whether the computer thinks a location holds a signed or unsigned quantity. To check out the system, assign a negative number to a signed short integer, then print it out using both the %d and the %u modes.

Other Bases

Computer workers often use number systems based on 8 and on 16. Since 8 and 16 are powers of 2, these systems are more closely related to a computer's binary system than is the decimal system.

Octal

"Octal" refers to a base 8 system. In this system, the different places in a number represent powers of 8. We use the digits 0 to 7. For example, the octal number 451 (written 0451 in C) represents

$$4 \times 8^2 + 5 \times 8^1 + 1 \times 8^0 = 297 \text{ (base 10) .}$$

Hexadecimal

"Hexadecimal" (or "hex") refers to a base 16 system. Here we use powers of 16 and the digits 0 to 15. But since we don't have single digits to represent the values 10 to 15, we use the letters A to F for that purpose.

For instance, the hex number A3F (written 0xA3F or 0xa3f in C) represents

$$10 \times 16^2 + 3 \times 16^1 + 15 \times 16^0 = 2623 \text{ (base 10) .}$$

Conversions to and from Binary

Converting from binary to octal or hexadecimal and back is simple because the various bases are all powers of two. Because eight is the third power of two, each octal digit corresponds to three binary digits. Similarly, because sixteen is the fourth power of two, each octal digit corresponds to four binary digits. Let's see how this works.

For octal numbers, the rule is to convert each octal digit to the corresponding three binary numbers. Suppose we have the octal number 06. This is 6 in decimal and 110 in binary. Okay, now consider 066. Each 6 is represented by the same binary pattern, 110, so the binary equivalent is 110110. What about 061? We must remember to represent the octal 1 by a three-digit binary number; that is, we must use 001, and not just 1. Thus the binary equivalent of 061 is 110001.

To go from binary, to octal, just reverse the process. Starting from the right, break up the binary number into groups of three digits and translate each group of three to the corresponding octal digit. Suppose a byte contains the pattern 01011101. Think of the number as looking like this:

001 011 101

We added an extra 0 to the left to make the final group three digits. Well, 001 is just 1 in octal, 011 is 2 + 1, or 3, and 101 is 4 + 1, or 5. This makes the octal equivalent 0135.

With hexadecimal, we use the same general method, except each digit corresponds to a four-digit binary number. For example, 0x6 becomes 0110. This really is the same value as octal 06, but now consider 0x66. This becomes 01100110, which is quite different from octal 066 (00110110), for now one of the extra 0s comes in the middle of the number.

Keep in mind that hexadecimal has the extra digits A,B,C,D,E, and F. Try converting 0xC4. C is 12 in decimal, or 8 + 4, making it 1100 in binary. The 4 is 0100, so 0xC4 becomes 11000100 in binary.

Going from binary to hex, break up the number into groups of four digits. Let's go back to 01011101 and convert it this time to hex instead of octal. First, break it up into groups of four:

1011 0011

The pattern 1011 is 8 + 2 + 1, or 11 in decimal, and B in hex. Similarly, 0011 is 3, so 10110011 becomes 0xB3 in hex.

The following table shows the relationship between decimal, binary, octal, and hexadecimal numbers.

Decimal	Binary	Octal	Hexadecimal
0	00000000	0	0
1	00000001	1	1
2	00000010	2	2
3	00000011	3	3
4	00000100	4	4
5	00000101	5	5
6	00000110	6	6
7	00000111	7	7
8	00001000	10	8
9	00001001	11	9
10	00001010	12	A
11	00001011	13	B
12	00001100	14	C
13	00001101	15	D
14	00001110	16	E
15	00001111	17	F

Table C.1 Conversion Table

D

Bit Fiddling

Some programs need (or, at least, appreciate) an ability to manipulate individual bits in a byte or word. For example, often I/O devices have their options set by a byte in which each bit acts as an on-off flag. C has two facilities to help you manipulate bits. The first is a set of 6 "bitwise" operators that act on each bit of a number individually. The second is the field data form, which gives you access to bits within an int. We will outline these C features here.

Bit Operators

C offers bitwise logical operators and shift operators. In the following, we will write out values in binary notation so you can see the mechanics. In an actual program, you would use integer variables or constants written in the usual forms. For instance, instead of (00011001), you would use 25 or 031 or 0x19. For our examples, we will use 8-bit numbers, with the bits numbered 7 to 0, left to right.

Bitwise Logical Operators

These four operators work on integer-class data, including char. They operate on each bit independently of the bit to the left or right.

~ **One's complement, or bitwise negation.** This unary operator changes each 1 to a 0 and each 0 to a 1. Thus

```
~(10011010) == (01100101).
```

& **Bitwise AND.** This binary operator makes a bit-by-bit comparison between two operands. For each bit position, the resulting bit is 1 only if both corresponding bits in the

operands are 1. (In terms of true-false, the result is true only if each of the two bit operands is true.) Thus

```
(10010011) & (00111101) == (00010001)
```

since only bits 4 and 0 are 1 in both operands.

| Bitwise **OR**. This binary operator makes a bit-by-bit comparison between two operands. For each bit position, the resulting bit is 1 if either of the corresponding bits in the operands are 1. (In terms of true-false, the result is true if one or the other bit operands is true or if both are true.) Thus

```
(10010011) | (00111101) == (10111111)
```

since all bit positions but bit 6 have the value 1 in one or the other operands.

^ Bitwise **EXCLUSIVE OR**. This binary operator makes a bit-by-bit comparison between two operands. For each bit position, the resulting bit is 1 if one or the other (but not both) of the corresponding bits in the operands are 1. (In terms of true-false, the result is true if one or the other bit operands -- and not both-- is true.) Thus

```
(10010011) ^ (00111101) == (10101110)
```

Note that since bit position 0 has the value 1 in both operands, that the resulting 0 bit has value 0.

Usage

These operators often are used to set certain bits while leaving others unchanged. For example, suppose we `#define MASK` to be 2, i.e., binary 00000010, with only bit number 1 being nonzero. Then the statement

```
flags = flags & MASK; /* set all but MASK bits to 0 */
```


would cause all the bits of flags (except bit 1) to be set to 0, since any bit combined with 0 via the & operator yields 0. Bit number 1 will be left unchanged. (If the bit is 1, then 1 & 1 is 1, and if the bit is 0, then 0 & 1 is 0.

Incidentally, the bitwise operators also have an assignment version. That is, the preceding statement could be replaced with this:

```
flags &= MASK;
```

Similarly, either

```
flags = flags | MASK;
```

or

```
/* set MASK bit, leaving others unchanged */  
flags |= MASK;
```

will set bit number 1 to 1 and leave all the other bits unchanged. This follows because any bit combined with 0 via the | operator is itself, and any bit combined with 1 via the | operator is 1.

Suppose you want to turn a particular bit off. We can do this:

```
/* turn MASK bit off, leave others unchanged */  
flags &= ~MASK;
```

Here the negation operator turns all the 0s of MASK to 1, and the 1 to 0. A 1 ANDed with any bit is just the bit, so all those bits are unchanged. A 0 ANDed with any bit is 0, so the bit originally corresponding to the MASK 1 (and ~MASK 0) is set to 0.

Our examples used a mask with a single 1, but they apply equally well to masks with multiple 1s.

Bitwise Shift Operators

These operators shift bits to the left or right. Again, we will write binary numbers explicitly to show the mechanics.

<< Left Shift. This operator shifts the bits of the left operand to the left by the number of places given by the right operand. The vacated positions are filled with 0s and bits moved past the end of the left operand are lost. Thus

`(10001010) << 2 == (00101000)`

where each bit is moved 2 places to the left.

>> Right Shift. This operator shifts the bits of the left operand to the right by the number of places given by the right operand. Bits moved past the right end of the left operand are lost. For unsigned types the places vacated at the left end are replaced by 0s. For signed types, the result is machine dependent. The vacated places may be filled with 0s, or they may be filled with copies of the sign (left-most) bit. The Macintosh uses the second approach. Thus, a negative number remains negative when right-shifted, for there is still a one in the left-most bit.

For an unsigned value, we have

`(10001010) >> 2 == (00100010)`

where each bit is moved 2 places to the right, and 0s are shifted into the two left-most bits.

For a signed value, we have

`(10001010) >> 2 == (11100010)`

where each bit is moved 2 places to the right, and the two left-most bits are filled with the original left-most bit.

Usage

These operators provide swift, efficient multiplication and division by powers of 2:

`number << n` multiplies number by 2 to the nth power.

`number >> n` divides number by 2 to the nth power if number is not negative.

This is analogous to the decimal system procedure of shifting the decimal point to multiply or divide by 10.

When the bits represent pixels, these operators let you shift patterns on the screen from left to right and vice versa. Of course, bits disappear when they reach the end of the integer.

Fields

The second method of manipulating bits is to use a field. A field is just a set of neighboring bits within an int or unsigned int. A field is set up via a structure definition, which labels each field and determines its width. The following definition sets up four 1-bit fields:

```
struct {  
    unsigned autfd : 1;  
    unsigned bldfc : 1;  
    unsigned undln : 1;  
    unsigned itals : 1;  
} prnt;
```

The variable `prnt` now contains 4 1-bit fields. The usual structure membership operator can be used to assign values to individual fields:

```
prnt.itals = 0;  
prnt.undln = 1;
```

Because each field is just 1 bit, 1 and 0 are the only values we can use for assignment.

The variable `prnt` is stored in an int-sized memory cell, but only 4 bits are used in this example.

Fields aren't limited to 1-bit sizes. We can do this:

```
struct {  
    unsigned code1 : 2;  
    unsigned code2 : 2;  
    unsigned code3 : 8;  
} prcode;
```

This creates 2 2-bit fields and 1 8-bit field. We can make assignments such as

```
prcode.code1 = 0;  
prcode.code2 = 3;  
prcode.code3 = 102;
```

Just make sure that the value doesn't exceed the capacity of the field.

What if the total number of bits you declare exceeds the size of an int? Then the next int storage location is used. A single field is not allowed to overlap the boundary between two ints; the compiler automatically shifts an overlapping field definition so that the field is aligned with the int boundary. If this occurs, it leaves an unnamed hole in the first int.

You can "pad" a field structure with unnamed holes by using unnamed field widths. Using an unnamed field width of 0 forces the next field to align with the next integer:

```
struct {  
    field1 : 1;  
           : 2;  
    field2 : 1;  
           : 0;  
    field3 : 1;  
} stuff;
```

Here, there is a 2-bit gap between `stuff.field1` and `stuff.field2`; and `stuff.field3` is stored in the next int.

One important machine dependency is the order in which fields are placed into an int. On some machines the order is left-to-right, and on others it is right-to-left. For this reason, bit fields can cause problems when transporting a program from one machine to another.

E

Macintosh ASCII Table

Dec	Hex	Oct	Binary	ASCII
0	00	00	00000000	NUL
1	01	01	00000001	SOH
2	02	02	00000010	STX
3	03	03	00000011	ETX
4	04	04	00000100	EDT
5	05	05	00000101	ENQ
6	06	06	00000110	ACK
7	07	07	00000111	BEL
8	08	10	00001000	BS
9	09	11	00001001	HT
10	0A	12	00001010	LF
11	0B	13	00001011	VT
12	0C	14	00001100	FF
13	0D	15	00001101	CR
14	0E	16	00001110	SO
15	0F	17	00001111	SI
16	10	20	00010000	DLE
17	11	21	00010001	DC1
18	12	22	00010010	DC2
19	13	23	00010011	DC3
20	14	24	00010100	DC4
21	15	25	00010101	NAK
22	16	26	00010110	SYN
23	17	27	00010111	ETB
24	18	30	00011000	CAN
25	19	31	00011001	EM
26	1A	32	00011010	SUB

Dec	Hex	Oct	Binary	ASCII
27	1B	33	00011011	ESC
28	1C	34	00011100	FS
29	1D	35	00011101	GS
30	1E	36	00011110	RS
31	1F	37	00011111	US
32	20	40	00100000	SP
33	21	41	00100001	!
34	22	42	00100010	"
35	23	43	00100011	#
36	24	44	00100100	\$
37	25	45	00100101	%
38	26	46	00100110	&
39	27	47	00100111	'
40	28	50	00101000	(
41	29	51	00101001)
42	2A	52	00101010	*
43	2B	53	00101011	+
44	2C	54	00101100	,
45	2D	55	00101101	-
46	2E	56	00101110	.
47	2F	57	00101111	/
48	30	60	00110000	0
49	31	61	00110001	1
50	32	62	00110010	2
51	33	63	00110011	3
52	34	64	00110100	4
53	35	65	00110101	5
54	36	66	00110110	6
55	37	67	00110111	7
56	38	70	00111000	8
57	39	71	00111001	9
58	3A	72	00111010	:
59	3B	73	00111011	;
60	3C	74	00111100	<

Dec	Hex	Oct	Binary	ASCII
61	3D	75	00111101	=
62	3E	76	00111110	>
63	3F	77	00111111	?
64	40	100	01000000	@
65	41	101	01000001	A
66	42	102	01000010	B
67	43	103	01000011	C
68	44	104	01000100	D
69	45	105	01000101	E
70	46	106	01000110	F
71	47	107	01000111	G
72	48	110	01001000	H
73	49	111	01001001	I
74	4A	112	01001010	J
75	4B	113	01001011	K
76	4C	114	01001100	L
77	4D	115	01001101	M
78	4E	116	01001110	N
79	4F	117	01001111	O
80	50	120	01010000	P
81	51	121	01010001	Q
82	52	122	01010010	R
83	53	123	01010011	S
84	54	124	01010100	T
85	55	125	01010101	U
86	56	126	01010110	V
87	57	127	01010111	W
88	58	130	01011000	X
89	59	131	01011001	Y
90	5A	132	01011010	Z
91	5B	133	01011011	[
92	5C	134	01011100	\
93	5D	135	01011101]
94	5E	136	01011110	^

Dec	Hex	Oct	Binary	ASCII
95	5F	137	01011111	-
96	60	140	01100000	.
97	61	141	01100001	a
98	62	142	01100010	b
99	63	143	01100011	c
100	64	144	01100100	d
101	65	145	01100101	e
102	66	146	01100110	f
103	67	147	01100111	g
104	68	150	01101000	h
105	69	151	01101001	i
106	6A	152	01101010	j
107	6B	153	01101011	k
108	6C	154	01101100	l
109	6D	155	01101101	m
110	6E	156	01101110	n
111	6F	157	01101111	o
112	70	160	01110000	p
113	71	161	01110001	q
114	72	162	01110010	r
115	73	163	01110011	s
116	74	164	01110100	t
117	75	165	01110101	u
118	76	166	01110110	v
119	77	167	01110111	w
120	78	170	01111000	x
121	79	171	01111001	y
122	7A	172	01111010	z
123	7B	173	01111011	{
124	7C	174	01111100	
125	7D	175	01111101	}
126	7E	176	01111110	~
127	7F	177	01111111	DEL
128	80	200	10000000	Ä

Dec	Hex	Oct	Binary	ASCII
129	81	201	10000001	À
130	82	202	10000010	Ç
131	83	203	10000011	É
132	84	204	10000100	Ñ
133	85	205	10000101	Ö
134	86	206	10000110	Ü
135	87	207	10000111	á
136	88	210	10001000	à
137	89	211	10001001	â
138	8A	212	10001010	ä
139	8B	213	10001011	å
140	8C	214	10001100	ð
141	8D	215	10001101	ç
142	8E	216	10001110	é
143	8F	217	10001111	è
144	90	220	10010000	ê
145	91	221	10010001	ë
146	92	222	10010010	í
147	93	223	10010011	ì
148	94	224	10010100	î
149	95	225	10010101	ï
150	96	226	10010110	ñ
151	97	227	10010111	ó
152	98	230	10011000	ò
153	99	231	10011001	ô
154	9A	232	10011010	ö
155	9B	233	10011011	ø
156	9C	234	10011100	ú
157	9D	235	10011101	û
158	9E	236	10011110	ü
159	9F	237	10011111	ù
160	AO	241	01000000	ˆ
161	A1	241	10100001	°
162	A2	242	10100010	¢

Dec	Hex	Oct	Binary	ASCII
163	A3	243	10100011	£
164	A4	244	10100100	\$
165	A5	245	10100101	•
166	A6	246	10100110	¶
167	A7	247	10100111	β
168	A8	250	10101000	œ
169	A9	251	10101001	ø
170	AA	252	10101010	™
171	AB	253	10101011	'
172	AC	254	10101100	ˆ
173	AD	255	10101101	*
174	AE	256	10101110	Æ
175	AF	257	10101111	Ø
176	B0	260	10110000	ˆ
177	B1	261	10110001	±
178	B2	262	10110010	≤
179	B3	263	10110011	≥
180	B4	264	10110100	¥
181	B5	265	10110101	μ
182	B6	266	10110110	ð
183	B7	267	10110111	Σ
184	B8	270	10111000	Π
185	B9	271	10111001	π
186	BA	272	10111010	∫
187	BB	273	10111011	Ω
188	BC	274	10111100	Ω
189	BD	275	10111101	Ω
190	BE	276	10111110	
191	BF	277	10111111	ø
192	C0	300	11000000	¿
193	C1	301	11000001	¡
194	C2	302	11000010	˘
195	C3	303	11000011	ƒ
196	C4	304	11000100	f

Dec	Hex	Oct	Binary	ASCII
197	C5	305	11000101	•
198	C6	306	11000110	Δ
199	C7	307	11000111	«
200	C8	310	11001000	»
201	C9	311	11001001	...
202	CA	312	11001010	
203	CB	313	11001011	À
204	CC	314	11001100	Ã
205	CD	315	11001101	Ö
206	CE	316	11001110	Ê
207	CF	317	11001111	•
208	D0	320	11010000	-
209	D1	321	11010001	-
210	D2	322	11010010	“
211	D3	323	11010011	”
212	D4	324	11010100	‘
213	D5	325	11010101	’
214	D6	326	11010110	÷
215	D7	327	11010111	♦
216	D8	330	11011000	ù
217	D9	331	11011001	í
218	DA	332	11011010	
219	DB	333	11011011	
220	DC	334	11011100	
221	DD	335	11011101	
222	DE	336	11011110	
223	DF	337	11011111	

Index

! 105
!= 49, 99
#define 59
#include 61
% 30
%= 93
&& 107
& 75, 134
* 29, 136
*= 93
++ 51
+ 28
+= 93
, 116
- 28-29
-= 93
-> 190
/ 29
/= 93
< 99
<= 88, 99
= 27
== 45, 99
> 99
>= 99
? 117
\\0 222
\\n 19
|| 107

address operator 75, 134
AND operator 107
arguments 35, 120-124
array in a structure 200-201
array initialization 173
array 164-177
arrays and functions 170-177
arrays and pointers 167-170
arrays and records 265-266
arrays of arrays 201-209

arrays,
 functions and two-dimensional
 206-210
 initializing two-dimensional 203
 two-dimensional 201-209
ASCII code 53
assembly language 6
assignment operators 93-93
auto 145
automatic variables 144

binary I/O 333-334
binary numbers 53, 355
bit 52
bitwise operators 361-368
bkgpat 281
block 86
branching 85
break 111
buffer 44, 47
button() 105
byte 52

c preprocessor 59-61
case 111
char 24
char 52-57
character arrays 223
character constants 54
character storage 52
character strings 66, 221-258
charwidth() 251, 255
closeport() 277
closern() 316
comma operator 116
comments 20
compiled language 7
compound statement 48, 86
conditional expressions 99-118
conditional operator 117

- continue 110
- control statements 85-118
- copyrgn() 316
- cursor() 288-291
- cursor,
 - hotspot 288
 - mask 288
- data fork 329
- data.h 270
- declaring variables 23-25
- default 112
- defs.h 271
- diffrgn() 316
- disposeptr() 305
- disposergn() 316
- do...while loop 88-89
- double 65
- drawchar() 80, 251
- drawstring() 82, 251, 256
- drawtext() 251, 256
- dynamic memory allocation 273
- end-of-file 57
- entry condition loop 88
- eof 57
- eraserect() 183
- escape sequences 55
- event mask 320-322
- eventrecord structure 322
- events 319-328
- exit condition loop 88
- expressions 33
- extern 147
- external static storage class 149
- external static variables 144
- external storage class 146-148
- external variables 144
- false 100
- fclose() 331
- fflush() 333
- fgetc() 331
- fgets() 233
- fgets() 331
- FILE 330
- files 329-334
- fillpat 281
- fillrect() 281
- float 25, 65
- floating point types 65
- flushing 47
- fopen() 330
- for loop 89-93
- format specifier 70, 76
- fprintf() 333
- fputs() 227, 333
- frameoval() 184
- framerect() 184
- framergn() 315
- fread() 334
- free() 277, 305
- fscanf() 331
- function pointers 215-217
- function types 126-131
- functions 4, 18-22, 35-42, 119-161
- fwrite() 333
- getc() 331
- getchar() 20, 44
- getfontinfo() 251
- getmouse() 289
- getnextevent() 320, 324
- global variables 144
- goto 109
- Grafport 271-291
- handles 306-309
- heap 274, 299
- heap memory 302-309
- heap,
 - nonrelocatable blocks 303
 - relocatable blocks 306-309
- hexadecimal 358
- hidecursor() 19
- Hippo C, using 10-14
- identifiers 25
- if...else 45, 97-99
- if 96
- increment operator 51, 102
- index() 241
- index 164
- indirect membership operator 190
- indirect value operator 136
- initcursor() 289
- initializing variables 50
- insetrect() 186

- int 24, 62
- integer division 30
- integer representation 64
- integer types 62-65
- invertval() 185
- invertrect() 198
- invertrgn() 315
- I/O functions 43-52
- isalnum() 104
- isalpha() 103, 104, 236
- isdigit() 104
- islower() 104
- ispunct() 104
- isspace() 104
- isupper() 104
- jt_theport 273
- keywords 3, 345
- line() 280
- logical operators 105-108
- long 63, 65
- looping 85
- machine language 6
- Macintosh Toolbox 4
- macros 155-159
- main() 18, 119
- malloc() 274, 303
- managers 5
- master pointer 306
- membership operator 179
- memory allocation 274
- memory management 298-309
- menu-selection 114
- moveto() 280
- multiple-file programs 149-151
- nested loops 93-95
- newhandle() 309
- newline character 19
- newptr() 304
- newrgn() 309, 311
- newrgn()313
- NOT operator 105
- NULL 234
- null character 222
- numbers 53

- octal 357
- offsetrect 186
- openport() 277
- openrgn() 316
- operators 27-35, 347
- operator precedence 30
- operators, table of 353
- OR operator 107
- paintrgn() 315
- Pascal TYPE definitions 267
- Pascal procedures and functions 259-263
- Pascal to C 261-277
- Pascal types 264
- passing by value 38
- pattern construction 283-285
- patterns 281-287
- pen parameters 278-280
- pennormal() 280
- pensize() 280
- pintrect() 198
- pnmode 278
- pnpat 278
- pnsiz 278
- pnvis 278
- pointer and array 209-210
- pointer arithmetic 142-144
- pointers as arguments 139-142
- pointers 76, 134-144
- portbits 272
- postfix form 102
- prefix form 102
- printf() 18, 67-74, 226
- ptinrect() 289
- ptinrgn() 315
- putchar() 46
- puts() 228
- random() 278
- rect structure 181
- recursion 151-154
- regions 309-319
- register variable 145
- relational operators 99
- resource fork 329
- return value 39, 125-130

- scanf() 74-79, 231
- scope 144-151
- sectrgn() 316
- setcursor() 289
- setport() 277
- setrect() 184
- setrectrgn() 313, 316
- short 62
- showcursor() 19
- sizeof 274
- .sound 335
- sound driver 335-340
- sprintf() 239
- stack 273, 298, 300-302
- standard input 44
- standard library 4
- statements 33
- static memory 298, 299
- static storage class 148
- static variables 144
- stdin 234
- stdio.h 21, 58, 61
- stdout 227
- strcat() 235, 238
- strcmp() 235-236
- strcpy() 238
- strcpy() 235
- strctop() 82, 250
- string I/O 225-235
- string constants 222
- string format conversion 249-250
- string functions 235-249
- string pointers 224
- string variables 223
- strings,
 - Macintosh Pascal 249-257
 - initialization 223
- stringwidth() 255-256
- strlen() 235
- strncat() 238
- strncpy() 235
- strptoc() 250
- struct 178
- struct rect 181
- structure in a structure 199-200
- structure member, type of 179
- structures 178-191
- structures,
 - arrays of 195-98
 - passing by value 190
 - pointers to 188-190
- stuffhex() 285
- subscript 164
- switch 111-114
- tag 178
- testsize() 255
- textface() 251
- textfont() 251, 277
- textmode() 251
- textsize() 79, 251
- textwidth() 251, 255
- theport 251, 254, 272
- tickcount() 185
- tolower() 236
- Toolbox 297
- top-down programming 17
- toupper() 236
- true 100
- two's complement 356
- type casts 133
- type conversions 131-133
- typedef 210-215
- unionrect() 293
- unionrgn() 315-316
- unions 268-270
- unsigned 63
- value parameter 261
- variable parameter 261
- variables 23
- variant records 268-270
- void 128
- wait() 185
- waitmouseup() 338
- while loop 47
- while loop 86-87
- word 53
- xorrgn() 316

About the Authors

Stephen Prata

Stephen Prata is a professor of physics and astronomy at the College of Marin, in Kentfield, California. He received his B.S. degree from the California Institute of Technology and his Ph.D. from the University of California, Berkeley. His association with computers began with the computer modeling of star clusters. He is currently involved in teaching UNIX and C at the College of Marin. Dr. Prata is author and coauthor of several programming books including *UNIX Primer Plus*, *IBM PC and PCjr Logo Programming Primer*, and *Advanced UNIX—A Programmers Guide*.

The Waite Group

The Waite Group is a San Francisco, California-based producer of high-quality books on personal computing. Acknowledged leader in the industry, The Waite Group has produced and written over forty titles. The Waite Group's books are distributed world-wide, and have been repackaged with the products of such major companies as Epson, Wang, XEROX, Tandy Radio Shack, NCR, and EXXON. Mitchell Waite, Chairman of The Waite Group, has been involved in the computer industry since 1976 when he bought his first Apple I computer from Steven Jobs.

About This Book

This entire book, including the artwork, was typeset and produced using the Apple Macintosh, Microsoft Word, and the Apple LaserWriter. The main body text was set using Times, Times Bold, Times Italic, and Times Italic Bold; the text for computer code was set using Courier; and the artwork was produced using Apple's MacDraw. Screens of the computer output were printed from Apple's MacPaint.

UNLEASHING THE POWER OF YOUR MACINTOSH

The Macintosh C Primer Plus is the first book to present the specifics of the C programming language on the Macintosh...for novice programmers who simply want to learn C, and for advanced users who enjoy developing large-scale Macintosh programs.

This primer covers the essentials of C, the language favored by those who program on the Mac because of C's power and flexibility. Learn all about C and its implementation on the Macintosh including:

- ☐ *C and the Macintosh*: Explaining the basics of Macintosh operations and C functions.
- ☐ *Control Statements*: Telling the computer how to follow orders.
- ☐ *Functions*: Defining and invoking C's modular programming units.
- ☐ *Arrays and Structures*: Classifying and storing data in your program.
- ☐ *Character Strings*: Using stored data for your purpose.
- ☐ All this and more!

The Waite Group is a Sausalito, California-based producer of high-quality books on personal computing. Acknowledged leader in the industry, The Waite Group has produced and written over thirty titles. Their books are distributed worldwide, and have been repackaged with the products of such major companies as Epson, Wang, XEROX, Tandy Radio Shack, NCR, and EXXON. Mitch Waite, Chairman of The Waite Group, has been involved in the computer industry since 1976 when he bought his first Apple I computer from Steven Jobs.



34197-9 * IN U.S. \$18.95 (IN CANADA \$20.95)

N 0-553-34197-9>1895