The Waite Group

PASCAL PRIMER FOR THE MACINTOSH®

A complete guide to Apple's own version of Pascal

- Shows how to program exciting QuickDraw graphics
- Covers the mouse, cursor, and window manipulation
- Demonstrates the new event-driven programming approach
- Teaches through hands-on programming examples

by Dan Shafer

Another book by the bestselling authors of ASSEMBLY LANGUAGE PRIMER FOR THE IBM® PC & XT and BLUEBOOK OF ASSEMBLY ROUTINES FOR THE IBM® PC & XT .... And rave reviews for The Waite Group:

"Anyone who is new to assembly language programming on the IBM PC and feels completely at sea will find a welcome port in Robert Lafore's Assembly Language Primer for the IBM PC & XT."

—John Figueras, reviewing
Assembly Language Primer for the IBM PC & XT, in *Byte*

"[Chris Morgan] has succeeded in producing a volume that no assembly language programmer can do without."

—*Bluebook of Assembly Language Subroutines*,
reviewed in *The Reader's Guide to Microcomputer Books*

"An outstanding example of how to write a technical book for the beginner ... refreshingly enjoyable ... accurate, readable, understandable, and indispensible. Don't stay home without it."

—Ken Barber, reviewing
CP/M Primer, in *Microcomputing*

"Mitch Waite ... has left a distinctive contribution to the literature of computer graphics ... seeing it here is like understanding it for the first time."

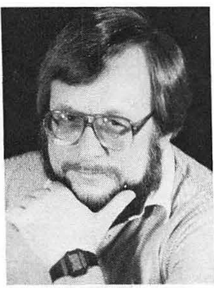
—*Computer Graphics Primer*, reviewed in
Computer Graphics World

"It's hard to imagine that a field only a decade old already has a classic, but Waite's book is just that."

—Tony Dirksen, reviewing
Computer Graphics Primer in *Interface Age*

"... does an excellent job of demystifying the whole study of computer programming in BASIC."

—Annie Fox, reviewing
BASIC Primer in *Creative Computing*



Dan Shafer is an independent product consultant and freelance writer who has been employed for eight years in various phases of high technology in California's Silicon Valley. A former journalist and public relations consultant, he lists writing as his first love, even though he is also keenly interested in programming, chess, stamp collecting, law, and religion. While this is his first published book, dozens of his product reviews and feature articles have appeared in computer magazines during the past five years. Dan lives in Sunnyvale, California, with his wife, Carolyn, and their four daughters: Sheila, Mary, Christine, and Heather.

Dan Shafer

Pascal Primer for the Macintosh®

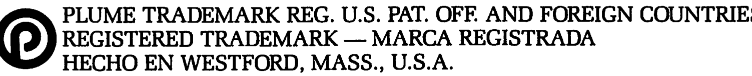
A Plume/Waite Book
New American Library
New York and Scarborough, Ontario

NAL BOOKS ARE AVAILABLE AT QUANTITY DISCOUNTS WHEN USED TO PROMOTE PRODUCTS OR SERVICES. FOR INFORMATION PLEASE WRITE TO PREMIUM MARKETING DIVISION, NEW AMERICAN LIBRARY, 1633 BROADWAY, NEW YORK, NEW YORK 10019.

Copyright©1985 by The Waite Group, Inc. All rights reserved. For information address New American Library.

Several trademarks and/or service marks appear in this book. The companies listed below own or are licensed to use the trademarks and/or service marks following their names.

Apple Computer, Inc.: Macintosh, MacPaint, MacWrite, Lisa
Digital Deli



SIGNET, SIGNET CLASSIC, MENTOR, PLUME, MERIDIAN and NAL BOOKS are published *in the United States* by New American Library, 1633 Broadway, New York, New York 10019, *in Canada* by The New American Library of Canada Limited, 81 Mack Avenue, Scarborough, Ontario M1L1M8

Library of Congress Cataloging in Publication Data

Shafer, Dan.
Pascal primer for the Macintosh.

"A Plume/Waite book."
Includes index.
1. Macintosh (Computer)—Programming. 2. PASCAL (Computer program language) I. Title.
QA76.8.M3S53 1985 001.64'2 85-4829
ISBN 0-452-25659-9

Interior design by Rick Chafian
Illustrations by Winston and Karen Sin
Typography by Walker Graphics

First Printing, June, 1985

2 3 4 5 6 7 8 9

PRINTED IN THE UNITED STATES OF AMERICA

For Carolyn,
who put wings on my weary heart

Contents

Acknowledgments xii

Introduction xiii

1 An Instant Start on Macintosh Pascal 1

Preparing to Run Macintosh Pascal 1
The Mac Pascal Display Screen 3
The Menu Bar 5
The Instant Window: Checking It Out in Advance 8
What's All This Talk about "Programs"? 11
Why Pascal Is Easier to Learn on the Macintosh 11
Summary 13

2 Your First Pascal Program 15

Typing Your First Program 15
The Pause Option 21
Saving the Program on the Disk 21
Running a Program 23
Programming One Step at a Time 24
Making It Different 26
The Programming Process 26
Summary 28
Mac-r-cises 28

3 Anatomy of a Macintosh Pascal Program 29

Some Elements of Program Structure 29
Giving Your Creation a Name 31
Comments: What Those Funny Brackets Are For 32

Begin at the Begin 33
Anything to Declare? 36
Summary 47
Mac-r-cises 47

4 Making Variables Valuable 49

Input and Output 49
The AgeTeller Program: An Example of Macintosh I/O 50
A Variable and Its Value 52
Entering a New Age 54
Reading New Information into a Program 59
Printing in the Macintosh Text Window 64
Summary 67
Mac-r-cises 67

5 Fundamental Control Statements 69

What If I Want to Do Something Different? 69
Looking for the Yes 70
More Complex Use of the If...Then...Else Group 72
Just in Case 78
Running Programs Repeatedly 81
"Next!" 81
But What If We Don't Know? 88
Checking Before or After the Loop Runs 92
Summary 95
Mac-r-cises 96

6 QuickDraw Graphics from Pascal 97

Why Program Graphics? 97
Quick! Draw Me Something! 99
Random Art 104
Getting Shapelier with Our Art 108
Adding Writing to Your Artwork 118
Getting the Picture 120
Summary 123
Mac-r-cises 124

7 Some Fun Things to Do with Letters and Words 125

What's a String Again? 125
Types of String Manipulation Procedures 126
Information-Gathering Procedures 126
Procedures to Compare Strings 129

New Strings from Old 131
One Other Way to Access Part of a String 139
Summary 139
Mac-r-cises 140

8 Can Numbers Be So Much Fun? 141

Types of Numbers in Pascal 141
Displaying Numbers 144
Getting Information about Numbers 145
Creating Numbers at Random 146
Let's Check the Odds 148
Math Quiz Program 150
Other Calculations 151
Summary 154
Mac-r-cises 155

9 Making Your Own Procedures and Functions 156

What Is a Procedure? 156
What Is a Function? 157
Examples of Procedures and Functions 157
When Should You Use Procedures and Functions? 159
A Grapeful Program 159
Procedures to Simplify Program Design 161
"Scoping" with Procedures 166
Circles Have a Point 172
Where Do Functions Come In? 174
Summary 175
Mac-r-cises 176

10 Structured Data Types I: Arrays 177

Will I Need an Array Gun? 177
Using Arrays 181
Array Dismay 183
Tables 185
Putting It All Together 187
Designing Arrays 189
A Word about Subrange Data Types 189
Summary 191
Mac-r-cises 191

11 Structured Data Types II: Sets and Records 193

The Type Section of a Macintosh Pascal Program 193

Types of Types 196
Set 'Em Up! 199
On the Record 206
Summary 213
Mac-r-cises 214

12 Of Mice and Events 215

Where's the Mouse? 216
What's the Mouse Doing in There? 218
The Event Queue 221
Summary 228
Mac-r-cises 228

13 Using the Printer and Disk Files 229

First, a Word about Files 229
Sending Output to the Printer 230
Permanent, Recoverable Storage: Disk Files 233
Summary 237
Mac-r-cises 238

14 Some Advanced Graphics Ideas 239

Cursor Manipulation 239
Window Pains 242
Moving and Resizing Rectangles 245
Summary 247
Mac-r-cises 248

15 Introduction to Dynamic Data Structures 249

Dynamic Data 249
An Example Program 251
Use of Linked Lists 256
Summary 262

16 Other Macintosh Goodies 263

Getting Time in Hand 263
The Mac: A Sound Decision 266
Beep-Beep! 268
Noteworthy Notes 270
Summary 275
Mac-r-cises 275

17 Three Complete Fun and Useful Programs 276

Learning from Others' Programs 276
The Three Programs 277
GridMaker 277
DumpFile 279
ChuckALuck 287
A Friendly P.S. 292

A Solutions to Selected Mac-r-cises 293

Index 309

Acknowledgments

No book is ever totally the work of one person. This book is no exception. The author wishes to acknowledge with great gratitude the contribution of the following to this effort:

Robert Lafore of The Waite Group for a finely balanced editing hand, a finely tuned friendly ear, and great tact and diplomacy in dealing with that most fragile of commodities, a writer's ego.

Don Huntington, a friend who stayed a friend despite the necessity we both saw for him to perform a strong copy edit on the manuscript. His contribution to this book is great indeed.

Kevin Jones of Apple Computer, Inc. and Andrew Singer, Terry Lucas, and Michael Byrne of THINK Technologies, Inc. for technical support at every turn. Mark Wozniak and Chris Lincoln of Computer Plus, Inc., and Gerald Wright of the Digital Deli™ computer store, all of whom read at least portions of the manuscript and provided feedback that was helpful in fine-tuning the book. Chuck Blanchard, a good friend and a first-rate Pascal programmer, for the programs in Chapter 17. His work can also be seen in my *Games and Utilities for the Macintosh* (New York: Plume/Waite, New American Library, 1985). Most of all, I want to thank my family: my wonderful wife, Carolyn, and my patient and loving children, Sheila, Mary, Christine, and Heather. Without them, I would not have had the place, time, or sanity to write this.

Introduction

If you've had your Macintosh™ computer longer than fifteen minutes, you've already discovered how easy it makes drawing, writing, and manipulating the computer itself. Now you're about to see that there is another good reason for buying a Macintosh: Macintosh Pascal.

Who Should Read This Book

Almost anyone with a Macintosh, a copy of Macintosh Pascal, and a desire to learn this exciting language can benefit from this book. You can use it whether you're a beginner with no previous knowledge of programming, a BASIC programmer who is ready to move on to the benefits of Pascal, or an experienced Pascal programmer who is encountering the Macintosh version of the language for the first time.

If you're a newcomer to programming, you can use this book as an introduction not only to Pascal but to programming in general. We'll use short, clear examples to *show* you how to program, rather than deluging you with a lot of complicated theory. If you type in and experiment with the program examples, you'll find yourself understanding the language and writing your own programs before you know it. Exercises at the end of each chapter give you the chance to test your knowledge before you go on to the next topic.

Because Pascal got its start in an academic environment, you may have been afraid that it had to be taught in a theory-oriented "textbook" style. Well, fear no more. We think that Pascal, especially Macintosh Pascal, can be enjoyable and as easy to learn as any computer language yet written. We use a step-by-step "primer" approach to ease you gradually into the language, and we explain and make extensive use of the Macintosh's special features to improve your programs and simplify the

learning process. We deliberately avoid some of Pascal's more complex aspects. Our goal is to introduce you to Macintosh Pascal and make you a competent programmer in the language, not to prepare you for graduate study in computer theory!

If you already know how to program in BASIC, you've probably heard that Pascal offers many advantages over that simpler language. What's more, Macintosh Pascal can be as easy to learn as BASIC was, maybe easier. A special feature of Mac Pascal lets you execute your program one line at a time while an Observe window on your screen shows you exactly what your program is doing at each step. An Instant window gives you the power to try out any Pascal statement any time, so you can experiment to your heart's content.

If you've had previous Pascal experience, this book will open the door to an exciting new world of programming on the Macintosh. Macintosh Pascal is like no previous implementation of the language. This form of Pascal is an interpreted language, rather than a compiled one as previous forms of Pascal have been; that means you can type in a short program and execute it immediately, avoiding the long and frustrating wait normal to compiled languages. Because Mac Pascal makes use of the Macintosh's built-in QuickDraw graphics and toolbox routines, it extends the power of the language far beyond that of traditional Pascal. Separate windows for text, graphics, and program listings, along with the Mac's powerful high-resolution graphics capabilities, sophisticated sound generation, "event-driven" programming, and control of cursor shape and window size, make Macintosh Pascal an exciting adventure for experienced Pascal programmers.

What's So Great About Pascal?

If you have not yet experienced the thrill of watching a Pascal program run through its paces, you are in for a real treat. But there's more to it than that. There are many good reasons for learning Pascal, either as a first programming language or as an addition to an arsenal of computer skills. Some of the more important reasons include Pascal's structured and disciplined approach, its modularity, its increasing popularity among colleges and universities, and its importance in designing and developing computer systems, especially the Macintosh.

Unlike BASIC (and many other "higher-level" programming languages), Pascal requires its programs to have a certain structure. You must follow definite format and syntax rules to become an effective Pascal programmer. This feature of the language forces the programmer to think

through a problem and its solution before writing the program and makes it easier to create a program that other people can figure out, use, modify, and help users run.

Pascal's modularity — its ability to allow a complex program to be broken into smaller parts for easy programming, check-out, and trouble-shooting — is especially handy for people who are employed as professional programmers. They find that teamwork becomes much easier with a programming language that not only *permits* but virtually *requires* that program parts or modules be developed in ways that make them work well together.

Many college students are now learning Pascal as their first computer language. If you are one of those students, this book can help you learn Pascal with more efficiency (and, we hope, more fun) than you might have thought possible.

Perhaps the most important single reason for you, as a Macintosh owner, to learn Pascal is that it is the "language of choice" for the Mac. Almost everything inside the Mac and its larger "sister" computer, the Lisa®, that makes them do what we tell them to do was written originally in Pascal.

Because of that fact, you gain two additional benefits from learning Pascal on the Mac. First, you can get a better understanding of the way the Mac works when you come to understand the language in which its built-in routines were designed and written. (You might become the first "MacGuru" on your block!) Second, virtually all of the Mac's considerable power is available through Pascal programming. You can get more out of your Mac if you learn to speak to it in its "native tongue".

What You Need

To get the most out of this book, you need access to an Apple Macintosh computer (with either 128K or 512K of memory) and an Imagewriter™ printer. You also need a copy of the Macintosh Pascal diskette sold by Apple and the manuals that come with it. If you have other things hooked onto your Mac, such as an external disk drive or telephone communications equipment, that's OK. You won't need them for this book, but they won't get in your way, either.

What You Need to Know

We're going to assume that you have explored your Macintosh fairly thoroughly, probably by using MacWrite™ and MacPaint™. Take a look at the

following list; if any terms or ideas on it sound unfamiliar, we recommend that you review the appropriate part of your Macintosh manuals before you begin your Pascal adventure.

mouse
clicking
icon
dragging
Menu bar
pulling down a menu
making a selection from a menu
windows
moving and changing the size of windows
activating a window
using Scroll bars in windows to see more
Cut
Paste
Copy
saving a document to disk
printing a document
renaming a document
Dialog boxes
scrapbook

What You'll Find in This Book

Chapters 1 and 2 show you how to get Macintosh Pascal up and running and how to write a very simple program. You will find this easy, even if you have never written a program before! In Chapters 3 to 5 we go on to explain some of the fundamentals of Pascal programming, using examples to make the concepts clear.

In Chapter 6 we get into one of the most exciting aspects of Macintosh Pascal programming — making pictures with QuickDraw graphics. Chapters 7 and 8 teach you how to handle words and numbers in Pascal, and Chapter 9 shows you how to write procedures to simplify and organize your programs. Chapters 10 and 11 cover three important ways of handling data: arrays, sets, and records. (Pascal's logical, structured way of handling data is one of the features that set it apart from other languages.)

Chapter 12 describes how to use the mouse in your programs, which leads to a discussion of "events", an important new concept used in the Macintosh. Chapter 13 teaches you about files and how to use them to send data to your printer and disk drive. Chapter 14 covers more exciting graphics concepts, including using and changing the cursor and changing the windows used by the program.

Chapter 15 shows you how to make sound and music using the Mac's sophisticated sound-generation capabilities, and Chapter 16 introduces you to the esoteric field of dynamic data structures. Finally, Chapter 17 provides examples of full-scale Pascal programs. These programs, considerably longer than the example programs used in the text, will let you see how an experienced Pascal programmer constructs an application program.

At the conclusion of most chapters, we've provided some "Mac-r-cises," that is, sample programs and problems for you to solve. It's been said so often that it borders on being a cliche, but it is nonetheless true that the best way to learn a language is to practice it. Experiment with the programs we give you throughout the chapters and then work each chapter's Mac-r-cises to get the most use out of this book. Solutions to selected Mac-r-cises appear at the back of the book.

Ready for the adventure? We'll start by taking a look at Macintosh Pascal and what makes it so easy to use. Sit back, relax, and get ready to enjoy a stimulating learning experience with your Macintosh and its own Pascal language.

1

An Instant Start on Macintosh Pascal

When you finish this chapter, you'll know

- **How to get Macintosh Pascal started**
- **What the Pascal Menu bar contains**
- **How and when to use the Instant window**
- **Why Pascal is easier to learn on the Macintosh**

In this book you won't just *read* about Macintosh Pascal; you'll *see* it and *do* things with it — right away! We'll use the Mac's great graphics, menus, and windows to demonstrate each feature of this powerful language. You'll quickly be able to show your friends or colleagues exciting things you and Mac Pascal can do, such as drawing circles, patterns, and squares with a single statement, creating your own menus, and controlling the mouse.

Let's get started right now by learning how to crank up Pascal on the Macintosh, so we can try something immediately!

Preparing to Run Macintosh Pascal

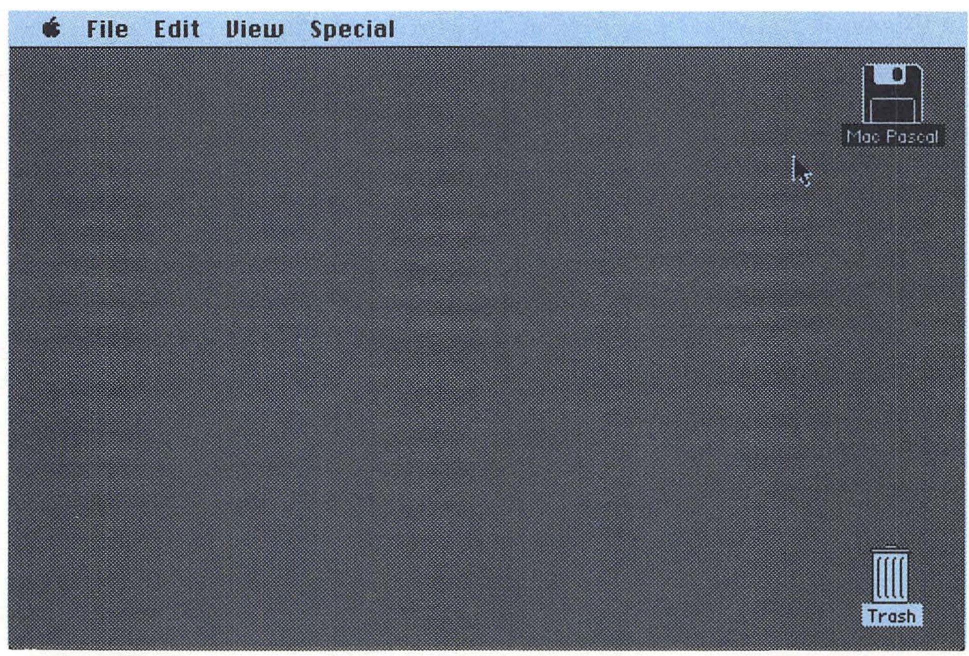
Turn on the Mac and insert the Mac Pascal diskette into the drive. (The metal slide cover on the diskette should be facing away from you, and the labeled side of the diskette should be up.) After the "Welcome to Macintosh" banner appears, you should see a blank screen followed in a few moments by the first Mac screen. This screen displays the Mac Pascal icon in the upper right corner and the trash icon (garbage can) in the lower right, as shown in Figure 1-1.

Open the Mac Pascal disk icon with the mouse. Either double-click on it or drag down the File menu and select the Open option. Your Mac will now display a window showing files, folders, applications, and other icons contained on the Macintosh Pascal diskette. One of these is called "Macintosh Pascal" and looks like this:

Move the pointer over the Pascal icon and double-click on it. After some disk activity the program displays the main Mac Pascal screen, shown in Figure 1-2.

That's all there is to getting Macintosh Pascal up and running!

Figure 1-1. The Mac Pascal and Trash Icons on the Opening Macintosh Screen



The Mac Pascal Display Screen

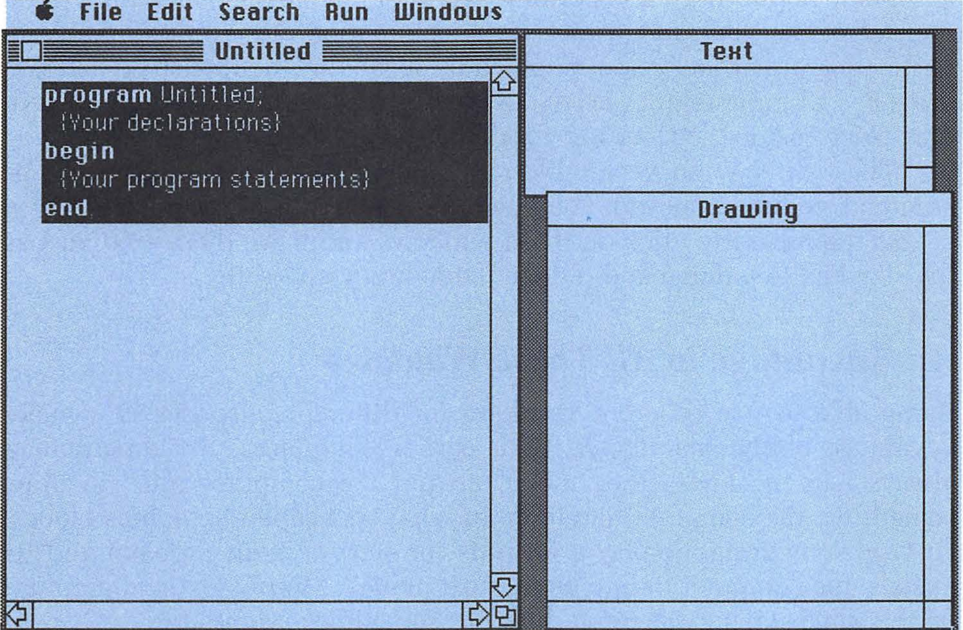
This section provides a quick guided tour of the display screen, which is the screen you saw when you began to run Macintosh Pascal for the first time. The section will explain briefly what each window, menu option, and "hidden" item means and how it can be used.

The Pascal display screen may seem a little bewildering at first. The Menu bar across the top bears some resemblance to those on other Mac programs such as MacWrite, but there are some obvious differences. You will also see three windows on the screen, labeled Untitled, Text and Drawing.

The Untitled (Program) Window

The window that is now labelled Untitled is the window into which you will type your Pascal programs. For this reason, it is called the Program window. Later we'll show you how to change "Untitled" to any name you like, so you can give each of your programs its own name.

Figure 1-2. Main Macintosh Pascal Display Screen



Inside the Untitled window is some text displayed in white letters on a black background. Some of the words are in boldface type: *Program*, *Begin*, and *End*. The meanings of these words in Pascal programming will soon become clear. These three words are part of a much larger list of "reserved words" or "keywords" in Pascal. The designers of the language put these keywords in boldface type to make them easier to see. In the programs printed in this book keywords will appear in bold, just as they do on your Macintosh screen. When we refer to a keyword in text, we'll capitalize it, to emphasize its special function in Pascal.

The Text Window

The upper right corner of the screen display contains a smaller window labeled Text. This window will display the words and numbers to be shown on the screen. If, for example, the program asks the user something, the question will be printed inside this window, as will the user's answer. Similarly, if we give the Pascal program a math problem to solve, the answer will appear in the Text window.

The Drawing Window

The third window on the display is labeled Drawing. If you've played with MacPaint, you know the great graphic power of the Macintosh. You'll soon learn how to put that same power to work for you, using a Pascal program instead of the mouse to direct graphics creation. You'll also be able to write programs that create graphics by means of the mouse. When you use a Pascal program (or part of one) to draw something, the drawing will take place inside the Drawing window.

These three windows are like other Macintosh windows. They can be enlarged, reduced, moved, activated, and erased from the screen. Try it — use the mouse to click on these windows and move them around. Use the size box to enlarge and reduce them. Enjoy yourself!

An Advantage to All These Windows

Being able to use different windows for different purposes is a major advantage of the Macintosh. With earlier computers, the programmer always had to choose the "mode" to put the computer into. To draw something, the computer had to be in what was called "graphics mode". To type something, display a formula, or show a math problem and its answer, the computer had to be in "text mode". Mixing text and graphics on the same screen was often difficult, sometimes impossible.

But Mac Pascal has only one mode. It is always ready for any use. That's one reason why Mac Pascal is such an "instant" language.

The Menu Bar

Look closely at the Menu bar on the top of the main Macintosh Pascal display screen. The Menu bar lists six menus from which you can select various actions for the Mac to take. Let's take a fast guided tour of the menus now; follow along by pulling down each menu as we describe it.

The Apple Menu

This menu is similar to the Apple menu on other applications you've probably run. The top item ("About Macintosh Pascal") provides information about the program's version number, release date, and other important data. Use this information when communicating with your dealer about a problem, question, or update. The other options relate to the system's alarm clock, control panel, and other desktop accessories with which we expect you're already familiar.

The File Menu

This is similar to the File menu in other applications such as MacWrite. There is one new command here: Revert. This command lets you experiment with changes in a program, then discard the changes and *revert* to the former version of the program. Before we finish this adventure, the time will come when we'll be grateful for the Revert option!

When you use the Quit option on this menu, you'll be asked whether you want to save or discard changes to your program before quitting. This is somewhat similar to the MacWrite Quit option.

The Edit Menu

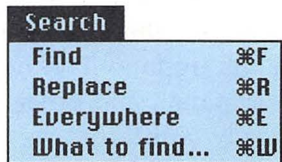
Pull the Edit menu down and find the two command options not seen in MacWrite or MacPaint. Did you find Clear and Select All? The Clear option is used just like the Cut option on this and other Edit menus except that it does not place the deleted copy into the Clipboard. Instead, it simply erases it. You'll want to be careful in using this editing option, since there is no way to recover from Clearing text you really wanted to keep.

The Select All option will act exactly as if you'd positioned your mouse pointer at the start of the program, clicked it, moved to the end of the file, and then used a Shift click combination to select the entire program for an editing action. The effect is that the entire program will be displayed in white letters on a black background (called "reverse video"), just as it is when you select a block of text in MacWrite.

There is no Undo option on this Edit menu. However, the Revert option from the File menu will get you out of trouble if you find you need to Undo something.

The Search Menu

This menu is slightly different from the MacWrite Search menu. Pull it down, and you'll see that it has the following options:



The Find option is used to locate a place in a program to make a change or examine something.

The Replace option makes a single change in a file, substituting one command, word, or other piece of information for another.

The Everywhere option makes that same change everywhere in the file it finds the command, word, or other piece of information that you specify.

The What to Find option tells the computer what words, commands or other information to look for.

Notice that you must tell the system the things to look for and, sometimes, the things to replace those items with, before you can use any of these four options. More about this later.

The Run Menu

This menu appears only with programming language implementations on the Mac. Pull it down and examine its contents briefly.

Run
Check ⌘K
Reset

Go ⌘G
Go-Go
Step ⌘S
Step-Step

Stops In

You can see that there are three parts to the Run menu. The first contains the options Check and Reset. The second contains four choices: Go, Go-Go, Step, and Step-Step. The final section offers one choice: Stops In.

Check instructs Macintosh Pascal to go through a program and ensure that it is *syntactically* correct. Check will make sure that there are no typing errors or obvious structural mistakes in the program. However, be aware that a program that Checks out will not necessarily do what it is supposed to do. Check will not verify the program's logic and design. The only way to check these is by running the program.

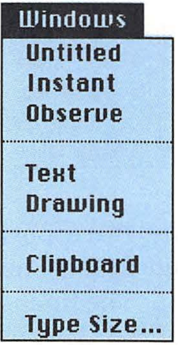
After a program has been run, Reset prepares the program to run again. Among other things, it erases the Text and Drawing windows.

Go, Go-Go, Step, and Step-Step are different ways of causing programs to run — that is, to carry out the instructions given in the program. Most of the time we'll use Go and Step. Go runs a program all the way through, while Step pauses the program at the end of each program line or statement and waits to be told to go on. You'll make extensive use of the Step option in learning Pascal on the Macintosh.

You can use the option labeled Stops In when a program does not work right but you can't quite figure out what's wrong. This option stops the program at certain key points. You can then examine what's happening, determine where the problem lies, fix it, and run the program again. We'll do a lot of run-fix-run-fix-scratch-our-head-run-fix kinds of things with the Stops In option in learning Pascal.

The Windows Menu

The Windows menu is probably another new item for you. Pull it down and look at it. It lists the following windows and window control choices:



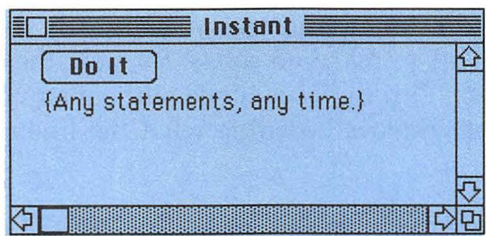
Some of the items in this menu are familiar friends: Untitled, Text, and Drawing are the three main Macintosh Pascal windows you've already seen. The Instant and Observe windows give you special ways of testing and checking out Pascal programs. We will spend time with them later in the book. You've probably used the Clipboard window in MacWrite and MacPaint; it works the same way here.

The last window item is Type Size. Try selecting it now. A Dialog box will ask you to select small, medium, or large type. Click on any button in the box and change the size of the type in which your Macintosh Pascal program is displayed. Smaller type is harder to read but permits viewing more of the program at one time, which is useful for a program having long lines. The larger type is easier to use during program check-out. Pascal begins with medium-sized type.

The Instant Window: Checking It Out in Advance

One of the most useful of all the Macintosh Pascal windows is the Instant window. It provides a peek at what a small part of a Pascal program will do when it's run. The window will be explained in detail a few chapters down the road, but we're going to provide a quick demonstration of this powerful tool right now.

Move the mouse pointer to the Windows menu and pull it down. Now select the Instant option by dragging to it and then releasing. You should then see a new window in the display area that looks like this:



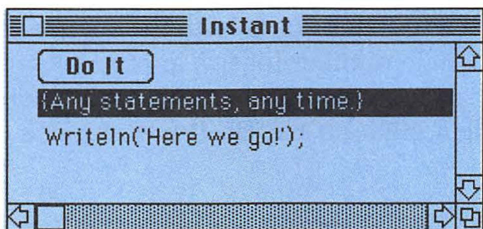
The Instant window will demonstrate any Pascal statement(s) you want to try out. As you'll see, the Pascal language has a set of strict rules, just like English and French. If you had to learn all these rules before you could do anything with Pascal, we'd have to rename this chapter. But you can type program lines into the Instant window without having to fit them into the way Pascal puts programs together, and they will still work! That way, you can begin to work with *pieces* of programs long before you've learned all the structural rules of Pascal. That's something you can do only with the Mac!

Proving the Point

To demonstrate the way the Instant window works, try a one-line mini-program in Pascal. Move the mouse pointer to the line below the one that says "{any statements, any time}" and click. That puts the pointer below this line. (You could also delete this line, but it isn't necessary to do so.) Now, type the following line exactly as it is shown. Don't forget to use the single quotation marks and to place the semicolon at the end of the line.

```
Writeln('Here we go!');
```

Your screen should look like this:



Check to be certain the line is entered exactly as shown above and then click on the Do It button in the Instant window. Watch the Text window; the message "Here we go!" should appear there.

There! You've just written your first line of Pascal code! Run this one-statement command as many times as you like. (By the way, don't worry if the statement seems to make no sense. For now, the important point is that you can type in a line of Pascal code without any great difficulty and can use the Instant window. Learning what the line does and why will come later.)

Now Add Some Lines

This is not a very complex program, so let's add some pizzazz to it. (If you have trouble following these instructions, review the part of the MacWrite manual that covers editing techniques such as inserting, deleting, moving and copying text.) Move the pointer to the line in the Instant window below the line you just typed. Then add these two lines, typing them exactly as shown — including the semicolons at the end of each line.

```
PaintOval (74,72,139,127);
EraseOval (84,74,138,125);
```

Now move to the Run window, drag it down, and select the Reset option. This will clear any leftover "Here we go!" messages from the Text window.

Position the pointer on the Do It button, click on it, and watch the result. Pretty cute, eh? You're already displaying words and pictures on the screen, and you're less than one chapter into your adventure with Mac Pascal. You should be impressed with yourself — and with Macintosh Pascal!

Do Some Experimenting

Let's try one more thing to give you a feel for how you might use the Instant window during actual program development. Move the pointer to the number 138 in the line that starts with the command EraseOval. Click twice to select the whole number (or use another editing technique to do the same thing). Then type the number 122. Pull down the Run window, select Reset, and then click on the Do It button once again and observe the result.

Play around with different combinations of numbers to make various shapes and sizes of ovals until you are comfortable using the Instant window, the Reset option from the Run menu, and the Do It button inside the Instant window. You can come up with some crazy shapes and combinations. Some of them won't work (for instance, erasing the oval at a bigger size than it was drawn will turn the whole window white), but you can't hurt anything. When you're finished, come back to the book and we'll pick up with our next topic. We're almost ready to get into real programming!

What's All This Talk about Programs?

We've discussed programs for a few pages as if you knew what they were. You obviously have *some* idea of what they are, since you bought a book telling you how to create them. But what exactly *is* a computer program?

A computer program is a sequence of instructions that causes a computer to carry out a series of steps designed to achieve a particular result, much as a list of instructions might tell you, for example, how to put a new bike together. Each instruction in the program tells the computer to do one specific thing, which usually leads to the next thing in the sequence. The computer examines a program statement, carries out its instructions, and then moves to the next one.

An infuriating thing about computer programs is that they always do *precisely* what we tell them . . . no more and no less. They are like a rebellious teenager who, when we say, "Pick up that mess in your room," does exactly that — and dumps it on his or her bed. Later, when we tell this person (calmly, of course) that we intended for him or her to put the things away, we are often met with a blank stare and a comment something like, "Oh, is *that* what you meant? I thought you just didn't want the stuff on the *floor* any more!"

The Macintosh, like any computer, is a high-speed idiot. It does very, very quickly *exactly* what we tell it. Once we understand that and come to terms with it, our relationship with the computer becomes much more manageable.

Why Pascal Is Easier to Learn on the Macintosh

You've already learned some reasons why you're fortunate to be learning Pascal on the Macintosh. Let's stop a minute to review some of these innovative feaures.

It's Bold

For openers, Macintosh Pascal *automatically* prints many of the key, or reserved, words (such as Begin and End) in boldface type in the program listings. This is very helpful while you're learning the language. The structure of a Pascal program becomes much clearer because of the obvious appearance of these important words, as you'll see in the next chapter.

Instant Success

The Instant window makes Macintosh Pascal much easier to learn than more traditional forms of Pascal. In previous versions of the language, the only way to find out what a particular statement or combination would do was to place it in a full-blown program, run the program through another program called a *compiler* (more about this later), and then run the program itself. If the oval was too large or in the wrong place or the answer to the math problem was incorrect, you'd have to change the program and then compile and run it all over again, which could be very frustrating and time-consuming.

The Instant window allows you (within certain limits) to try any command or group of commands quickly, see what it does when it's run, and easily revise it. Then, when you have *exactly* what you want, you can use the Mac's editing features to cut or copy it from the Instant window into your program window. This is one of the strongest learning tools we've ever seen for Pascal.

One Step at a Time

Other features of Macintosh Pascal that make learning easy include the Step and Step-Step abilities of the Run menu, which we will soon be using. Briefly, these allow you to walk through the program one line at a time until you find what's causing problems, fix the problem, and then run straight through the corrected program. The Observe window allows you to watch things as they happen in your program. More about this later, too.

Helping Hands

When you're single-stepping through a Pascal program on the Mac, a little hand points to the line that is currently running. This little hand has a brother who's quite useful in letting you know when something's gone wrong.

Let's observe one of these helpful hands in action. Bring up the Instant window and clear it by using the Select All option on the Edit menu and then either hitting the [Backspace] key or selecting the Cut option from the Edit menu. Now type in the following line, exactly as shown:

```
Print ('Hello');
```

Click on the Do It button in the Instant window. You should hear two beeps and see an Alert box that tells you, "The name 'Print' has not been defined yet". A hand should be pointing its thumb down next to the line that is causing the problem.

Obviously, we've told Mac Pascal to do something it doesn't understand in the line that starts with the word Print. So long as the problem is in the Instant window, we could probably find the problem without the help of the little hand. But if a 75-line Pascal program kept doing something unexpected, the hand might become a very helpful friend.

To get out of the mess the example put us in, click in the Alert box, put the Instant window away (by clicking its upper left corner box, just as with any other Macintosh window), and get ready for a really exciting step in the next chapter: your first Pascal program that does something real and interesting.

Summary

We've come a long way in the first chapter of our Macintosh Pascal adventure. Just so you can be impressed with your accomplishments, let's list what you've learned already. You know how to get Mac Pascal up and running, what the purposes of the three main Mac Pascal windows are, and what all six of the Menu bar selections and most of their options are used for. You've used the Instant window to type in your first line of Mac Pascal program code and used the Do It button in that window to cause the line to run or execute. You've gained an idea of what a computer program really is. You've taken a look at some features of Macintosh Pascal that make it very easy to learn and use, including the hands that help us figure out where problems exist in our programs.

Not bad for a first session, eh?

Mac-r-cises

Most chapters after this one will provide you with a few sample problems or programs on which to test your new-found skills in Pascal. This chapter, though, will give you a break and let your head clear a little.

2

Your First Pascal Program

When you finish this chapter, you'll know

- **How to type in a Pascal program**
- **How to run a Pascal program**
- **How to save a program to disk**
- **How to get a printed copy of a program**
- **How to change a program later**

This chapter will take you step by step through the process of creating a Pascal program on the Macintosh. We'll use an intriguing little program that draws some shapes on your screen. You'll have a chance to enter the program into the Mac's memory, run it, save it on the diskette, list it, change it, and load it back into the computer. The chapter will, therefore, leave you with a program to show your friends and also with a good understanding of how programming with Macintosh Pascal works.

The program in this chapter is short, but the *process* of program creation is the same regardless of program size. Rather than explaining the process first and then applying it to this program, we'll first show you the program, letting you see the process in action, and then summarize the steps so you can put them to work in other Pascal programs.

Typing Your First Program

If you follow the steps listed below *exactly* as presented, the program will run the first time you try it. Later we'll talk about ways to modify the program — without knowing a single Pascal statement or programming technique!

1. Be sure your Pascal screen display is set up.

Your screen should look like Figure 1-2 in the last chapter. If necessary, move the windows around and resize them, or turn off the machine and restart.

2. Move pointer to word "Untitled" in programming window.

This is the "Untitled" following the word *program* in boldface type inside the window labeled Untitled. The window name will change automatically later. Choose the word by dragging across it or by positioning the pointer anywhere inside the word and double-clicking. You'll know you're successful when the background changes so that all the text in the Program window is black on white and the word *Untitled* is reversed — that is, white type on a black background.

When you've become more accomplished at Pascal programming, you may choose to simply hit the [Backspace] key once when the main Pascal display screen appears. That will delete all of the beginning framework provided by Macintosh Pascal. But for now, you'll probably find this framework quite helpful.

3. Type the word "First".

Don't type the quotation marks. We're going to call our first program, logically enough, "First". Notice that the word *Untitled* disappears as soon as you type the first letter of the program name. If you've used MacWrite, you won't be surprised by this editing shortcut. When you finish typing the title, *don't press the* [Return] *key!* (If you do so, use [Backspace] to get the semicolon at the end of the line back up where it belongs.)

> ***When typing in Macintosh Pascal, you may use any combination of upper- and lower-case letters. You can, for example, call a program First, first, FIRST, or even fIrSt. Mac Pascal ignores case.***

4. Delete the line that says "Your declarations".

This line doesn't do anything. The curly brackets at the ends of the line tell Pascal that the line is just a comment about the program for people who are reading the code. These extra doo-dads clutter up the program, so let's take out the entire line.

Three ways to delete a line in Pascal are similar to those used in MacWrite: click at the beginning, drag to the end, and then press [Backspace]; click anywhere in the line three times quickly and then press [Backspace]; or click at the beginning, move the pointer to the end of the line without dragging, hold down [Shift] while clicking, then press [Backspace]. Of course, you could also use the Cut option from the Edit menu to delete the selected material, but that's usually slower than using [Backspace] when you're editing a program because you're already using the keyboard.

5. Type the first program lines.

Type the partial word *Var* on the line immediately following the line where you named the program and then press [Return]. No semicolon is needed here because this is just a kind of attention-getter that lets Pascal know that you're about to declare a variable. Don't worry about understanding variables or declaration at this point; we'll get into that soon enough.

Now type the following line between the line you just typed (which now says *Var* in boldface type) and the boldfaced word *begin*. That line should look like this:

```
x:integer;
```

When you type the semicolon, if you look quickly, you'll see a little "jump" in the screen display of this line. Pascal looks at the line and adjusts its position and spacing slightly.

A computer wag has suggested — only partly tongue-in-cheek — that the Pascal language was *not* really named for a famous mathematician. Rather, he says, it is an acronym for *P*lacing *A*ll *S*emi*C*olons is *A* *L*abor.

Semicolons are an important part of the structure of a Pascal program, whether on the Mac or on any other computer. Almost every line of a Pascal program must end with a semicolon. This punctuation mark is Pascal's way of saying to the computer, "Okay, I'm finished with that line now, you can go on to the next one." But you'll notice that we said *almost* every line ends with a semicolon — and therein lies the rub.

Pascal statements are divided into two groups for this purpose: simple and compound. A compound Pascal statement consists of more than one line of Pascal programming code. In some cases, the lines contained within such a group may optionally omit the semicolon; in other cases, semicolons *must* be eliminated. As we introduce various statements and statement groups throughout this book, we'll point out rules about semicolons. Right now, just remember the basic rule: Each statement must end

with a semicolon. In most cases, that means every line ends with a semicolon.

6. Delete the line that says "Your program statements".

Use the same technique as in Step 4 to delete the second comment line.

7. Type in the rest of your program.

By now, you're probably getting the hang of this. Type the rest of the program statements as shown below. All the lines should appear on the screen between the *begin* and *end* statements. Watch for semicolons (and the absence of semicolons!). Don't forget to press [Return] after you enter each line.

```
while x< =190 do
begin
FrameRect(0,0,x,x);
x:=x+5;          ← Don't forget the colon before the equal sign!

end;
```

As you enter these lines, watch the Macintosh display. Notice how it prints some of your words in boldface type and shifts lines as you finish typing them so they line up neatly.

Proofread the program to be sure it looks *exactly* like this:

```
First

program First;

  var
    x : integer;

begin
  while x <= 190 do

    begin
      FrameRect(0, 0, x, x);
      x := x + 5;
    end;

end.
```

If the program you typed doesn't look this way, correct the mistakes with the same editing techniques you use for MacWrite. Once you're sure the program is entered correctly, move to the next step.

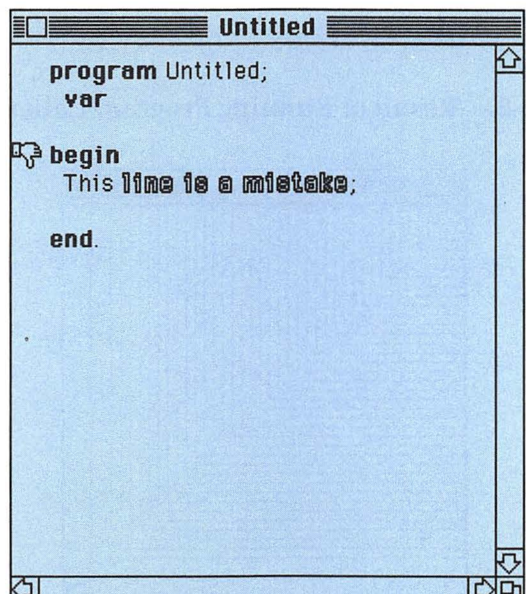
8. Check the program for correctness.

This may sound like what you just did, but it isn't. Move the pointer to the Run menu and select the Check option (the first one on the list). The disk drive will activate (probably), and (if all is well) nothing will appear on the screen. If there *is* a problem, a down-pointing thumb (similar to the one shown in Figure 2-1) and an Alert box will tell you where the mistake is and what went wrong.

The check feature ensures that a program is syntactically correct and will not fail because of a typographical error. (Remember, though, that it does *not* guarantee that the program will do what you want it to do!)

Sometimes the thumbs-down pointer will point to a line *following* that in which the error occurs. For example, if the semicolon is left out after the word *integer* in our program, the hand will point to *begin* instead of *integer*. This is because of the importance of semicolons in Pascal. If the hand points to a line that seems to be all right, check the preceding line and make sure it ends with a semicolon if it's supposed to do so.

Figure 2-1. Thumbs Down Sign Points To Program Error



9. Run the program

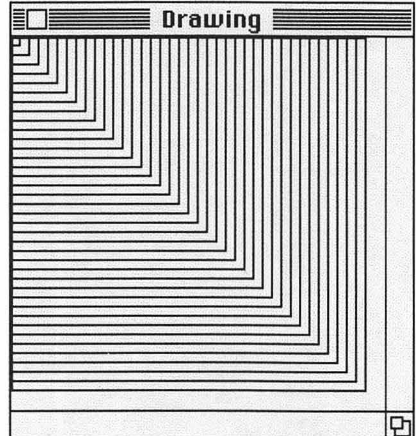
There are two ways to run a Macintosh Pascal program (not counting single-stepping procedures, which we will save for later). You can either pull down the Run menu and select the Go option or hold down [⌘] and the G (for "Go") key together. The latter is a shortcut you'll probably find preferable as time goes on. However, you'll sometimes find it more convenient to use the mouse. Run the program now.

What happened? You should see a series of rectangles nested inside each other in the Drawing window, as shown in Figure 2-2.

Now you know why QuickDraw Graphics from Pascal sometimes work better than MacPaint. You saw how quickly the program put those boxes on the screen. Imagine how difficult it would be — and how long it would take — to draw those by hand, even with MacPaint, as fast as it is!

By the way, a fascinating thing about windows in Macintosh Pascal is that their position on the screen doesn't affect the appearance of program output. For example, move the Drawing window around on the screen. Now Reset the First program and run it again. The window in its new position looks the same as before, doesn't it? (Except, of course, for size, if you changed the size of the window.) This may not seem like a big deal, but on most computers you have to be *very* careful about precisely where on the screen your graphics are displayed or terrible things happen to the program. The Mac is designed so that once you have the program working in the Drawing window, you can move the window without affecting program output.

Figure 2-2. Result of Running Program Called "First"



The Pause Option

When the program is running, a new Menu bar item labeled Pause appears. This option lets you do two things: temporarily pause or permanently halt the program. Neither of these options is useful with the First program because it runs so fast. However, longer programs may be interrupted while running. Positioning the pointer over the Pause menu item and holding down the mouse will cause the program to pause until you release the button. Dragging down on the menu reveals the Halt option, which causes the program to stop running and the Pause option to disappear. We'll use this feature fairly often, especially when we write a program that never stops running (an error that programmers call an "infinite loop").

Saving the Program on the Disk

When writing longer and more complex programs, you'll want to save your work. Always save a copy of your program once it's been Checked and found to be OK. It may still not be correct (remember, the Check option finds only certain kinds of mistakes), but save it anyway, because there's always a possibility it will "blow up" and make a recovery very difficult when you try to run it.

Saving a Program

1. Pull down the File menu and drag to the Save As... option.
2. A Dialog box will appear (see Figure 2-3).
3. Type in the name of the program as you want to save it (in this case, type *First*).
4. Either click on the Save button in the Dialog box or simply press [Return].
5. The program is now saved on the Macintosh Pascal disk.

After you've saved the program on the disk, you may notice something new about the Program box. Instead of being labeled Untitled, it is now called First.

Now that Macintosh Pascal knows the program's name, you can use the Save option on the File menu to save new copies of the program after you make changes, rather than using the Save As... option. The Save option

assumes you want to use the same name over again and doesn't ask you for a name.

Making a Printed Copy of the Program

It's good programming practice to save a "hard copy" of programs. You can display all the lines of the First program on the screen at one time (it's only ten lines long, after all!), but you won't be able to see the longer programs you will be writing in the future. The printed copy will help you see your longer programs at a glance. Also, if a program should ever get "lost" or damaged on the disk, having a printed copy makes it much easier to type it back in.

To print a copy of a program, drag down the File menu, choose the Print... option, then click on the OK button in the Dialog box that shows the format, pages, and so on. The Dialog box shown in Figure 2-4 is the same Dialog box you've seen in MacWrite. The program will print out on the Imagewriter printer. By the way, the best printout results are obtained by using the Standard quality mode for the printer. High quality is quite slow, and Draft quality doesn't supply the indentations and spacing that make a Macintosh Pascal program easier to read and understand.

Figure 2-3. Dialog Box For Save As... Option

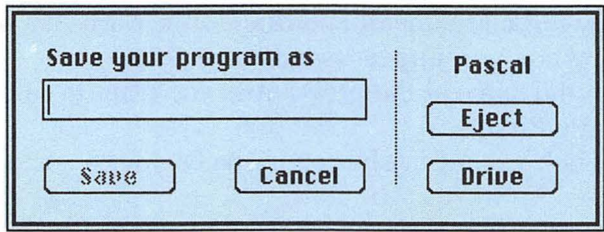


Figure 2-4. Dialog Box For Print Option

Quality:	○ High	◉ Standard	○ Draft	OK
Page Range:	◉ All	○ From:	To:	
Copies:	1			
Paper Feed:	◉ Continuous	○ Cut Sheet		Cancel

Running a Program

You may want to show your spouse, friend, kids, or boss this program. It is, after all, quite an accomplishment that you can already type in and run a Pascal program that creates an interesting visual image on the screen. How do you run a program more than once? That depends on whether it's stored in memory or on the disk.

Running a Program Stored in Memory

To run a program more than once, drag down the Run menu and choose the Reset option to clear the Drawing window after each execution (*execute* is simply a fancy word for *run*). This Reset step is not necessary, but it's nice to start each run of a program with a clean Drawing window and an empty Text window. Once that's done, use the [⌘] G combination or the Run menu Go option to run the program again.

Running a Program Stored on the Disk

Once you turn off the machine, your program is no longer in the Mac's memory. If it's stored on the disk, Open it and move it from the disk into memory. This requires two steps:

First, if there is a program now in the Programming window, eliminate it by pulling down the File menu and clicking on the Close option. You'll be asked if you want to save or discard the changes; click on the Discard button (assuming the program is either already saved or is one you don't want to save), and the Program window will disappear.

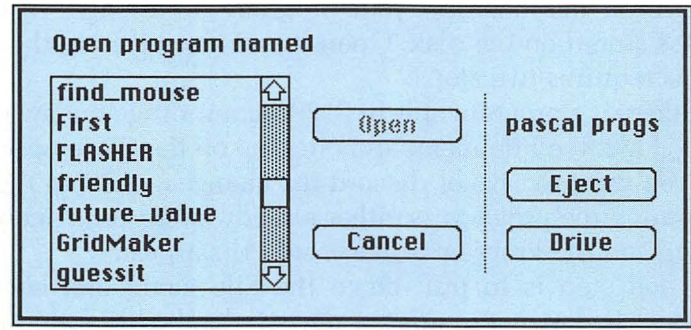
The second step is to pull down the File menu and click on the "Open" option. Following a pause, a new Dialog box like that shown in Figure 2-5a will appear. (Of course, the files shown in the little box inside the Dialog box will be different from those in the figure.) Use the Scroll bar on the right side of the little Catalog box, if necessary, to bring the program to be Opened into the window, then move the pointer to the program's name. You can now Open this program (that is, bring it into the Mac's memory for use) either by double-clicking on its name or by single-clicking. If you choose the latter route, the program's name will be highlighted as shown in Figure 2-5b, and you can then move the pointer to the Open button in the window and click once. The program will load into Mac's memory and will be displayed in the Program window.

Programming One Step at a Time

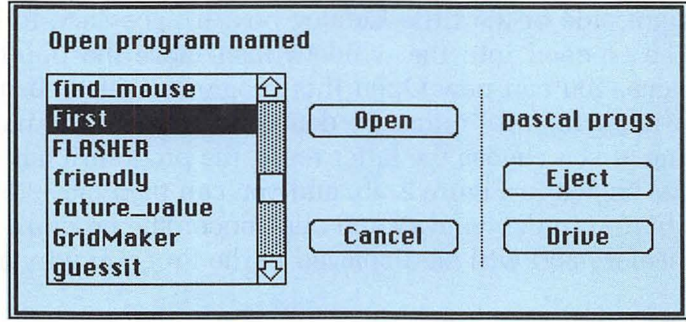
Murphy's Laws apply unerringly to programming. In fact, people in the industry sometimes claim Murphy wrote his laws just for computers. (Murphy would probably deny this, if he weren't too busy scraping dust off his jelly bread.) No matter how experienced you become, you'll still spend a substantial part of your programming time "debugging" your programs.

Incidentally, you might amaze your friends by telling them how the term bug came to mean an error in a computer program. Back in the dark age of computers, say 35 years ago, information moved around in gigantic, room-sized computers by means of relays — electrical switches, like big light switches. When the switches were "on", they closed, and when "off", they opened. One day some technicians had a problem with a program

Figure 2-5. Dialog Box For Open Option



Before selecting file



After selecting file

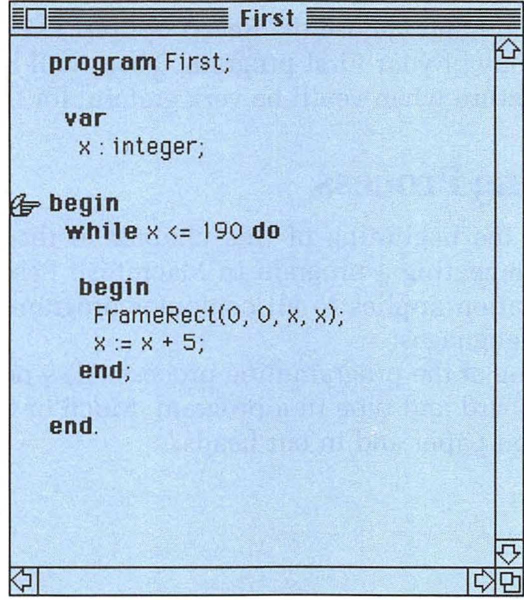
that had run successfully many times. After days of looking for the problem, an engineer discovered that a moth had flown into the circuits and, after choosing to land in the middle of a relay, died. Electricity couldn't flow through the switch because the moth acted as an insulator. Many people swear this story is true. If it isn't, it ought to be!

In any case, one of the most useful tools for debugging programs on the Macintosh is the ability to single-step through a program. You can execute a single line of the program and then have the program stop so you can see exactly what the line did. This is a luxury not available to most Pascal programmers.

Stepping Techniques

Drag down the Run menu and select the Reset option. Now drag it down again and select the Step option. After a pause the program will stop. A little hand will appear at the left edge of the screen, pointing at the line that will be carried out next (see Figure 2-6). The Mac is waiting to be told what to do next. Hold down ⌘ and press S once, then release both keys. The hand will move down one line and the program will wait again.

Figure 2-6. Hand Pointing to Line Being Stepped Through



Do this for a dozen steps. You'll notice rectangles building up in the Drawing window one at a time. Perhaps it will begin to be apparent how the program works. That kind of intuitive insight is one of the great fringe benefits of Stepping through a program. For continuous stepping, just keep [⌘] and S pressed down.

If you get tired of the one-step-at-a-time approach to running a program, just use the [⌘] G combination to restart the Go option. The program will then run on its own.

Making It Different

When you are tired of looking at nested rectangles, change the program by trying one or both of the following:

1. Use the usual editing techniques to change the value of the number 190 in the fifth line of the program. Smaller values will create fewer rectangles and end them closer to the middle or upper left corner of the screen. Larger values will create more rectangles and end them closer to the lower right corner of the screen. A value larger than about 200 will cause the program to run out of room in the Drawing window.
2. Change the value of the number 5 on the eighth line of the program. A number larger than 5 will increase the distance between the rectangles; a smaller number will decrease it. A number less than 1 won't work. You'll understand why after you read the next chapter.

If something goes wrong and you can't figure out what it is, pull down the File menu and choose the Revert option. This will retrieve the original, saved copy of your First program. There will be many times in your Pascal adventure when you'll be very grateful for this feature!

The Programming Process

We promised at the beginning of this chapter to discuss the *process* involved in implementing a program in Macintosh Pascal. Actually, the following information applies to all computer programs on all types of machines in all languages.

The beginning of the programming process takes place before we sit down at the keyboard and type in a program. Much of the programming task takes place on paper and in our heads.

Beginning Thoughts

The first step in writing a computer program is to define the problem. In our First program, the problem was trivial — to find a simple way to draw rectangles and other shapes on the screen.

Identification of the problem leads to the second step, a statement of a solution in general terms. For example, the solution to our problem might have been stated as follows: "Write a program that is easy to type and draws increasingly larger rectangles in the Drawing window." In real programming situations, the solution statement is refined many times before the program is finished.

Step-by-Step Planning

Next, the general statement is translated into a step-by-step plan for achieving the solution. For our sample program, the step-by-step solution might have looked something like this:

1. Draw a very small rectangle, using a counter to permit increase or decrease in size and spacing.
2. Draw a larger rectangle around the first.
3. Continue to draw slightly larger rectangles until the Drawing window is filled.
4. Stop.

Beginning to Program

We have outlined a solution. The next step is to translate it into the programming language we have chosen. Most of this book will be concerned with how to do this step in Macintosh Pascal.

Next, you type the program into the computer and save it on a disk. (As we mentioned earlier, putting programs on a disk before fooling around with them is a very good idea. Things sometimes go wrong, even on the Macintosh!)

The next step in the programming process is to run the program and see what happens. Did it work as expected? Did it run as fast as you wanted? Did unexpected things, good or bad, happen? Do you see ways to make the program faster, more interesting, more useful, or more understandable?

The Program Development Cycle

The next step is not a single step; it is a cycle involving debugging the program, modifying it, saving it, running it, debugging it again, modifying it again, and so on. People who manage professional programmers will tell you this process *never* stops. Don't believe them! I've never completely finished a program myself, but I knew a woman who said she had a program she didn't want to change. (She also told me her cousin had written *three* programs he couldn't think of improvements for, but I suspect she was telling what Huck Finn called "a stretcher" at that point.)

The final step in the process is the most important one: using the program to solve whatever problem you identified in step one.

In a nutshell, that is what programming a computer is all about!

Summary

This chapter completed our introduction to Macintosh Pascal by describing the use of the editing, entry, debugging, Check, File, and Run options needed to get a simple program working. With the example program before us, we are now ready to move on to the real internal workings of a Mac Pascal program.

In the next chapter, we'll take a program apart and find out what makes it tick. You will then understand more about why the First program works as it does.

Mac-r-cises

1. Experiment with the five semicolons in the First program. Try removing them one at a time. Which one(s) can you eliminate without causing an error condition? What do the results tell you about the statements involved, in view of what we said about times when semicolons are required and when they are optional?
2. Move the Drawing window so that it occupies the entire screen display. Now experiment with the upper limit of "x" in the statement that begins with the word *while*. What's the largest value you can put in without making the rectangles disappear off the edge of the window? (Hint: after each run, click on the upper left corner of the Drawing window to make it disappear and then use the Windows option to make it reappear when you need it. You can also simply use the Size box to reduce the Drawing window and then reenlarge it.)

3

Anatomy of a Macintosh Pascal Program

When you finish this chapter, you'll know

- **How to name a Mactintosh Pascal program**
- **What a variable is**
- **How to use the Observe window**
- **What the Begin and End statements are for**
- **How to use four main variable types**

This chapter introduces two crucial Pascal programming concepts: structure and variables. You need to understand these before we can move on to the concepts in Chapter 4. Mastering these concepts will make learning the rest of the language easier. By the way, structure and variables are used very differently in Pascal from the way they are in BASIC. Because of this, nonprogrammers sometimes have an advantage over BASIC programmers when it comes to learning Pascal — they have nothing to unlearn!

Some Elements of Program Structure

Any Pascal program has a defined structure or format that contains a number of elements. Throughout this book, we'll return to the idea of program structure and the way it's handled in Macintosh Pascal. To begin with, we'll focus on the three simplest and most obvious elements of Mac Pascal program structure.

Now, fire up your Mac, get Pascal running, and let's get started! Make sure your screen looks like Figure 3-1. (If the screen doesn't look right, go back to Chapter 1 and review how to get Pascal started.)

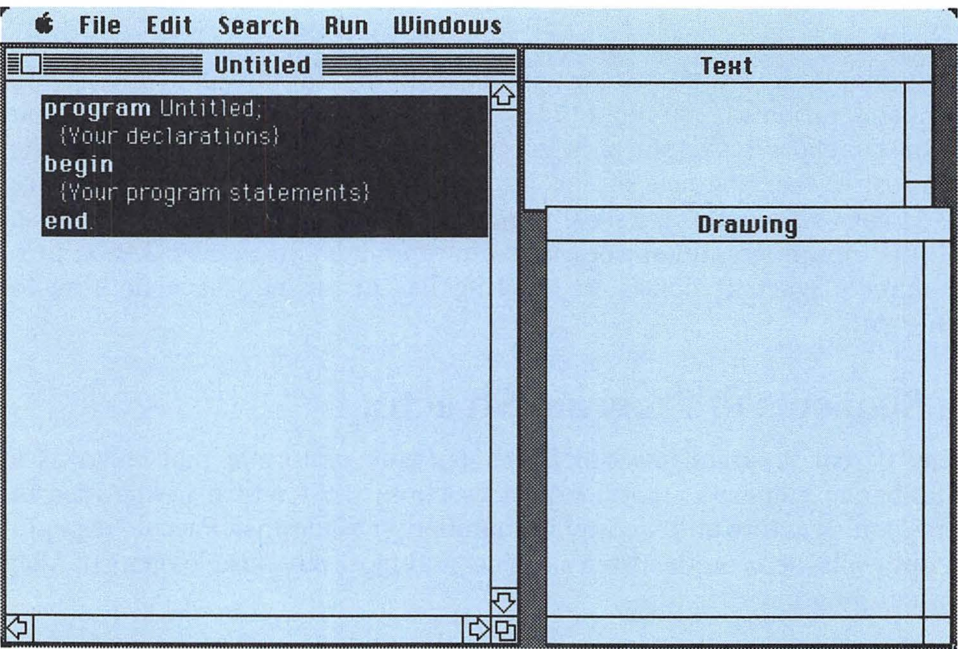
Before we do anything else, take a quick look at your Macintosh Pascal desktop. Notice that three components are already provided in the Program window. One line says *Program* and is now followed by the word *Untitled*; this calls our attention to the first element of Pascal program structure, which involves the necessity of naming things like programs.

The second thing you may notice is a line in the window suggesting that you're going to give the program some *declarations*. These are the second basic components of program structure that we'll look at in this chapter.

Finally, there are two Pascal keywords shown in boldface type: *Begin* and *End*. These two words are separated in the Program window by the line:

```
{Your program statements}
```

Figure 3-1. Opening Macintosh Pascal Screen



This line suggests that between the *Begin* and the *End*, you're going to be typing one or more program *statements*. This pair of statements — Begin and End — is the final major component of Pascal program structure we'll be looking at during this discussion.

Now let's move to the first program structure component: naming things in Macintosh Pascal.

Giving Your Creation a Name

When you create a program in the window labeled Untitled, get rid of the unceremonious name "Untitled" by naming your program. You learned how to do this in Chapter 2, but we didn't talk then about how to *select* names for Macintosh Pascal programs.

What's in a Name?

A program name is actually a Pascal *identifier*. Macintosh Pascal uses identifiers throughout programs to identify things such as constants, types, variables, procedures, functions, programs, and fields within records. (All these terms will become old friends before you've finished this book.)

There are three simple rules for identifiers in Pascal. They apply to program names and to all other identifiers.

Start with a Letter

First, identifiers must begin with a letter. That rules out the use of numbers and all special symbols as the first character in a Macintosh Pascal identifier. Special symbols are characters like @, %, and &, which cannot be called either letters or numbers; there are many such special symbols on the Mac. Numbers may be used in identifiers, but they may not be the first character. Special symbols may not be used at all.

No More than 255 Characters

Second, an identifier may contain up to 255 characters. Macintosh Pascal is far more generous in this regard than most implementations of BASIC and Pascal on home computers, which place severe restrictions on names. In fact, early versions of BASIC for personal computers would permit identifiers in a program to be only *one letter* long! Even today, many versions of Pascal allow identifiers with no more than eight characters. Some versions of BASIC have two-character limits.

So what? Well, it's hard to remember, six months after we wrote a program, that the identifier called N1 meant "first name" and the one called N2 meant "last name". For all we know, they might have meant "nest", "nerd", or even something entirely unrelated to the letter N. But we would have a hard time being confused by a program named FirstName or LastName.

Mac Pascal shows its generosity about names in another way, too. Each character of the 255 is significant or meaningful to Macintosh Pascal. This is a contrast to some other systems, which allow a long string of characters in identifiers but count only, say, the first two as significant. Such systems will read the identifiers Mycat and Mydog as the same thing. Mac Pascal won't. In Macintosh Pascal, even the following two identifiers would be differentiated from one another:

```
An_identifier_naming_a_job_I_did_last_month
An_identifier_naming_a_job_I_did_last_week
```

Macintosh Pascal's design allows us to use meaningful words and terms as identifiers. Take advantage of its generosity and give meaningful names to all the things in your programs that need identifiers. This will permit easy understanding and modification of the programs in later weeks or months.

No Spaces Allowed!

The third rule about Macintosh Pascal names is that each one must look like a single word. Notice that in the two very long names just mentioned, *no spaces* were left between the parts. Instead, underlines connected them. All the words could have been run together, as in Anidentifiernamingajobididlastmonth, but that would be far less readable. It's better to break up long names with underlines, or with capital letters as we do in this book(AnIdentifierNaming . . .). In any case, never use spaces. Mac Pascal will break an identifier with a space in it into two names at the space. Strange things will then happen as Mac tries to do something with the second "name".

Comments: What Those Funny Brackets Are For

The next part of the program skeleton contains the words *Your declarations* between two funny-looking curly brackets. We've seen this combination before, but we haven't taken time to explain it. As it turns out, we can put

anything we want between two curly brackets in a Pascal program and the computer will simply ignore it all. "But, wait!" you may be saying. "What good does it do to put something into a program that the computer is going to ignore?" Good question!

The brackets permit us to include inside the program notes to ourselves and to others who may want to know how the program works. These notes, called "comments", describe what is going on in the program. Get accustomed to using them frequently and liberally in programs you intend to save for later use. Use them especially in longer programs, where they can describe what the various sections do and how they do it.

For reasons that will become clearer in a few moments, we're going to save our discussion of the section of the program hinted at by the phrase "Your declarations" until *after* we've discussed the third major program structure element: the Begin-End pair.

Begin at the Begin

The Begin-End pair is an important structure in Pascal programming. All the parts of a Pascal program that *do* things must appear between a Begin statement and an End statement.

In the program called First, you see two Begin-End pairs. Most Pascal programs contain several Begin-End pairs. You can recognize where each part of a program begins and ends by looking down the program listing and finding the pairs. The Begin-End pair structure is intuitive (that is, it feels "natural" to talk about program sections that *begin* and *end*) and also quite useful. It is one of the features that has made Pascal such a popular language.

What If You Break the Rule?

Let's take a look at what will happen if we forget this Begin-End pair rule in writing a Pascal program. The result will be instructive.

First, go back to a desktop that looks like Figure 3-1. The easy way to do this is simply to Quit Pascal from the File menu and then reselect the Mac Pascal icon from the main Macintosh desktop.

Now press [Return] seven times so the whole Pascal program skeleton moves down in the Untitled window. You're doing this because we're going to cause a Bug box to cover up part of the top of the screen, but we still want to be able to see the program.

Now type the following program, inserting it in the appropriate place in the program skeleton and using all the entry and editing techniques

you learned in Chapter 1. Remember, this generally means deleting the comment lines contained between the curly brackets (although leaving them in place doesn't harm anything).

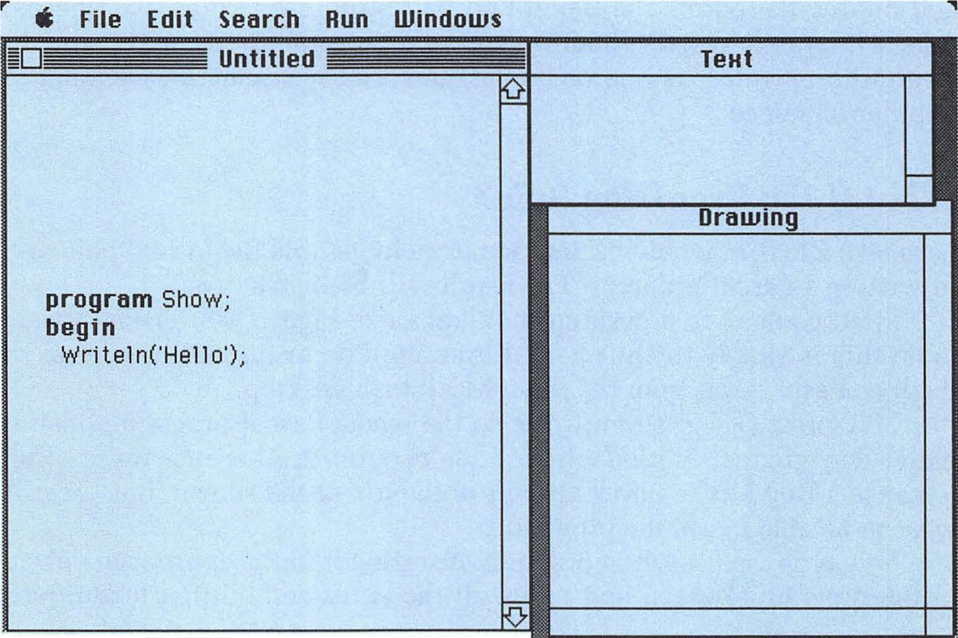
```
program Show;
begin
Writeln('Hello');
```

Be sure the desktop looks like Figure 3-2. Find the Begin. Notice that there is no End in sight.

Okay, are you ready for our first bold, revolutionary attempt at breaking the laws of Pascal? Run that program, using the Go option from the Run menu. (From now on, any time we say "Run the program", we mean to use the Go option from the Run menu.)

If all failed as expected, the Mac beeped and a new box appeared at the top of the desktop with the message, "'END.'is required at the end of the program, but it was not found". Figure 3-3 shows two strong hints that Pascal didn't understand something.

Figure 3-2. Program without End Statement



The Dreaded Bug Box

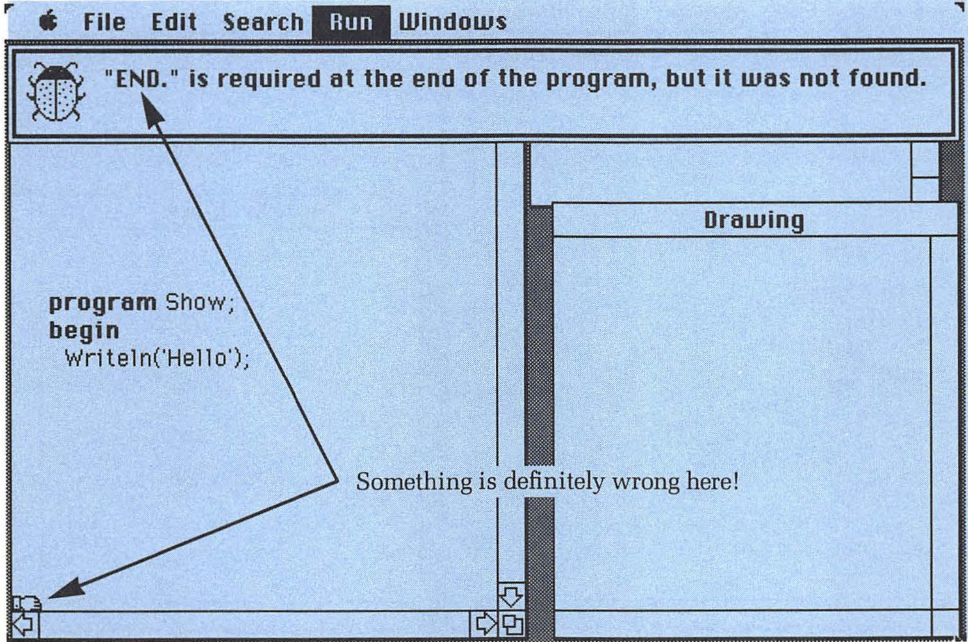
The new box at the top of your desktop is called a Bug box; it even has a little bug in it. Make the Bug box go away (by clicking anywhere inside the box) so we can fix the bug. Find the "thumbs down" sign in the bottom left corner of the untitled window. The sign will stay in place until you begin editing the program.

Go into the Program window and add the word *End.* to the program. (Be sure to include the period, or you'll get another Bug box.) *Now* Run the program, and everything should be fine. You should end up with a screen like Figure 3-4, but without our arrow and comment.

The Rules, Then, Are . . .

Every Pascal program will have a minimum of one Begin-End pair. Most programs will have several or even a great many such pairs, marking the starting and stopping points of various groups of Pascal statements. As

Figure 3-3. What Happens with No End in Sight



we go through our Mac Pascal adventure, we'll point out statement groups that must start with Begin. Anywhere a Begin is needed, an End must also be present.

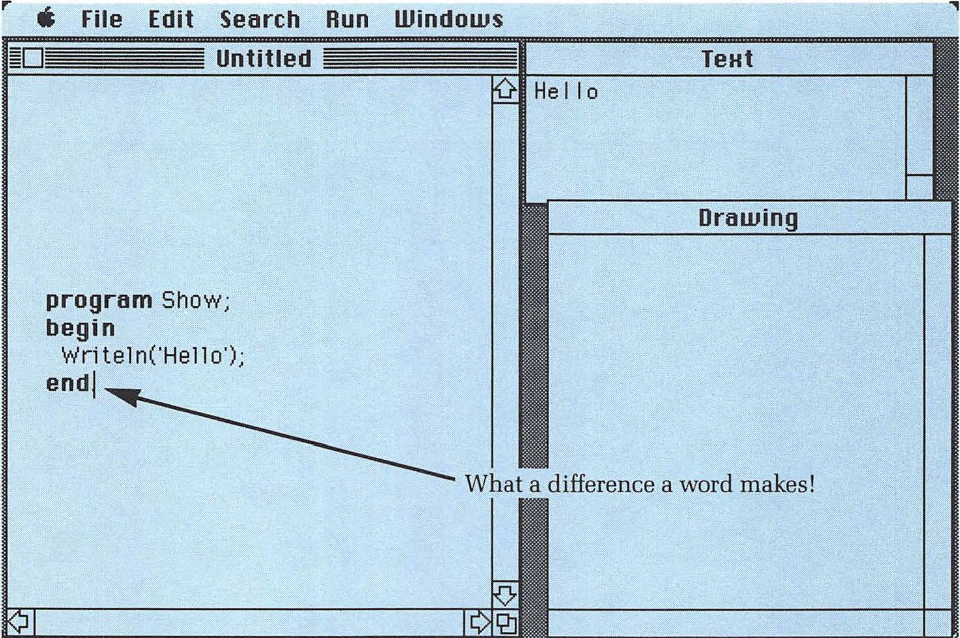
Now we're ready to move on to a discussion of the last major component of Macintosh Pascal program structure: declarations.

Anything to Declare?

Let's return now to the section of the Macintosh Pascal program that begins with the comment line about "declarations". What are we supposed to declare here?

We must "tell" Macintosh Pascal about any names we're going to use during the running of a program *before* we use them. The process of informing Pascal about names before they are used is called "declaration". As we move through our Mac Pascal adventure, we'll see that we can — and, in fact, *must* — declare things of many different types before we can use them.

Figure 3-4. With the End It Works Fine



Variables

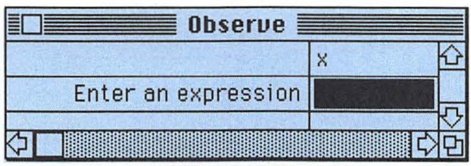
The most common kind of name we'll be defining in our Pascal programs is an item called a variable. We warn Pascal that we're about to declare a variable (or several variables) by typing in the keyword *Var*. In this case, we'd type Var right after the statement that names the program.

Before you can declare any variable, though, you need to understand what a variable is. The idea of variables is simple, though it is often made unnecessarily complicated. In fact, we've all used variables hundreds of times, perhaps without being aware of them.

Let's explore variables briefly before we get involved in an explanation of them. To begin, use the File menu to Close the program we've been looking at. Discard the changes, since you won't be working with this program again. Now Open the program from Chapter 2 called First. (If you didn't save it, or if you've already figured out how to delete files and have deleted this program, simply retype it into the Program window.)

Now Observe Closely . . .

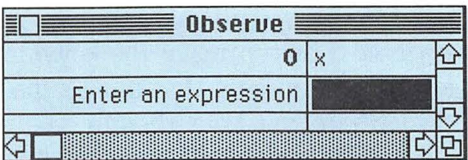
Once the program is shown in the Program window, pull down the Windows menu and select the Observe option. A new window will appear on the display. The window is divided into a small grid pattern, one section of which contains the message "Enter an expression". The pointer is now in the small box to the right of that message. Type a lowercase x and press [Return]. The Observe window should look like this:



Don't worry right now about what an expresssion is or why you entered "x". Just notice the letter x on the line below Var in the program listing (just like the x you just typed in), followed by a colon, the word *integer*, and the inevitable semicolon. This line is the first variable declaration we did together, though we didn't call it that at the time (see Figure 3-5).

Use the Run menu Single Step option or the [⌘] S key combination to single-step through the program. When you begin, the pointing finger

will show up at the line that says Begin, and the Observe window will change so it looks like this:

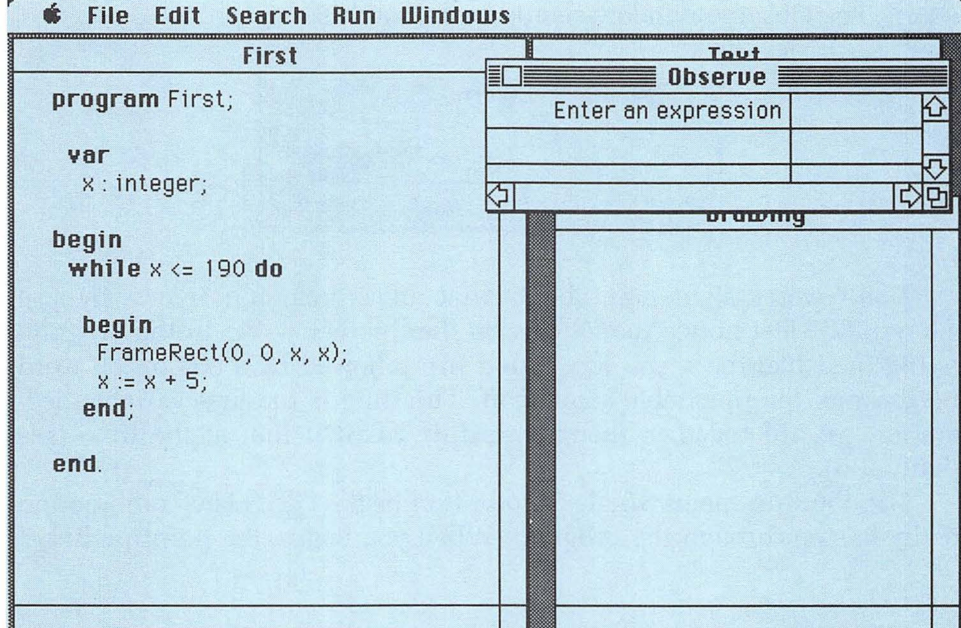


There's now a zero next to the x in the Observe window.

Single-step five more times, and your desktop will look like Figure 3-6. The hand is now pointing to a statement which says "end;", and the number next to the x in the Observe window has changed to 5. Notice that the line just before the End; statement says "x:=x+5". This means that we're adding the value 5 to x, which explains why its number in the Observe window has just gone from 0 to 5. (That ":=" will be explained in Chapter 4.)

Hold down the ⌘ and S keys together for a few seconds and watch what happens. The number next to the x in the Observe window will

Figure 3-5. The Observe Window Is Opened



keep increasing by 5, and rectangles will begin appearing in the Drawing window. Pretty soon the screen will look like Figure 3-7.

Are you beginning to understand variables from the way x is behaving in this program?

> ***A variable is something that changes ("varies", which is where its name comes from).***

Variables in Daily Life

Have you ever filled out a job application form? Every blank on that kind of form involves variables. Take the line "AGE:____yrs.", for example. When you run across that line in 1985, you may write in 23. But, if two years later you have to fill in such a line again, you'd write 25. Your age, then, is a variable. Other variables on a job application include years of experience, last job held, current salary, even your address and telephone number.

It might seem that some things on the form are not variables. Social Security numbers, for example, normally don't change. But people reading

Figure 3-6. Observe Window Shows First Pass through Program

```
program First;

  var
    x : integer;

begin
  while x <= 190 do

    begin
      FrameRect(0, 0, x, x);
      x := x + 5;
    end;

end.
```

your job application will see *all* information on the form as variables. After all, they will probably see a number of applications in the course of a workday, each with different names, addresses, phone numbers, ages, work experience, and so on. From their perspective, Social Security numbers are variables.

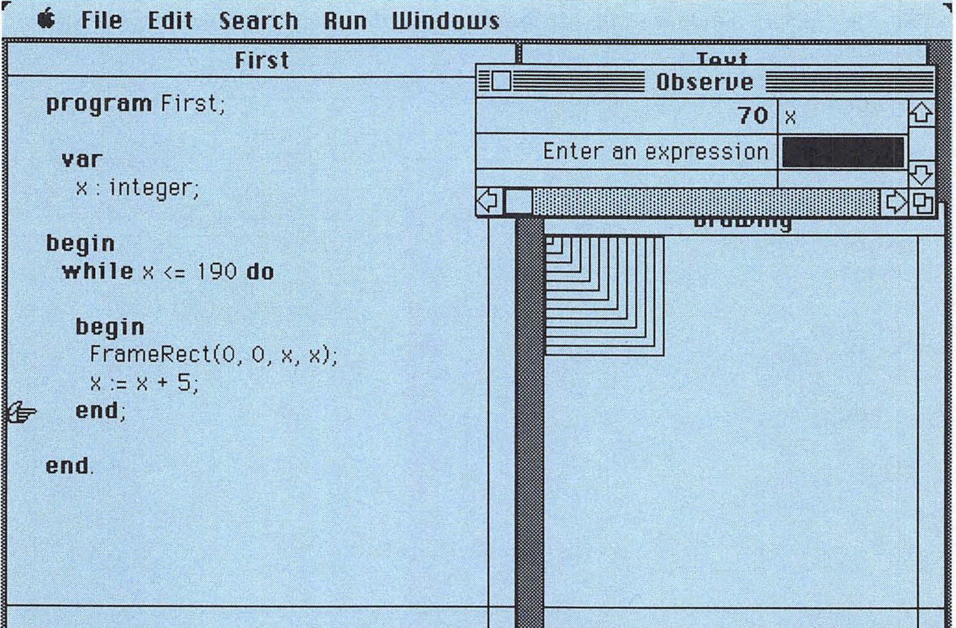
Basically, we can say that a variable is anything that can change. For computers, we can refine the definition by saying that a variable is anything that can change *from the program's viewpoint*.

Variables are one of the things we must tell Pascal about (that is, declare) before we begin a program. If we use a variable we haven't declared, the program containing the variable simply will not run. Some people think this is good for programmers' souls in that it forces a measure of discipline and planning into the programming process that is not always required by other languages.

Sorry, You're Not My Type

Pascal also requires us to decide in advance the kind or *type* of variable we want. We must declare the variable type as well as the variables

Figure 3-7. Observe Window after Several Passes Through Program



themselves near the beginning of the program. Letters and numbers are different variable types. To see why, you need to understand what letters and numbers mean to a computer.

All higher-level computer languages deal with information as either numbers or letters (individual letters or groups of letters in the form of words and phrases). Computers must be able to tell the two kinds of information apart, because they can't process letters in the same way they do numbers. Suppose a computer tried to add *two* (a word) to 5 (a number). Would it get "two5" or 7 or even 25? Adding the two just doesn't make sense, and even your cooperative Mac will refuse to try it.

How Do You Declare a Variable?

Declaring a variable in Macintosh Pascal is easy. Simply determine the type of the variable you want to define (we'll tell you how to do that later in this chapter), give the variable a name, and put the name and type declaration together in the variable declaration section of the program. To see what we mean, look at the First program again.

Notice the line just under the word *Var* in the program. It says:

```
x:integer;
```

This statement has three parts: the variable name (x), a colon, and the variable type (in this case, integer, which means whole number — more on that soon). The variable name is on the left of the colon, the type on the right.

Variables in Bunches

It is possible to use a single statement to define the type of more than one variable. For example, to define three whole numbers, just type:

```
x,y,z:integer;
```

This statement defines three variables, which are separated by commas. It declares them all to be of the integer type. It is the same as typing:

```
x:integer;
y:integer;
z:integer;
```

Combining several variable declarations on a single line makes typing and reading easier. It's good practice to group variables by type.

Meaningful Names . . . A Reminder

Variable names need not be limited to a single character. In fact, they shouldn't be. The names in our example would be much better if they referred to things the variables stood for in the program. For instance, we might write the line like this:

```
Rectangles,Ovals,Lines:integer;
```

An Example

Let's look at another example. Suppose you want to write a program that requires variables to store a name, a pay rate, a number of hours, a one-character code that tells what kind of job the person is doing, and the name of the department in which he or she works. If we tell you that name is a variable with the type String, pay rate is of type Real, number of hours is Real, the code is of type Char, and the department name is of type String, what would the variable declaration statements look like? Try figuring it out on your own before you read on.

Here's how we would handle the variable declarations we just gave:

```
var
    Name,DepartmentName:string[125];
    Payrate,Hours:real;
    Code:char;
```

The order in which variables are defined and the names they are assigned are arbitrary.

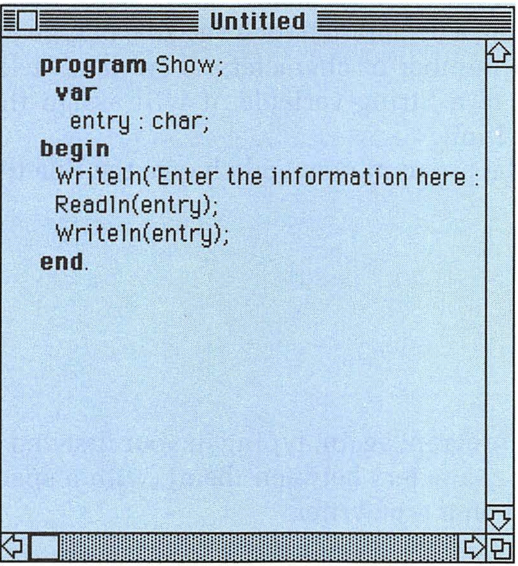
Did you notice the number 125 in brackets after the word String in the first variable declaration statement? A number in this position tells Pascal the maximum number of characters to be assigned to the variable named on that line. The largest string variable allowed in Macintosh Pascal is 255 characters long, just like the largest identifier.

Types of Variables

One of the most powerful features of Macintosh Pascal is the richness of the types of variables we can use when we write programs in it. Check the *Macintosh Pascal Reference Manual* for a description of all the variable types available. Here we'll discuss only the four most useful ones: Char and String for letters, words, phrases, and sentences; and Integer and Real for numbers.

Quite a CHARacter

At the main Macintosh Pascal Desktop (see Figure 3-1) type the following program:



When you've entered the program, use the ⌘ K combination (which is a shortcut way of invoking the Check option from the Run menu) to check it for acceptability to Macintosh Pascal.

Save the program if you wish, then run it. The Text window will display a message, "Enter the information here:" The pointer will then appear on the next line, and the program will wait for you to type something. Type the word *Sam*, and press Return. The screen should now look like Figure 3-8.

Notice that only the first letter of what you typed is displayed by the program in the Text window, even though the whole world appears where you typed it.

Is there a limit to the *kinds* of characters Char will hold? Try typing a single number, like 9. Does that work? What about a special symbol, such as an exclamation point? Or a *really* special symbol like the filled-in circle called a "bullet" (made by holding down the Option key and the 8 key together)?

Longer Groups of Letters and Numbers

The Char type of variable has a limited value. Programs often need information in the form of words, sentences, and phrases. In those situations, you'll want to define a variable or variables as the type called String.

As you've seen, a declaration for a variable of type String may specify a length limit in number of characters. If we don't tell Pascal anything about the length of a String variable, it will assign the maximum 255 characters as the limit.

Go back to the program now and change the line that says:

```
entry:char;
```

to read:

```
entry:string[24];
```

Now run the program again, typing in your first and last name (if they have 24 or fewer characters between them), with a space between them, just as you would on a typewriter.

Figure 3-8. Show Program with One Word as Entry

File Edit Search Run Windows

Untitled

```
program Show;
  var
    entry : char;
begin
  Writeln('Enter the information here: '
  Readln(entry);
  Writeln(entry);
end.
```

Text

```
Enter the information here:
Sam
S
```

Drawing

The screen should look something like Figure 3-9 (although if your name is *really* Blaise Pascal, it would probably be a good idea to use an alias!). The program will accept anything you type, up to 24 characters (since that was the specified limit of the string), and will display it in the Text window.

What if you type something longer than 24 characters? Try it and see.

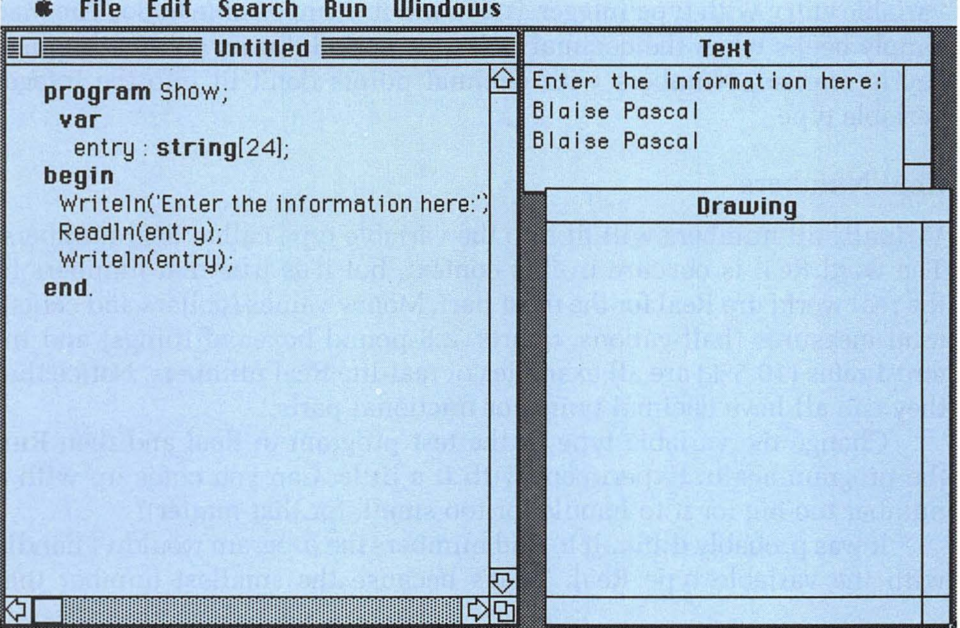
If you tried it, you got a Bug box that said, "String too small for assignment". Now that you know what happens, get rid of the Bug box, Reset the program, and move on to the next topic.

Integers: Numbers of the Simplest Variety

Now let's turn to the world of numbers. This book will be as free of mathematics as possible, but we can't simply ignore the subject. After all, computers are mainly used for "number crunching" — adding up your telephone bill incorrectly, assigning your paycheck to the wrong account, and other similarly useful numerical calculations.

Of the many types of numbers recognized by Macintosh Pascal, the most common is the type known as an Integer. An integer is commonly

Figure 3-9. String Variable Holds Your Name



called a "whole number" — that is, any number, positive or negative, that has no decimal points or fractions. The number 13 is an integer; the number 3.1416 is not.

The Biggest Number Is . . .

Change the program line in our test program from String[24] to Integer. Be sure to delete the brackets and the number 24. Now run the program and try putting numbers in. Try some big ones and some small ones, some positive ones and some negative ones. See if you can answer this question without looking ahead or in the *Reference Manual*: what is the biggest number that can be represented with an Integer variable?

While doing this, you may discover that including commas in numbers causes problems — which brings up an important point about numbers in computer programs:

> ***Commas inside of numbers are nothing but trouble-makers. Leave them out.***

As it turns out, Macintosh Pascal recognizes as an integer any number between −32,767 and 32,767 (without the commas). Any number smaller than −32,767 or larger than 32,767 is represented by another variable type, which we'll describe in the next section.

If you got rebellious and tried to type a number like 3.1416 into the variable entry with type Integer, you found it is impossible to do. The Mac simply beeps when the decimal point (or period, for us non-math types) key is pressed. Numbers with decimal points don't fit into the Integer variable type.

Real Numbers

Virtually all numbers will fit into the variable type called Real numbers. The word Real is obscure in this context, but it is true that numbers in the real world *are* Real for the most part. Money values (dollars and cents), food measures (half-gallons, quarts, 2.5-pound boxes of things) and interest rates (10.5%) are all examples of real-life Real numbers. Notice that they can all have decimal points or fractional parts.

Change the variable type in the test program to Real and then Run the program again. Experiment with it a little. Can you come up with a number too big for it to handle (or too small, for that matter)?

It was probably difficult to find numbers the program wouldn't handle with the variable type Real. That's because the smallest number that

Macintosh Pascal will allow in a variable of this type is the number 15 preceded by a decimal point and *46 zeros*! Now, *that's* a small number! And the largest number Mac Pascal can put into a Real variable is 34 followed by 39 zeros. That number's so big it can represent the national debt — with a lot of zeros left over!

Summary

This chapter has introduced quite a bit of new information. We've examined the structure of a Pascal program and found, among other things, that it has:

1. A specific place to declare variables;
2. A strict rule about Begin-End pairs;
3. Rules about how long names can be and what they can and cannot contain;
4. A need for semicolons at the end of most program lines.

You also learned about including comments in programs, using the principle that the computer ignores anything that appears between two curly brackets.

You learned what variables are, why we have to declare them in Pascal, and how four different types of variables are declared and used in programs.

The next chapter will look more closely at the use of variables in programs. You'll learn how to write programs that do something interesting, and you'll discover how to get information into a program and, equally important, how to get it out and have it displayed in a place where it can be read and used by us humans.

Mac-r-cises

1. Which of the following identifier names would *not* be permitted in Macintosh Pascal and why?

a. ILLEGAL_NAME
b. 2otherguys
c. First Name
d. macintosh-pascal
e. beginningprogram
f. Where_Is_It?

g. xyz
h. solution_step_9
i. one_of_the_very_longest_names_in_the_western_world_of_Mac_Pascal

2. The following program will not run correctly in Macintosh Pascal. Find out why, and revise it so it will be acceptable. (You won't have to understand all the workings of the program to find the error or errors in it.)

```
program BadExample;
x,y:integer;
3name:string;
begin;
  {this program is short but interesting
  Writeln('What is your name?');
  Readln(3name);
  Writeln('Give me a number.');
  Readln(x);
  Writeln('Give me another number.');
  Readln(y);
  Writeln('Those two numbers multiplied together');
  Writeln('will give you ',x*y);
end.
```

3. What type of variable would you declare to hold the results of the following kinds of computer operations?
   a. The total of three golf scores.
   b. This month's electrical bill for late-night Mac work.
   c. A Social Security number.
   d. The total number of times the word *Mac* appears in this book.
   e. The average word length in Blaise Pascal's *Pensees*.
   f. A one-letter response to a Yes/No question in a program.
   g. The value of *pi*.

4. (Extra difficult) Write a program that asks the user to type in the weight of an object in pounds and tells him what its weight is in kilograms. (Use 1 lb. = 0.453592 kg as the conversion value.) This program will require some statements you haven't learned yet but which have been shown in this chapter's sample programs without being explained. (One thing you'll need: to multiply two numbers in Mac Pascal, use an asterisk (*) instead of the x you'd use on a calculator or in working the problem out with pencil and paper.)

4

Making Variables Valuable

When you finish this chapter, you'll know

- **How to assign a value to a variable**
- **How to use Read and Readln statements**
- **How to use Write and Writeln statements**

Two of the most important jobs of any computer program are accepting information (input) and giving out processed results in a usable form (output). A computer program that could not accept or give information would be worse than useless.

This chapter will look at program input and output in Macintosh Pascal. It will explain how the Readln and Writeln statements work. Every program in the book so far has used these terms, even though we have not discussed them.

You'll usually want a computer program to output the value of one or more variables, and the computer *always* accepts input as variables, so we'll also tell you how to assign values to variables you use in your programs. In Chapter 3 we discussed variables at some length but didn't get very specific about their use in a program. We'll remedy that shortly, so that some of these abstract concepts will start to take on meaning.

Input and Output

In a general sense, thousands of sources provide possible input for a computer program. These include filled-out forms, newspaper articles, cash register receipts, thermometers, report cards, and our own memories,

to mention just a few. In a more specific sense, however, only a few sources can make information available to our Macintosh Pascal program:

1. the program itself;
2. the computer keyboard;
3. the computer's storage peripherals (in Mac's case, the disk drives); and
4. other computers that communicate with the Macintosh through a "modem" device, which connects the Mac to a telephone line.

A computer program obtains information (data) from one of these sources. Then it does something with that data — adds numbers, calculates statistics, counts words, checks spelling, or whatever. Finally, it outputs the results in a usable form. Outputting results requires the computer to do two things: translate the results into meaningful arrangements of letters, numbers, and symbols; and put the results on a screen, printer, or some other medium that can be seen (or heard).

Most output from the Macintosh will appear on the system's TV-like screen or monitor. Other means of output include the Imagewriter printer, which most Macintosh owners have; the Macintosh sound capability, which can be made to resemble human speech so that Mac's built-in speaker (or an external speaker hooked up to the Mac) becomes an output device; Macintosh disks; and a modem, which allows the Mac to "talk" with other computers.

The process of putting information into a computer and getting answers out is sufficiently important to rate its own acronym: I/O, meaning (as you may have guessed) "Input/Output". We'll talk about "I/O devices" and "I/O problems". It's a computer buzzword that's worth knowing.

The AgeTeller Program: An Example of Macintosh I/O

The following short computer program demonstrates some of the main I/O facilities of the Macintosh. Before typing it into the Mac, enlarge the Program window so you can see the longer lines as you type them. Then type carefully:

```
program AgeTeller;
 const
  days_in_year = 365.25;
 var
  name:string[80];
  age:integer;
  age_in_days:real;
```

```
begin
  age: = 29;          ← Don't use 29 here; use your real age to the nearest year
age_in_days: = age*days_in_year;
  Writeln('What is your name, please?');        ← Single quotes are important here!
  Readln(name);
  Writeln('Oh, I know you,',name,'.   You are ',age,' years old.');
  Writeln('That means you are about ',age_in_days:,' days old.');
end.
```

After typing this program into the Macintosh, check it for correctness by using [⌘] K or the Check option on the Run menu. If you find trouble, proofread your program very carefully against the listing in the book. Some common mistakes include:

1. leaving out a semicolon where one is needed;
2. using hyphens or dashes instead of underlines in names;
3. inserting space between *writeln* or *readln* and the parenthesis that follows (Pascal is picky about this!)
4. using double quotation marks instead of single ones; and
5. putting a semicolon instead of a period after *end*.

Look for each of these problems and make any necessary corrections. Then run Check again. Once you have run through Check without a problem, save the program on a disk. You'll use this program extensively, and you'll need to be able to retrieve a clean copy after making the changes we suggest.

Running the AgeTeller Program

When your copy of the program is correct, Run it. The result should look something like Figure 4-1. The program will ask for your name. It will then tell you how old you are, first in years and then in days. Don't worry about the funny-looking number with all the digits after the decimal point; we'll make it look right later. Also, don't worry about the colons and numbers in the next to last line of the program: (:12:6). We'll explain what they do later.

Now we have some good news and some bad news. The good news is that you can run this program over and over and always get the same result. That kind of reliability is hard to find these days! The bad news is also that you can run this program over and over and always get the same result. Not everyone in the world is 29 years old (or whatever age you put into the program). The results indicate once again that computers are ignorant — even a great computer like the Mac!

The next section will tell you how to assign a value to a variable, such as age. Once you know about variable assignment, you'll understand why the AgeTeller program in Figure 4-2 doesn't do what you expect—and how to fix it.

A Variable and Its Value

Look at the first line after the Begin statement in our program, the line that says "age:=29" (or whatever value you put there). Age was defined to be a variable of the integer type in the var section of the program a few lines earlier. The number to the right of the variable Age is the number you put into the program in the first line below begin. But what are those weird characters between the word *age* and the number?

Using ":=" to Initialize a Variable

The := symbol is called an *assignment operator*. It assigns to the variable on its left the value that appears on its right. This step is referred to as

Figure 4-1. Sample Run of AgeTeller Program

File Edit Search Run Windows

AgeTeller

```
program AgeTeller;
 const
  days_in_year = 365.25;
 var
  name : string[80];
  age : integer;
  age_in_days : real;
begin
 age := 29;
 age_in_days := age * days_in_year;
 Writeln('What is your name, please?');
 Readln(name);
 Writeln('Oh, I know you, ', name, '.  You are', age, ' years old.');
 Writeln('That means you are about ', age_in_days : 12 : 6, ' days old.');
end.
```

Text

```
What is your name, please?
Stephanie Smythe
Oh, I know you, Stephanie
Smythe.  You are          29
years old.
That means you are about
10592.250000 days old.
```

"initializing a variable", which means giving a variable in a program its *initial* value — the first value assigned to it.

Why initialize a variable? To find out, you'll need to learn a little more about the way variables work.

How the Macintosh Handles Variables

In both Check and Run modes, Macintosh Pascal goes through a program line by line. When it comes to a Var statement, it expects variable names and types to follow. It sets up places in memory where it can "remember" these variables and values so they can be recalled easily and quickly. In the present example, when Age is declared to be a variable of type Integer, Mac Pascal sets aside a small portion of its memory and puts the name "Age" on it. The type of variable, by the way, helps Pascal decide how much memory to set aside.

When the Macintosh is first turned on, its memory locations are filled with arbitrary values (usually with zeros but sometimes with other values). These values may be used to test the computer when it first gets turned on to verify that it is working correctly. While we are using various

Figure 4-2. Program Gets Beethoven's Age Wrong

```
 File  Edit  Search  Run  Windows
                          AgeTeller   Text
program AgeTeller;
  const
    days_in_year = 365.25;
  var
    name : string[80];
    age : integer;
    age_in_days : real;
begin
  age := 29;
  age_in_days := age * days_in_year;
  Writeln('What is your name, please?');
  Readln(name);
  Writeln('Oh, I know you, ', name, '.  You are', age, ' years old.');
  Writeln('That means you are about ', age_in_days : 12 : 6, ' days old.');
end.
```

```
What is your name, please?
Ludwig von Beethoven
Oh, I know you, Ludwig von
Beethoven.   You are
29 years old.
That means you are about
10592.250000 days old.
```

parts of the computer's memory, the computer itself uses other parts for "housekeeping chores" of its own. In other words, unpredictable values are saved in particular memory locations at particular times. A given location could contain a zero or –2545.

The ":=" operator erases this arbitrary value for a given variable and gives the variable its initial value in the program. It causes the computer to go to the memory location set aside for this variable and put the assigned figures (or characters) into memory at that location so they can be retrieved correctly later when they are needed by the program.

There are other ways besides initializing to give a variable a value. It is important, however, that the program assign *some* value to the variable before using or printing it.

Entering a New Age

It should be easy to predict what will happen if you change the number assigned to the variable Age in the AgeTeller program. If you're not sure, or if you would like to try it out for yourself, change the value in the line that initializes the Age variable, and then Run the program to see what happens.

Another Way to Assign Value

Look at the line below the one that assigns age. You'll discover another variable assignment operator (:=) in the midst of a rather long line. As you may remember if you did the last Mac-r-cise in Chapter 3, the * sign in the line means "multiplied by." (Pascal doesn't use the symbol commonly used for multiplication because x is often used as a variable name.) Can you figure out what's going on in this line?

Let's look at the line, one item at a time. The identifier Age_in_days comes first. What can we tell about this identifier? Look at the third line in the Var section of the program and figure this out.

Meet a Constant Helper

You can see that Age_in_days is a variable of type Real. So far, so good.

The variable Age follows the assignment operator in this line. The * sign comes next. Finally there's the identifier Days_in_year, which is defined in another section of the Declarations part of the program — the Const section. Items in this section define *constants* which are the exact

opposite of variables. A constant is an identifier whose value does not change as the program runs; in other words, it stays constant, as you might guess from the name.

You can probably see by now that the line we've been discussing assigns to the variable Age_in_days the value obtained by multiplying the value of the variable Age by the value of the constant Days_in_year. This example shows that a variable may be assigned a value directly, in this case by a number representing age, or indirectly, in this case by having Pascal carry out a calculation and put the result into the variable's memory location.

Another Way

One other important way of assigning values to variables may be illustrated by a change in our AgeTeller program. Add this line after the line that assigns the value of your age to variable Age:

```
My_age:=age;
```

Declare this new variable, My_age, in the Var part of the program. When you've finished, the program should look like this:

```
program AgeTeller;
  const
    days_in_year=365.25;
  var
    name:string[80];
    age,my_age:integer;
    age_in_days:real;
begin
  age:=29;        ← Don't use 29 here; use your real age
  my_age:=age;
  age_in_days:=age*days_in_year;
  Writeln('What is your name, please?');        ← Single quotes are important here !
  Readln (name);
  Writeln('Oh, I know you,',name,'.   You are ',age,' years old.');
  Writeln('That means you are about ',age_in_days,' days old.');
end.
```

Now Run the program. It looks the same, doesn't it? The output is identical, even though you've added a whole new variable and variable assignment line to the program. At this point, the program does not tell

Mac Pascal to *do* anything with the new variable My_age. The change is important, however, because the program now assigns the value Age to the variable My_age, even though the word *age* isn't itself an integer. This illustrates a major point: Macintosh Pascal may assign a variable a value that is represented by a second variable. Please understand that Mac Pascal doesn't assign the second variable's *name* to the first variable; rather, it assigns the variable's *value*.

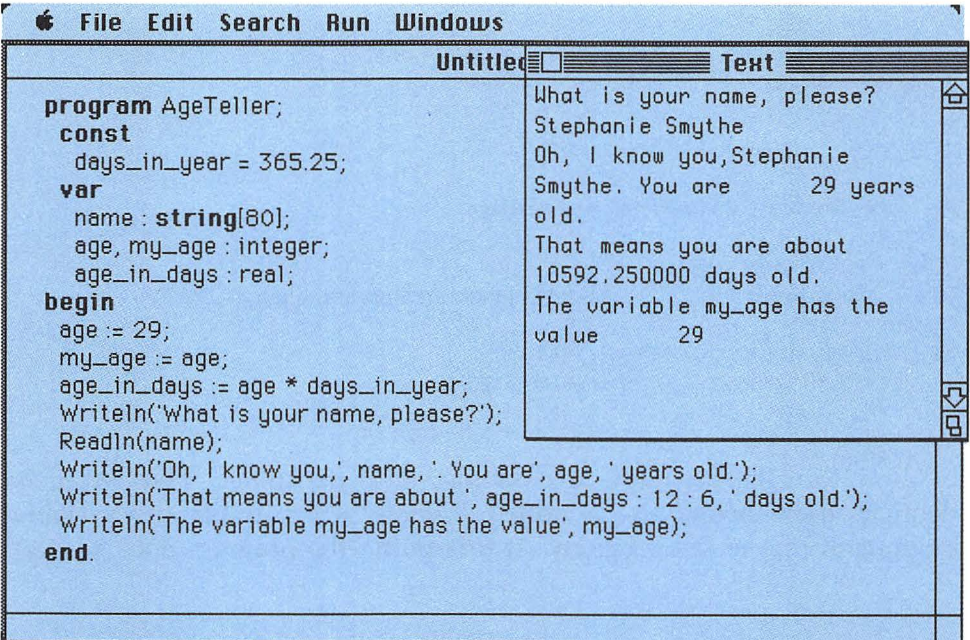
To show that this really happens, add the following line to the program just before the End statement:

```
Writeln('The variable my_age has the value ', my_age);
```

(If you save this revision to the program, save it under a different name from the original program. The original, unchanged program will be necessary later.)

Run the program. The results, shown in Figure 4-3, should be fairly predictable; the last line of the program confirms the point we are making, that Mac Pascal can assign a variable a value represented by a second variable.

Figure 4-3. Assigning a Variable's Value to Another Variable



You're Not My Type, Revisited

There is one more thing to learn about initializing variables and using them in programs before we can move on to the "real" I/O stuff. To get started on this, make one more change in the AgeTeller program. Change the line that initializes the variable my_age to the value of the variable Age so that it reads as follows:

```
my_age: = name;
```

Now Run the program again. The results should look like Figure 4-4. The message in the Bug box reveals a new truth about variables. (Bug boxes, though often aggravating, do reveal new truths at times.) The types of variables assigned to one another as values must be "compatible", which usually means "identical". Here we have tried to assign the value of the variable Name, which is a String variable 80 characters long, to a variable we've told Mac Pascal will be an Integer variable. The Macintosh

Figure 4-4. Bug Box for Incompatible Variable Types

```
File  Edit  Search  Run  Windows

An incompatibility between types has been found.

   days_in_year = 365.25;
  var
   name : string[80];
   age, my_age : integer;
   age_in_days : real;
 begin
  my_age := name;
  age := 29;
  age_in_days := age * days_in_year;
  Writeln('What is your name, please?');
  Readln(name);
  Writeln('Oh, I know you,', name, '. You are', age, ' years old.');
  Writeln('That means you are about ', age_in_days : 12 : 6, ' days old.');
  Writeln('The variable my_age has the value', my_age);
 end.
```

doesn't know how to do that, and the result wouldn't have any meaning anyway. It would be like saying to someone, "I'm Dustin Hoffman years old." So Mac Pascal notifies us that we've made an error.

Type Compatibility and Numbers

Computers use letters and numbers so differently that the *number* 29 is *not* the same as the *characters* 2 and 9 next to each other in a program. Mac Pascal can store the *number* 29 in a variable with a type like Integer or Real, but the *characters* 2 and 9 are merely separate characters. A program can no more do something mathematical with them than it can add L and 3.

As a result of this peculiarity, programs can put numbers into string and character variables, but they *cannot* put letters into numeric variables (that is, integer or real). People in the early computer days created a word to reflect this situation: alphanumeric. This word designates information consisting of letters, numbers, and symbols, where the numbers are not usable as mathematical values but only as characters.

An Overview of Variable Assignment

Let's review what you've learned about assigning values to variables, since we'll be doing a great deal of this as we continue our Pascal adventure. In Macintosh Pascal, all of the following things are true:

Truths About Variables and Their Assignment

1. We must initialize a variable (give it a value) before using it.
2. We can assign to a variable a value that is a number, a string, or a character, depending upon the variable's type.
3. We can assign to a variable a value represented by another variable of a compatible type.
4. We can assign to a variable a value calculated by the program using numbers — either constants or variables.

Reading New Information into a Program

Let's look at a second way of initializing variables in Macintosh Pascal — and a way of getting information into a program from the "outside world". In this section you'll learn to use the Read and Readln statements to enter data from the Mac's keyboard.

Programs would be both boring and useless if the only way to assign a value to a variable was to do it within the program, as we did in the previous section. The Read and Readln statements provide a way of greatly enriching programs by allowing input from oustide; as a result, they are among the most often used statements in Pascal programming.

Find the line with the Readln statement in the age-telling program we've been working with. It looks like this:

```
Readln(name);
```

This statement instructs Pascal to get information from the keyboard and store it in the place reserved for the variable Name. Running the program and typing "Stephanie Smythe" in response to the Readln statement is roughly equivalent to having a program statement that looks like this:

```
name: = "Stephanie Smythe";
```

Including the name assignment in the program permits the name to be changed only by changing the program itself. With the Readln statement, however, we can change the name by simply typing a different response next time we run the program.

A Closer Look at Readln

Let's try something with Readln to get a better understanding of the way it works. Change the Readln statement in the age-telling program so that the typed answer will go into the variable Age instead of the variable Name. (You're right, that doesn't make sense. But programmers often do senseless things inadvertently, so it's nice to be prepared for them.)

If you change the line as suggested, the new Readln statement will look like this:

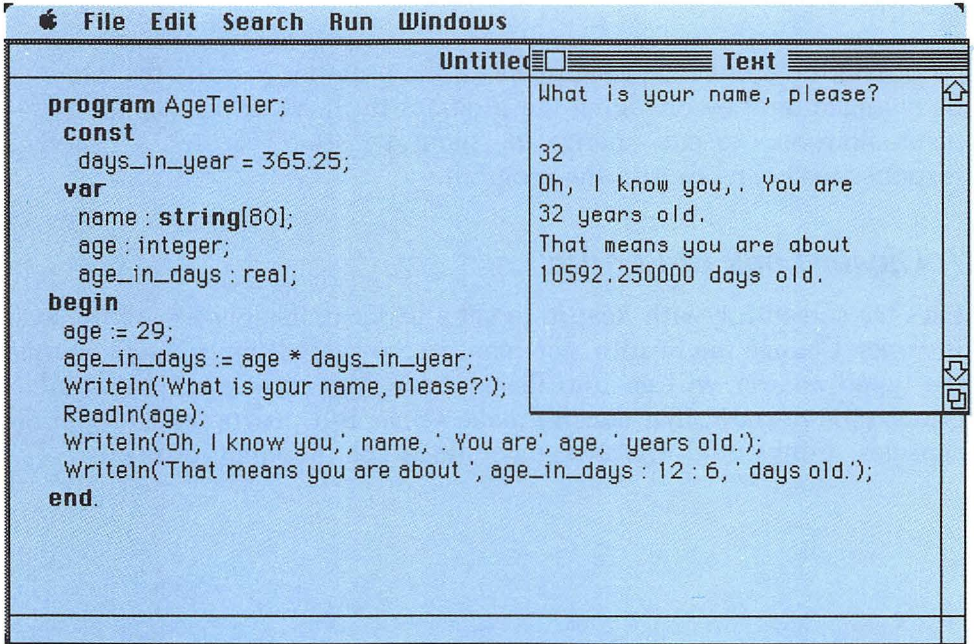
```
Readln(age);
```

The program will still ask for a name, since we intentionally didn't tell you to change the line that asks the question. Try typing in your name and see what happens. Can't do it, can you? The Mac simply beeps each time you try to type a letter. Can you guess why? Hint: Try typing in a number instead of a letter and see if the Macintosh likes that any better.

As we'll bet you've guessed, the result of this foolishness is the same as the result of trying to assign a string value to a numeric variable. The Macintosh won't let us get away with such imprecision.

You may also have noticed that when you typed in a number, the program responded in two ways you've never seen before (see Figure 4-5). First, it printed a line that says, "Oh, I know you, ." — printing a blank space where it used to display your name. This is because we've now left the variable Name uninitialized, so the Macintosh doesn't have a value for it in its memory.

Figure 4-5. Variable Assignment in Readln Statement



The Last Shall Be First

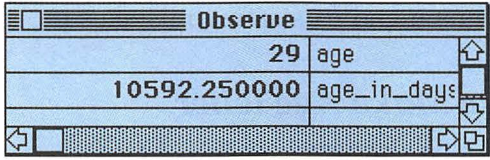
The second new thing you saw is that the program told you that you had an age that is *not* 29 (unless, of course, you typed *29* in response to the question). A look at the program reveals that the line "age: = 29" is still there; so why does the program now think your age is what you've typed in here instead of 29? It does so because you changed the value of the variable after initializing it to the value 29. Macintosh Pascal will *always* use the *last* value assigned to a variable when it does something with that variable. Want some proof? Just compare the value for your age in days now that you've used some value other than 29 with that in Figure 4-2, which shows what happened the first time we ran the program successfully.

You can see that Mac Pascal has used the value 29 in calculating your age in days, even though the printing of that statement occurs *after* you've told the program you are some other age entirely. Why did that happen? Let's use the Observe window to help figure this out.

Observing the AgeTeller Program

Pull down the Windows menu and click on the Observe option. When the Observe window appears, move it down below the Text window so you can see what's going on there while the program is running. It's also a good idea (though not necessary) to make the Observe window slightly larger horizontally. Now type in the variable Age in the first right-hand box. Press Return and type the variable name Age_in_days in the next right-hand box. Now let's single-step through the program and watch the Observe window for some clues as to what's going on.

With the little hand pointing to the first Writeln statement, the Observe window should look like this:



Now Run the program, rather than single-stepping. (When you single-step through a program, the system won't let you answer Readln statements.) It should be easy to see what happened. Just to be sure you

understand it, though, let's go through it in the order in which things happen in the program:

Step By Step in the AgeTeller Program

1. The program finds a variable called Age and sets aside storage space for it as an integer.
2. The program finds a variable called Age_in_days and sets aside storage space for it as a real number.
3. The program encounters a variable assignment statement and puts the value 29 in the location where it has the variable Age stored.
4. A calculation is ordered, so the program multiplies Age times 365.25 and puts the answer into the variable area called Age_in_days.
5. A Readln statement assigns a new value to the variable Age. It carries out that instruction by placing this new value into the place where it has the variable Age stored, replacing the former value of 29 (unless, of course, the answer here is 29 as well).
6. The program is asked to print the person's age in years and in days. It finds the value 32 in the variable Age and the value 10592.250000 in the variable area called Age_in_days, so it dutifully prints these values.

Study this example carefully. It contains some valuable lessons about how and when to manipulate variables and their values. Try hard to understand these ideas, because incorrectly changing variable values is one of the most common and frustrating mistakes programmers make.

Using Readln to Make AgeTeller More Useful

Now that you know about the Readln statement and its power, let's make a slight but important modification to the age-telling program. This time, instead of initializing the Age variable in the program, we'll ask the user to give us his or her age in years. This way, we can have a *generalized* age-telling program that will give the right age in days for any age the user puts in. This will be a far more useful program to show your friends!

The modifications are quite simple. Try them without reading the next listing. (If you save this new version of the program while keeping the former version for some reason, be sure to use the Save As... option from the File menu. It might also be useful to change the name of the program on the first line, as we have done in our solution.)

Ready? Okay, here's our solution. If yours is different but works, don't worry. That may indicate you're more creative than we are!

```
program NewAgeTeller;
 const
  days_in_year = 365.25;
 var
  name:string[80];
  age:integer;
  age_in_days:real;
begin
 Writeln('What is your name, please?');        ← Single quotes are important here !
 Readln(name);
 Writeln('How many years old are you, ',name, '?');
 Readln(age);
 age_in_days: = age*days_in_year;
 Writeln('That means you are about ',age_in_days,' days old.');
end.
```

Readln's Brother, Read

Let's look at a variation in our way of getting input from outside the program. Retrieve the original age-telling program and change the Readln statement to a Read statement. Leave everything else the same. Now Run the program. See the difference?

Read is identical to Readln *except* that Read does not insert a carriage return after the information has been entered. As a result, the following line prints next to the information that has been entered. This is usually not desirable, because it tends to create a confusing screen display with information crowded together and hard to read. But there may be times when we'll want to use this technique, so it's good to know that it's available.

In this section you've learned that the Read and Readln statements accept input from the keyboard and place that information in a variable location in the Macintosh's memory. The type of data being entered must match the type of the variable. You've also learned the importance of being aware of what's going on in a program when you change a variable's value.

Printing in the Macintosh Text Window

You've undoubtedly noticed the Writeln statements in the Pascal programs we've been using. Writeln is a close cousin of Readln. You use Writeln to put messages, questions, and other information into the Macintosh Pascal Text window for the user to see and, if appropriate, respond to. Let's experiment with this important statement to learn how it works.

Get a clean Mac Pascal desktop, with no program in the Program window and the Text window Reset and empty. Now activate the Instant window and type in the following line:

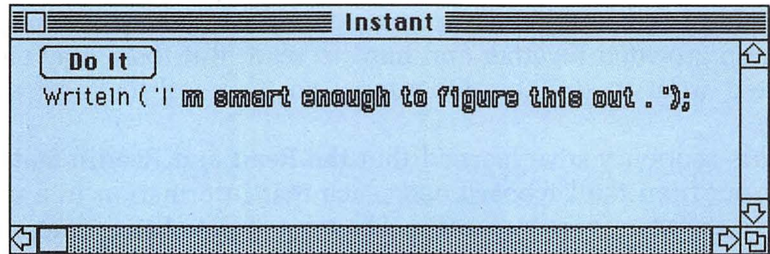
```
Writeln('I am smart enough to figure this out.');
```

Click the Do It button in the Instant window and see the encouraging message appear in the Text window. Anything between the two single quotation marks inside the parentheses in a Writeln statement will be displayed in a program's Text window exactly as written. Try putting other messages — silly or serious — inside those quotation marks. Include numbers, special characters, or anything else you choose.

There's one possible problem. At some point, you might include a single quotation mark in the words you want the program to display. If you haven't already done so, try that now. If you haven't played around with the display much, you can simply edit the line we gave you a few moments ago:

```
Writeln('I'm smart enough to figure this out.');
```

When you press the semicolon to indicate you've reached the end of the line, part of the characters in the string will turn to outline characters, an indication that something has definitely gone wrong:

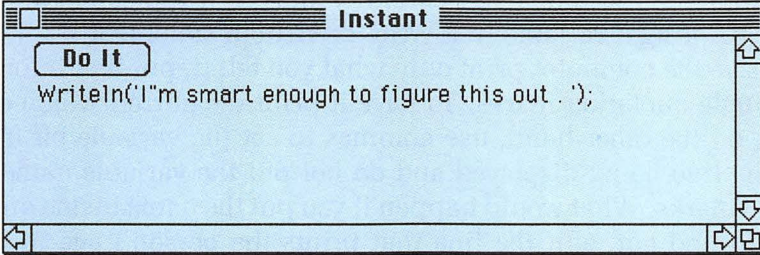


Click the Do It button anyway. You'll get an error message that tells you there's a mistake in the construction of this Pascal statement. What could that mistake be?

Think about the way Pascal looks at the statement we've given it. It sees the Writeln command and says to itself, "Aha! I'm going to put something into the Text window." It then looks for the first parenthesis, which it finds, and the first single quotation mark, which it also finds. Now it looks for the next quotation mark, which should signal the end of the data, and finds it two characters later. But instead of finding another parenthesis and a semicolon right after the quotation marks, it finds the letter *m*. It is confused, so it does what every computer program does when it gets confused — it screams for human help.

Printing Single Quotation Marks and Apostrophes

So how can we ever put an apostrophe (or a single quotation mark) into a printed line of information in the Text window? Simple. Edit the line in the Instant window, changing the apostrophe to two consecutive single quotation marks so it looks like this:



Now click the Do It button, and everything works fine. In order to put an apostrophe or a single quotation mark into text displayed by a program in the Text window, just use two single quotation marks in a row (*not* a double quotation mark).

Using Write for Effect

Want to know a trick to make your programs look professional? Professional programmers often have their programs accept answers on the same line as the question being asked, rather than on the line below. In other words, they set up their questions so that the pointer is left at the end of the question, waiting for the reply.

To do this in Macintosh Pascal, simply substitute the Write statement for the Writeln statement. Write differs from Writeln in the same way that Read is different from Readln; it doesn't insert a carriage return at the end of the line. It simply prints the question and leaves the pointer at the end of that line, waiting for something to happen. If you use Write in your Mac Pascal programs, leave a blank space at the end of the line you're printing and *before* the closing single quotation mark so that the user's typing won't run into the display produced by the program.

Printing Variables in the Text Window

Often the statements printed in the Pascal Text window will include variable information. Two such statements are found in the age-telling program: one tells how old the user is, and the other tells the user's age in days. Examine these statements carefully and see if you can figure out the rules for printing a variable in a Write or Writeln statement.

To have the computer print only what you tell it, put that information inside single quotation marks. To have it print the current value of some variable, on the other hand, use commas to set the variable off from the rest of the line being displayed and *do not* put the variable name inside quotation marks. What would happen if you put the name inside quotation marks? To find out, edit the line that prints the person's age in days so that the quotation marks and commas in the middle of the line are no longer there. The line should then look like this:

```
Writeln('That means you are about age_in_days old.');
```

Now Run the program and see what happens. Pascal has no way of knowing that when we tell it to print "age_in_days", we really mean that it should print the *value* of a *variable* with that name. As before, it simply displays on the screen everything inside the single quotation marks. To print a variable value on the screen, then, you have to set it off from the rest of the line in the Write or Writeln statement.

How Big Is the Text Window?

One final note about the Write and Writeln statements: Macintosh Pascal is smart enough to know how big the Text window is and where it's positioned on the Desktop. You can move it around, make it bigger or smaller (within reason), and you'll still be able to see the whole line being printed by the Write or Writeln statement. Prove that by moving and sizing the Text window, using the Reset option on the Run menu to clear the window. Now Run the program again.

The result should make you sing the Mac's praises once more. On virtually all other machines, you must know precisely where on the screen you want to print something or you may not be able to read it.

Summary

You now know how to make the computer ask questions and furnish information, using Write and Writeln statements. You know how to get information from the user with Read and Readln statements. You know how to give values to variables in two different ways: by using the ":=" operator in the program itself, and by using Read or Readln to get information from the user at the Macintosh keyboard. You can already write programs that do interesting, if not highly useful, things. And we've only just begun!

Mac-r-cises

1. Modify the AgeTeller program to tell the user his or her age in *hours* as well as days.
2. Create a program that will permit you to check the accuracy of a paycheck based on an hourly wage. The program should ask the user for the number of hours worked, the rate of pay per hour, the percentage of income taken out in taxes, and the total of other deductions. It should then print the amount that should appear on the paycheck. (It would be nice, too, if it printed the gross amount of the check before deductions.)
3. Write a program that asks the user for two numbers, divides the first by the second, and prints the result *without using a variable for the result*. (Hint: Division in Pascal is indicated by a slash (/) between numerator and divisor.)

4. Modify the program from Exercise 3 to use a variable to hold the result before it is printed. (What type of variable will it have to be?)
5. (Difficult) Modify the program called "First" from Chapter 2 so that it asks the user for the outside size of the Drawing window to use and the spacing to use between squares and then follows those instructions. In other words, have the user input values where the numbers 190 and 5 now appear in the program. (Hint: Values of variables can be used in place of constants in many cases.)

5

Fundamental Control Statements

When you finish this chapter, you'll know

- **How to use an If statement to help your program make decisions**
- **How to set up a way to avoid rerunning your programs needlessly**
- **How to use While and Repeat statements to tell your programs when to stop running**

When you've finished this chapter, you'll know everything you need to fully "control" your Macintosh Pascal programs and their execution. You will learn how to get programs to choose correctly among several courses of action. You will be able to make programs run several times automatically. You will be able to control the circumstances under which they run. In short, you'll become familiar with a fundamental idea of computer programming: controlling the *flow* of a program.

What If I Want to Do Something Different?

You will often want a computer program to do different things depending on the answer to a question or other information fed into it. In last chapter's sample program, for instance, we would have preferred the computer to "know" the ages of at least a few people and respond with the correct age when we typed in one of those people's names. It would also be nice if it could tell us that we had asked for the age of a person it didn't know. Both of these improvements can easily be made in the program. We'll show you how in this chapter.

You will frequently want a program to ask the user a question and then do something based on the answer. For example, the program might ask, "Is this answer OK?" If the user types the letter Y (for *Yes*) in reply, the program will do one thing, but if the answer is N (for *No*) the program will do something entirely different. This kind of computer action is called *conditional processing*. If means that what the computer does is a *condition* of the input it receives from somewhere.

Looking for the Yes

Let's begin this exploration with this simple program:

```
program LetterChecker;
var
c:char;
begin
write('Type the letter y: ');        ← Extra blank after colon for readability
Readln(c);
if c = 'y' then
   Writeln('Right!');
end.
```

Type this program into your Mac, Save it on a disk, and Run it. When the computer asks you to type in a *y*, do it and see what happens. Then Run the program again and give the program some other answer — any other answer will do — and observe what happens. What can you conclude about the operation of the If statement from what you just did?

The If statement has three basic parts: the word If itself, the condition the program is to check for, and the statement that tells the computer what to do if the condition is valid (true). Figure 5-1 shows these parts marked in the If statement from our example program.

Beginning with If

The first thing in an If statement is the word If. This word alerts Pascal to the fact that a conditional program statement follows. A conditional statement means that if a certain condition (or, sometimes, combination of conditions) is met, the program will do something specific.

The Condition Expression

The condition expression defines the circumstances under which the program will do certain things. It immediately follows the keyword If. In

our example, the condition is fulfilled if the character typed into the program for variable c is equal to (the same as) the lowercase letter *y*. All programming languages, Pascal included, take the equal sign literally. For the condition to be fulfilled, or satisfied, the value of c must be a lowercase *y* and nothing else. Even a capital *Y* won't do.

Remember the computer is a high-speed idiot. It will recognize only the precise thing it was told to look for.

If the Condition is Met, What Then?

The final part of the If statement tells Pascal what to do when the condition is met. This part of the If construction always begins with the keyword *Then*. A single program line can follow the *Then*, as in our present example. However, in more complicated programs, the *Then* will more likely be followed by a *series* of program lines instructing the computer what to do next. We'll look at these more complex situations later.

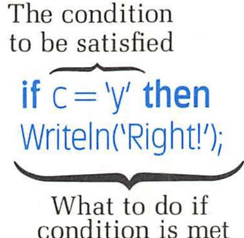
In our present case, a simple Writeln statement like the ones you looked at in Chapter 4 follows *Then*. The Writeln statement tells the user that the answer that has been entered is "right" — that is, it's the one the computer was programmed to look for.

What If the Condition Isn't Met?

You noticed that trying to enter any answer besides *y* produced no effect at all. The program simply ended. This illustrates a fundamental truth about If statements:

> ***If you don't tell the computer what to do when the condition in an If statement is* not *met, it will simply continue with the program.***

Figure 5-1. The Parts of an If Statement



The condition in this example is very simple and straight-forward: if the variable c contains a lowercase y, the program prints "Right!" and quits. However, sometimes the condition to be met in an If statement is quite complex. It may require that the value entered meet some mathematical requirement or match up exactly with one of several words or sentences, for example. In every case, though, if you don't tell Pascal what to do if the answer *differs* from those you've provided, and the answer does differ even a little, the program is going to continue its execution.

Do It. . .or Else!

The next part of the If statement is illustrated by a line we'll add to our program example. After the Writeln statement in the LetterChecker program, add the following lines:

```
else
Writeln ('Oops!');
```

In order for this statement to work correctly (for reasons we'll explain later) you need to delete the semicolon at the end of the first Writeln statement. The program should then look like this:

```
program LetterChecker;
var
c:char;
begin
Write('Type the letter y: ');          ← Extra blank after colon for readability
Readln(c);
if c = 'y' then
   Writeln('Right!')                ← Notice the semicolon is gone
else
   Writeln('Oops!');                ← Be sure this semicolon is present
end.
```

Now Run the program again. First type in the answer the program asks for, then try one or two different answers. Obviously, with the addition of the *Else*, the If statement takes on more flexibility. The computer will now respond to *any* answer the user types when asked to type the letter *y*.

More Complex Use of the If...Then...Else Group

There are a couple of ways in which the If statement and its associated Then and Else statements can be made more complex and useful. For one thing, the program can be told to check for more than two conditions. So

far, the program will give one response if the user types the expected answer and another for all other answers. But what if you wanted to have different responses for each of several different inputs, *plus* one to cover unexpected or undesired answers? A second way of making the program more complex is to have it execute more than one line of Pascal program code in the event of a condition matching or failing to do so. We'll cover both of those circumstances in this section.

Checking for More than One Condition

First, let's learn how to tell Pascal to check for multiple responses and do something different for each response. One of the problems already noted with our small example program is that if the user types in *Y* instead of *y*, he or she will get an "Oops!" message, even though the answer probably was intended. There are several ways to solve that problem; here's a relatively simple one.

Putting Ifs Inside Ifs

Edit the sample program to add two new lines after the first Writeln (not Write) statement, as follows:

```
else if c = 'Y' then
  Writeln('Close, but no horseshoe.')      ← Notice no semicolon here
```

By now your program should look like Figure 5-2. We're sure you can figure out what's going to happen now if you type in a *y*, *Y*, or any other character from the keyboard.

The process of putting Ifs inside other Ifs can go on for a long time. Sometimes this arrangement is the best way of conducting a series of checks to determine what to have the program do next. (At other times you'll want to use "logical operators", which we'll cover later in the book.)

All Things Being Equal. . .or Unequal

We will not always want to check answers just for their equality with some value. Sometimes, we will want to do one thing if the answer is equal and something else depending on *how* it is not equal. The easiest way to show this situation is by using numbers. But first you need to learn about how "relational operators" work.

A Relational Operator

Numbers can be equal to one another, or one can be smaller than the other. You can express the *relation* between two numbers by using symbols called *relational operators*. Table 5-1 lists and defines the relational operators we will use in this book. The first three — equal, greater than, and less than — are almost certainly familiar to you. The other two — greater than or equal to and less than or equal to — may be new, but they are really just combinations of the familiar ones. Let's look at the way these relational operators might be used in a practical program.

We'll use a program that plays a familiar childhood game to demonstrate the relational operators and the way they work with the If...Then...Else statement. The program listing and what it does should be largely self-explanatory. We've pointed out the only line in the listing that contains something new. The line will produce a random number from one to ten. (How it does that will be covered in Chapter 7.)

Here's the program. (Before you type it in, be sure you've saved the LetterChecker program with its modifications on your disk or that you're prepared to type it in again later.

Figure 5-2. LetterChecker Program with Modifications

LetterChecker

```
program LetterChecker;
 var
  c : char;
begin
 Write('Type the letter y: ');
 Readln(c);
 if c = 'y' then
  Writeln('Right!')
 else if c = 'Y' then
  Writeln('Close, but no horseshoe.')
 else
  Writeln('Oops!');
end.
```

```
program GuessIt;
  var
  number,guess:integer;
begin
 number: = random mod 10;        ← Generates random number
 Writeln('I''m thinking of a number');        ← Notice double apostrophe
 Writeln('between 1 and 10.');
 Writeln;        ← Makes screen easier to read by adding a blank line
 Writeln('What is it? ');
 Readln(guess);
 if guess = number then
   Writeln('Fantastic!   You guessed it!')
 else if guess < number then        ← Notice the relational operator
   Writeln('Sorry, you were too low.   It was ',number)
 else
   Writeln('Sorry, you were too high.   It was ',number);
end.
```

The Text window in Figure 5-3 shows how the screen will look when you guess the computer's number incorrectly.

Doing More than One Thing If. . .

Now you know that you can put If statements inside other If statements to get your programs to make a broader range of decisions. You also know that the condition for which you check need not be simple equality between two items; you can use fairly complicated relationships to make such decisions. But what if you want the program to carry out more than one Writeln statement (or other one-line statement) when a particular condition occurs?

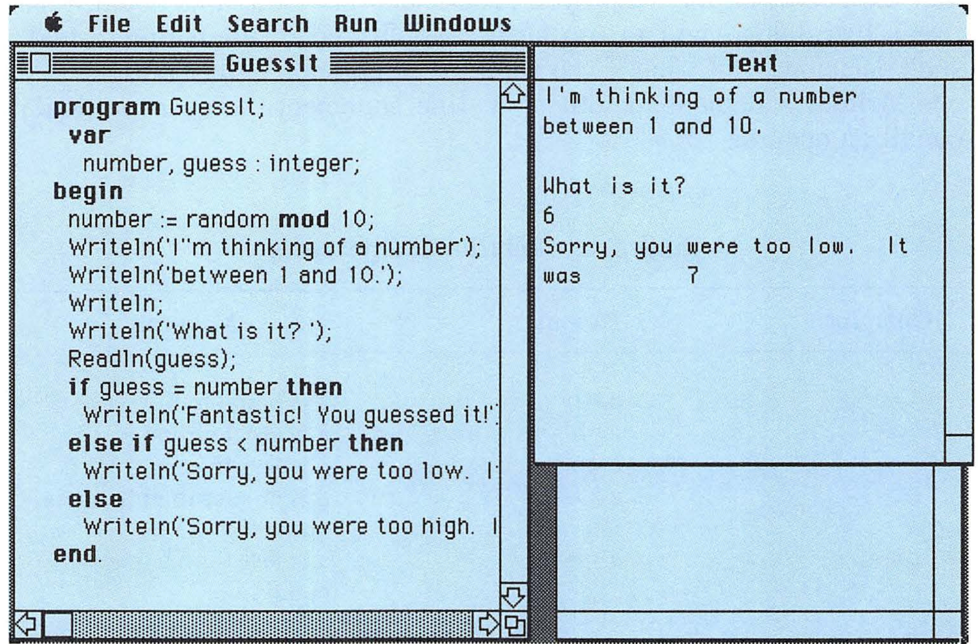
Table 5-1. Relational Operators

Operator	*Format*	*Meaning*
=	a=b	a is equal to b
>	a>b	a is greater than b
<	a<b	a is less than b
>=	a>=b	a is greater than b or a is equal to b
<=	a<=b	a is less than b or a is equal to b

It probably will not surprise you to find that this is possible in Macintosh Pascal. To see how it's done, type the following program into your Mac (after saving the GuessIt program, if you wish) and then Run it.

```
program Friendly;
var
   name,color:string[40];
   any:char;
begin
  Writeln('What''s your name?');
  Readln(name);
  Writeln('Hello, ',name,'.   Tell me your favorite color.');
  Readln(color);
  if color = 'purple' then
    begin
      Writeln('Wow!   Purple is MY favorite color, too!');
      Writeln('Good choice, ',name,'.');
      Writeln;
    end
  else
    Writeln('That''s a nice color, I guess, ',name,'.');
end.
```

Figure 5-3. The Mac Screen after Running GuessIt Program



When you Run this program, your screen should look something like Figure 5-4. (Incidentally, all periods and commas inside the single quotation marks on Writeln statements are optional as far as Mac Pascal is concerned. If you don't mind unpunctuated sentences, you can omit them.) A common error people make here, by the way, is putting a semicolon at the end of the first *end*, just before the *else*. If you do that, the program will not run. Instead, you'll get a Bug box telling us that the next line, the one with the *else* on it, is "not a valid executable statement". This is one of those places where a Pascal line should *not* end with a semicolon.

Begin. Again?

As you typed the program in and looked it over you probably noticed that it had *two* Begin statements. Perhaps you wondered how you could begin something and then begin something else in the middle of the first something.

Pascal programs often use this construction. It is technically referred to as a "compound statement", which is two or more Pascal statements sandwiched between a Begin statement and an End statement. Programs

Figure 5-4. Screen after Running Friendly Program

```
File  Edit  Search  Run  Windows

Friendly
program Friendly;
  var
    name, color : string;
    any : char;
  begin
    Writeln('What''s your name?');
    Readln(name);
    Writeln('Hello, ', name, '. Tell me you
    Readln(color);
    if color = 'purple' then
      begin
        Writeln('Wow! Purple is MY favori
        Writeln('Good choice, ', name, '.');
        Writeln;
        Writeln('Press the RETURN key to g
        Read(any);
      end
    else
```

```
Text
What's your name?
Ernest Hemingway
Hello, Ernest Hemingway.
Tell me your favorite color.
purple
Wow!  Purple is MY favorite
color, too!
Good choice, Ernest
Hemingway.

Press the RETURN key to go
on.
```

would have to be limited to conditional statements one line long if Pascal didn't use this (or some similar) technique to mark the start and finish of a group of statements to be executed together.

One of the beauties (from a programming viewpoint) of using Begin and End statements this way is that Begin-End pairs clearly mark off groups of compound statements. Such groupings make reading a Pascal program written by someone else (or one written so long ago that even the programmer can no longer remember exactly what he or she was trying to do) much easier than it would be without such structure.

Fancy Programming With Ifs

You've seen that we can put If statements inside other If statements, and each statement can contain compound statements to be carried out if its conditions are met. By carefully placing *Ifs* and their associated statements throughout our programs, we can do some very complex programming indeed.

In fact, that's how complicated programs — programs that calculate rocket trajectories for interplanetary space probes, keep track of huge amounts of data in government computers, or play complex strategy games like chess — are organized. The most complex programs are really collections of small subprograms, each of which is used only under specific circumstances. (Sometimes the "specific circumstances" are designed so that a particular subprogram *always* executes — so-called unconditional statements.)

In Chapters 6, 7, and 8, we'll take a look at a Pascal construction called a *procedure*, which is a refinement of the idea of simple building blocks being used to make complicated programs. This way of doing things is called "structured programming". For now, if you understand the basic concept of building blocks in programs, you've come a long way toward being able to design complex and useful Pascal programs.

Just in Case

Nested If statements are great, but sometimes they can get pretty difficult to read, understand, and keep track of, even for the programmer who wrote them in the first place. For situations where we have a number of possible values for a particular variable and we want to do something different for each one, Pascal provides a control statement called the Case statement. It's not used as often as the If statement or some others we'll encounter, but you should become at least somewhat familiar with it.

What Day Is It?

Type in the following small program and Run it after Checking it and Saving it on your Mac disk. (Note that although you may type the program in just as it appears here, the display on your screen will differ slightly; the numbers and colons will appear on separate lines from the statements that follow each number-colon combination. This is an example of Macintosh Pascal's automatic formatting.)

```
program DayFinder;
var
    numdays:integer;
begin
    Writeln('How many days since the last Sunday?');
    Readln(numdays);
    case numdays of
        0:
            Writeln('This must be Sunday.');
        1:
            Writeln('This must be Monday.');
        2:
            Writeln('This must be Tuesday.');
        3:
            Writeln('This must be Wednesday.');
        4:
            Writeln('This must be Thursday.');
        5:
            Writeln('This must be Friday.');
        6:
            Writeln('This must be Saturday.');
        7:
            Writeln('This must be Sunday.');
    end;
end.
```

When this program is run, you'll be asked to tell the computer how many days have passed since the last Sunday. If you enter a zero, obviously today *is* Sunday. If you answer, for example, 3, then it must be Wednesday.

How the Case Statement Works

In a Case statement, each number or value followed by a colon represents one *case* of the value that could be assigned to the variable being checked. In our sample program, the variable Numdays is being checked for one of

eight different values, from 0 through 7. The effect is precisely the same as if you had typed something like this:

```
if numdays = 0 then'
        Writeln('This must be Sunday.')
else if numdays = 1 then
        Writeln('This must be Monday.')
```

and so on for all the other values. However, the Case statement is obviously more efficient and easier to understand.

What If It's Something Else?

There is one small problem with the Case construct as we've presented it so far. Try Running our program and answering, say, 18 when asked how many days have gone by since the last Sunday. You'll get a Bug box that looks like this:

The value of the expression in a CASE statement above does not match any of its case constants.

The Bug box message is largely self-explanatory. We've told Pascal what to do if the user types in any number between 0 and 7, but not what to do if the answer doesn't match one of these cases. To solve that problem, Macintosh Pascal provides the Otherwise statement, which, incidentally, is not part of most Pascal languages available for microcomputers. In versions of Pascal that don't offer such a construction, the programmer has the responsibility of checking to make sure that the value of the variable is within the proper range before the Case statement is executed.

Otherwise, What Do I Do?

The Otherwise statement tells Mac Pascal what to do when the value of the variable being checked doesn't match any of the constants in the list of cases provided. It's the catchall of the Case statement. It is placed at the end of all the constant values we provide, just before the End; statement. To see it at work, add this line to the sample program, right after the line that checks for the value 7:

```
otherwise Writeln('There can''t be more than 7!');
```

Now run the program and try some value other than 0-7 and see what happens. You'll see another example of the Case statement in the next chapter.

Running Programs Repeatedly

So far, each program we've seen could be run only once. We could pull down the Run menu and click on Go or choose some other option that would cause the program to execute one time. To run it a second time, we had to instruct Pascal specifically to run it again via the Run menu.

This approach works fine when we want the answer to one question and we want the answer only once. But there aren't many times in life when that's the case. Most often we'll want to execute a program repeatedly with different data.

There are ways to execute statements in a program more than once without having to run the program again. First, we can tell the program to execute a specific number of times before it ends. For example, if we want to run ten columns of information through a program that calculates the averages of the Monday MacStrikers bowling league, we could set up the program to run 10 times and then quit. The second way of repeating execution is to tell the computer to run the program until a certain condition occurs: until, for instance, we type *No* in response to a question about whether it should go on, or until a variable reaches a certain value, or even until a certain time of day.

This section will discuss the first method of repeated execution — causing the program to run for a number of times that is specified in advance. The next section will cover the other alternative, repeating program execution until a certain situation arises.

"Next!"

In a meat market, a deli counter, or other place where customers are waited on in turn, people sometimes have to take a number and wait for it to be called before placing orders. Every few seconds (or minutes, or, it may seem, hours), a new number is called and you check to see if it's your turn to order yet. If you were a computer, you might think of the process something like this. (This "program" is written in English, *not* Macintosh Pascal, so don't try to enter it into your Mac!)

1. Take a number.
2. Listen for a number to be called.
3. Check your number.

4. If it's your turn, go to Step 5. If not, go back to Step 2.
5. Place your order.
6. Go home.

Between Steps 2 and 4 this program "loops". Its action resembles loops in a string or coils in a rope — it keeps coming back on itself. Loops are used often in programming any computer in almost any language, so they're important to understand. There are many kinds of loops in Macintosh Pascal; the most common and useful is called the For loop.

For What?

Let's begin exploring the For loop by typing in the following little program.

```
program CountTo10;
var
  count:integer;
begin
  for count: = 1 to 10 do         ← New concept—see text
     Write(count);          ← Note: Write, not Writeln
Writeln;
Writeln('Done!');
end.
```

Save this program on disk as CountTo10 and then Run it. The screen should look something like Figure 5-5. The simple example shows the use of the For loop in a straightforward way.

Obviously, it would be easy to change the number following *to* in the For loop line and have the program count to some other number — 25 or 50 or 100 or even 20,987. (If you're feeling adventuresome, try changing the value and running the program a few times. A word of caution, though: Don't make the value following the *to* a negative or a zero. You wouldn't appreciate the results, since nothing will be printed out except "Done!".) Perhaps less obviously, you can also change the value of the Count variable following the := symbol so that counting starts at a point other than the number 1.

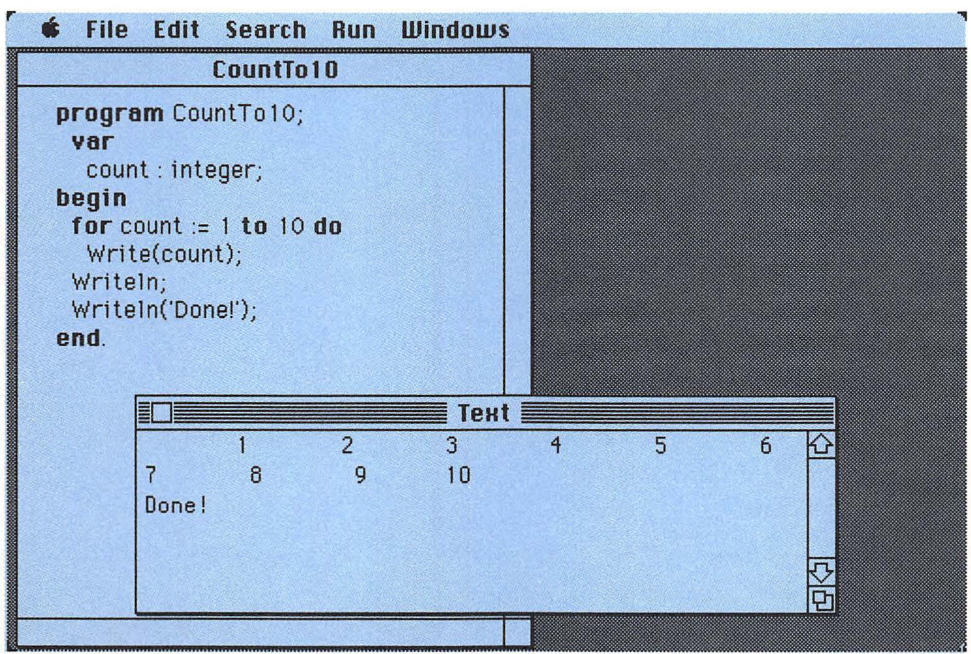
By now it's probably clear to you that the For loop simply starts a variable at the value after the := sign and adds one to it each time it executes the line following the Do command until the value reaches the value specified following the *to* part of the loop. In the original program (before you modified it), the variable Count began at 1, increased by 1 each time through the loop, and quit when it reached 10.

Compound Statements in For Loops

It probably comes as no surprise by now (since you're getting to be an old hand at this Mac Pascal business!) that a compound statement can easily follow the Do portion of the For loop instead of a single statement — write(count) in our example. Just be sure it begins with a Begin statement and ends with an End; statement. Let's use this fact to make our program CountTo10 do something a little more interesting, like printing the numbers from 1 to 10, telling us if they are odd or even, and providing their squares. Edit the program to add the information shown in the next listing, save it as SquaresTo10, and then Run it. Figure 5-6 gives an idea what the results should look like.

```
program SquaresTo10          ← Change program name (optional but good idea)
var
  count:integer;
begin
  for count: = 1 to 10 do
```

Figure 5-5. Execution of CountTo10 Program



```
    begin        ← Start of the compound statement inside the For loop
    Write(count);
     if odd(count) then        ← New line to be added
       Write (' Odd')       ← New line
      else      ← New line
       Write (' Even');       ← New line
     Write(count*count);        ← New line
    Writeln;
   end;        ← End of compound statement inside For loop
end.
```

Cinching the Loop

Notice that an If...Then...Else construction forms part of the compound statement, which itself forms part of the For loop! Also notice that the lines within the If...Then...Else construct do not end in semicolons. The single exception, the *last* line in the group, reads "write ('Even');". You don't need semicolons at the ends of the other lines because, in effect, Macintosh Pascal treats the If...Then...Else construct as a single statement (which, in many ways, it is).

Figure 5-6. Results of Running SquaresTo10 Program

```
 File  Edit  Search  Run  Windows

SquaresTo10

program SquaresTo10;
 var
  count : integer;
begin
 for count := 1 to 10 do
  begin
   Write(count);
   if odd(count) then
    Write(' Odd')
   else
    Write(' Even');
   Write(count * count);
   Writeln;
  end;
 Writeln('Done!');
end.

Text

 1 Odd     1
 2 Even     4
 3 Odd     9
 4 Even    16
 5 Odd    25
 6 Even    36
 7 Odd    49
 8 Even    64
 9 Odd    81
10 Even   100
Done!
```

The program is impressive if you experiment with the values of the variable Count inside the For loop, as we did a few moments ago. To get even trickier, add a line of Macintosh Pascal code to the For loop section to make the program draw in the Drawing window a rectangle of a size determined by the square of the value in the variable Count. (We did this with the very first program in the book, which might provide a clue for how to do it.) If you want to figure this one out on your own, close the book and try it a few times. When you're ready to see our solution, come back.

Incorporating a Graphic Result

Convert the SquaresTo10 program to draw rectangles (which in this case are really squares, too!) by adding the following line right after the line that says "write(count*count);":

```
FrameRect(0,0,count*count,count*count);
```

Make sure the Drawing window on your desktop is active and then Run the program. (If the window isn't active, reactivate it by using the Windows option on the Menu bar.) The results should look like Figure 5-7. Now *that's* impressive! Your friends may be able to count from 1 to 10, tell which numbers are even and odd, and calculate their squares all in their heads, but let's see them draw squares to coincide with those values — and not take all day with the job!

Troubles with For Loops

Sometimes programmers accidentally program For loops that never execute. On the other hand, they occasionally program a For loop that never *stops* executing. If you encounter one or the other of those problems in programs, consider the possibility that you made one of two common mistakes.

A For loop will never execute if the first value of the variable used to count through the loop's execution is larger than the second value. If a computer is given a For loop like this:

```
for count: = 5 to 3 do
```

it won't get very far. Most Pascals (including Macintosh Pascal) simply never execute this kind of loop.

So How Do We Count Backwards?

But what if we *want* a FOR loop to count backwards? Can that be done? The answer is yes; merely change the *to* in the For loop statement to *downto*. Thus while the line

```
for count: = 13 to 5 do
```

results in the loop not being executed, this line:

```
for count: = 13 downto 5 do
```

will execute nine times, ending when the variable Count has a value of 5.

If you're curious about this, modify the SquaresTo10 program to loop from 10 to 1 instead of from 1 to 10 and observe the results.

Figure 5-7. SquaresTo10 Program with Graphics Added

File Edit Search Run Windows

SquaresTo10

```
program SquarestTo10;
  var
    count : integer;
begin
  for count := 1 to 10 do
    begin
      Write(count);
      if odd(count) then
        Write(' Odd')
      else
        Write(' Even');
      Write(count * count);
      FrameRect(0, 0, count * count, cou
      Writeln;
    end;
  Writeln('Done!');
end.
```

Text

```
 9 Odd     81
10 Even    100
Done!
```

Drawing

Never-Ending Loops

Songs are written about love that goes on forever, but loops that go on forever are serious problems (and perhaps one of the most common trials encountered by beginning programmers). There are many ways (all mistakes) to create "endless" or "infinite" loops in Macintosh Pascal. Don't learn them. Just learn what to do to avoid them and how to stop the program if it gets stuck inside one.

How'd I Get Into This Mess?

The only way to create an infinite loop in a For construct is to fiddle with the value of the variable being used to count up or down inside the loop. In our example, if we reset the value of Count inside the For loop to 1, the program will re-execute the statements inside the loop, add 1 to the value of the variable Count, then reset the value to 1 and so go back through the loop again. This problem can arise only in For loops, and it leads to a common-sense rule:

> ***Don't*** **ever** ***alter the value of the counting variable inside a For loop.***

Use the value of the variable (calculate with it, print it, or whatever), but don't *change* it or the program will produce unpredictable results.

If you read the preceding paragraphs too late, and your Mac is already chasing its tail in endless loops, you may be impatient to find out how to stop it without having to reset the system.

Luckily, it is relatively easy to get out of such a problem with the Mac. Just move the pointer (which becomes a cross hair during program execution) to the Pause option on the Menu bar, drag down to Halt, and release. The program will politely stop, with everything intact.

On rare occasions (caused by some unknown bug in Mac Pascal itself or by a strange construction in your program), this doesn't work. Then you have to use the Programmer's Key (the little plastic plug-in push-buttons) on the side of the Mac to reset the entire system. The program will be lost from memory at that point. A more crude but equally effective solution, of course, is simply to switch the Mac's power off and back on.

Sometimes This Many, Sometimes That Many

We are faced with one final issue concerning For loops: What if the programmer doesn't *know* how many times a user will want to execute a loop until the program is executed? Macintosh Pascal deals with this common programming situation by simply allowing the user — before the loop is executed — to enter the number of times the program will loop and then assigning the answer to the upper limit of the variable used in controlling the loop.

Let's look at a simple program that averages numbers to see how the idea can be put into practical use. Type in the following program, Save it on a Macintosh diskette, and Run it. It's self-explanatory.

```
program Averages;
 var
  count,total,number,value:integer;
  average:real;
begin
 total: = 0;
 Writeln('How many numbers do you');
 Write('want me to average? ');
 Readln(number);
 for count: = 1 to number do        ← Notice use of the variable "number"
  begin
   Write('Give me number ',count,': ');
   Readln(value);
   total: = total + value;
  end;
 average: = total/number;
 Writeln('They average ',average);
end.
```

Play with this one a little and let your imagination run in all kinds of directions. Do you see the power inherent in being able, at the time the program is run, to set the number of times the For loop will execute? We have added one more level of control over the program — and, in turn, over the power of Pascal and the Macintosh.

But What If We Don't Know?

The For loop, powerful as it is, has a distinct drawback: We must know before we execute the loop how many times we're going to want it to go through its paces. What about situations where we don't know in advance

how many times we're going to want to do something? Can we make the computer run a set of instructions over and over again until we're done — whenever that is? The computer can't read our minds, can it?

Of course not. But we *can* use another powerful loop construction to carry out this task: the While loop. The While loop is a close cousin of the If...Then...Else construction we met earlier in the chapter. In many situations, it controls the repetition of statements more cleanly and efficiently than the For loop.

While-ing Some Time Away

To get a feeling for While's power, modify the Averages program. Edit it to look like the following listing, change the name to WhileAverages, and Save it on the disk under the new name. Then Run it to see what happens in a While loop.

```
program WhileAverages;
 var
  count,total,value:integer;      ← Delete variable Number; not used
  average:real;
begin
 total:=0;
 count:=1;        ← Initialize this variable
 value:=1;        ← Initialize this variable
 while value <>0 do        ← Replaces lines in former program
  begin
   Write('Give me number ',count,':');
   readln(value);
   total:=total+value;
   count:=count+1;
  end;
 average:=total/(count-2);      ← See discussion in text about the value "2" here
 Writeln('They average ',average);
end.
```

The program will keep asking for numbers forever until someone types in a zero (unless it runs out of memory or the total gets too big to be held in an integer variable). At that point it will stop accepting new numbers and give their average.

We tell the program to compute the average by dividing the total by the value of the variable Count *minus 2*, because if we didn't, Count would be two higher than the number of values we've given. The program will add one to Count the first time through the program, since we put the

"count: = count + 1" line after the main part of the loop, and it will count the zero we input to end the process as another value.

By setting up a predetermined value for a variable that will tell the computer "Okay, I'm done now!", we can control the number of times any program or loop executes without knowing in advance how many times we want to run it. While loops can be used to great advantage, as the next program shows. Save WhileAverages, then type the following into the Mac.

```
program FindMouse;
var
  x,y:integer;
begin
  while not button do
    begin
      GetMouse(x,y);
      Writeln(x,y);
    end;
end.
```

The program has some new things in it, but don't worry about them now. "Not button" and "GetMouse" are useful constructions we'll look at more closely later. The program allows the user to move the Mac mouse all over the top of the desk. The computer will report the mouse's position on the screen's desktop layout by giving the "coordinates" of the mouse's location: the left-hand number is the horizontal coordinate, and the right-hand number is the vertical coordinate. (These coordinates will make more sense when we study Macintosh graphics in detail in Chapter 11.) Now let's do some interesting exploring. For instance, what are the upper, lower, left, and right limits of the desktop? What about the limits for the windows? Where are the Menu bar's upper and lower boundaries?

The program will go on displaying the horizontal and vertical coordinates of the mouse's position (barring a power failure or machine breakdown) until someone clicks on the Mouse button for a second time. At the second click of the mouse, the program stops. To understand why, we'll look at a new type of variable called "boolean".

"A WHOlean?"

If you're the curious type, two questions just popped into your head. First, what in the world is a "boolean"? Second, why are we talking about a new type of variable now, when we covered the subject of variables in Chapter 3? In true cantankerous fashion, we're going to answer the second question first.

There are, as we said in Chapter 3, a wide variety of variable types available in Macintosh Pascal. In fact, the range of kinds of information that can be used in a Pascal program is one of the features that make Pascal such a powerful and useful programming language. In Chapter 3, we introduced only the most basic and commonly used types of variables. Before the end of the book, we'll encounter quite a few more.

Now let's talk about variables of the type we call "boolean". A boolean variable can contain only one of two values: true or false. We've really *seen* boolean situations before, but we haven't looked at them as variables. In an If statement, the condition that we are testing for is going to be true or false, right? In the statement

```
if x = 23 then do
```

either x is equal to 23 or it is not. In other words, the expression "x = 23" is either true or false, so it's a boolean.

If we define a variable to be a boolean type, we can either make it true or false, or determine at some point in the program whether it is true or false. We can do so without using the = operator because Pascal knows that this type of variable can be only true or false. We have a shorthand way of testing its value. Let's work with the variable "button" which appears in the FindMouse program. (This predefined function of Macintosh Pascal does not appear in other versions of Pascal, since they don't have mice, poor things.) To determine the position of the button on the mouse, we don't type "if button = true" or even "if button = down", but simply "if button". Macintosh Pascal understands that we want to do something only if the boolean variable Button has a value of true. Similarly, to find out if the variable button is *false*, we can instruct the program to do something if it is *not* true: "if *not* button".

Because the Mac handles this particular boolean variable on the basis of whether or not the mouse button is down, we can't change its value in our program except by pressing or releasing the mouse button. But we can change the values of other boolean variables in other ways, as the following tidbit shows:

```
program Bool1;
  var
    serveit:boolean;
    answer:char;
begin
  Writeln('Is the steak cooked yet?');
  Readln(answer);
```

```
  if answer = 'y' then
    serveit: = true          ← Sets boolean variable to be true
  else
    serveit: = false;         ← Sets boolean variable to be false
if serveit then          ← Tests boolean variable for truth
    Writeln('It''s ready to eat!')
  else
    Writeln('Sorry.   Not done yet.');
end.
```

This program example could have been written by using an If...Then...Else construct that would display "It's ready to eat!" when the variable Answer was y or "Sorry. Not done yet." if it was anything else. But in many situations, particularly where we want to test for a whole range of conditions, any one of which could cause the answer to be different, the boolean variable will prove invaluable in our Macintosh Pascal programs.

Incidentally, every time we carry out a relational task (that is, when we use =, <, >, <=, or >= operators to test two things against each other), the result of that test is boolean whether we assign it to a boolean variable or not.

What's This Got to Do With While Loops?

We've introduced the boolean type of variable here because it is used most frequently with the While loop construction. In fact, practically the only reason to use a boolean variable is to determine which of two courses of action to take in a particular situation. Boolean variables are sometimes called "flags" by programmers (for some obscure reason). They are said to be "set" if they are true and "cleared" if they are false.

Thus in our little MouseFinder program earlier, the line that says

```
while not button do
```

simply checks the boolean variable button and does the next thing in the program until it finds the variable to be false, at which point it quits and ends the program.

Checking Before or After the Loop Runs

As you have seen, the While loop statement checks a condition for truth or falseness, then either carries out a loop, of instructions or skips to the instructions following the loop depending on what it finds in the boolean

test of the condition. But what if we want to check the value of a boolean expression *after* we run through the loop each time instead of before?

You may wonder, "What difference does it make whether we test the variable at the beginning or the end of the loop, since it's not going to execute the next time if the variable condition isn't met anyway?" The key difference between the two approaches shows in what happens the *first time* the program runs into the loop construction. A While loop will not execute the first time it is encountered if the condition test results in a finding of "false" (that is, if the condition specified is not met).

This is demonstrated in Figures 5-8 and 5-9. If we approached the Las Vegas blackjack dealer in Figure 5-8 and said, "While I have money, deal the cards!" and the dealer happened to be a Macintosh computer, he'd deal us one hand of cards and *then* check to see if we had any money. That's hardly what a Las Vegas blackjack dealer would do, is it?

Repeat Yourself Until You're Blue in the Face!

The other kind of control loop *always* executes the first time through the program and then tests to see if it should run again. This type of loop uses the two-word control statement Repeat-Until. The statement explains

Figure 5-8. "WHILE I Have the Money, Deal the Cards!"

itself, doesn't it? The program Repeats a certain statement or compound statement *Until* the condition specified at the end of the loop is satisfied, at which point it goes on to whatever instructions follow the loop.

Figure 5-9 shows what we'd *really* expect to happen in our mythical blackjack game in Las Vegas. The dealer would be only too happy to Repeat dealing the cards Until our money runs out! But he's also going to check the first time before he deals to be sure we do have money.

To understand the differences between these kinds of statements better, look at two small programs, both of which are designed to test whether the mouse's button is pushed or not. The first uses the While loop and the second uses the Repeat-Until construction. Type these programs into the Mac and Run them.

Figure 5-9. "REPEAT the Deal UNTIL I'm Broke!"

First the While version:

```
program WhileMouse;
begin
  while not button do
    Writeln('Nope');
  Writeln('Yep');
end.
```

When you Run the program the first time, don't hold the mouse button down and observe what happens. Then click the mouse button, and the program will print "Yep" and end. Next, Run it so that, as soon as the program begins, you're holding the mouse button down. What happens? Do you get any "Nope" message? Nope!

Now type in the following relative of the program:

```
program RepeatMouse;
begin
  repeat
    Writeln('Nope');
  until button;
Writeln('Yep!');
end.
```

Now Run the program the same way you ran the WhileMouse program and observe the differences. No matter how hard you try, you can't eliminate the first "Nope" from the screen. Why? Because the program's instructions are to print "Nope" and *then* look at the mouse button.

Summary

In this chapter, you learned how to use the If statement to instruct the computer how to decide among several courses of action. You also learned how to use For loops to avoid having to rerun programs continually when you know how many times you want certain things to be done.

On top of that, you've learned to use the While and Repeat-Until statements. With these you can control the number of times your programs will carry out a given set of instructions even when you don't know in advance how many times you want them to do so. You've learned about a new type of variable, the boolean, as well.

Pascal programs wouldn't amount to much without these important and powerful control statements. Please be sure you understand them before you move on to the next chapter.

Mac-r-cises

1. Modify the AgeTeller program from Chapter 4 so that it:
   a. Recognizes three people by name;
   b. Prints each person's age when his or her name is entered;
   c. Tells the user, "I guess I don't know you." if it's not one of the three you've informed the program about;
   d. Continues to ask for names until the user types "Stop" (or some other word of your choosing).
2. Use the If...Then...Else loop construction to write a program that accepts two numbers from the user and stores the smaller of the two in a variable called Smaller and the larger in a variable called Larger. The program should then print the two numbers in the correct sequence with appropriate labeling.
3. Write a program that prints all the multiples of three between 3 and 90 inclusively.
4. The price of a pound of filet mignon (you *do* eat filet mignon, don't you?) is $4.89. Write a program that will ask the user to input the number of pounds and ounces of filet mignon wanted and then will calculate the price of the purchase. After that, have the program ask the buyer if that amount is OK. If not, let the user get more steaks or put some back until he or she says "yes" when asked if the amount is acceptable. Then have the program say thanks (after all, this person has got to be an important customer, right?)
5. (Difficult) Letters and words can be compared in exactly the same way that numbers can be compared. It's true, for example, that the letter *a* is *less than* the letter *b* (that is, *a* comes earlier in the alphabet). Similarly, if you have the letter *c* and the word *cat*, the relation

```
'c'<'cat'
```

will be true. Using that information, write a program that accepts strings up to 20 characters long from the user and informs him or her whether the word is in the first half of the alphabet (that is, A-M) or the second half (N-Z). The program should continue asking for words until the user types the word *end* or some other word of your choosing.

6

QuickDraw Graphics from Pascal

When you finish this chapter you'll know

- **How to use Mac's pen for drawing**
- **How to create shapes using Mac's built-in power**
- **What a boundary coordinate is**
- **How to insert comments into a program**
- **How to create complex pictures in Pascal**

This chapter will look at some of the more useful and easily understood graphics commands available in Macintosh Pascal. There are dozens of Mac Pascal graphics commands, many of which we'll save until you've had more experience with the Mac and with computer graphics. But even leaving them out, this discussion will add a powerful array of programming weapons to your Macintosh arsenal.

Why Program Graphics?

Before we begin our discussion of graphics commands in Mac Pascal, let's pause to consider why these commands are in the language and why they are worth the trouble to learn. After all, with a program called MacPaint, which makes creating computer art as easy as pointing the mouse and moving it around on the tabletop, why in the world would anyone want to go through all the hassle of writing a program to do graphics? There are three solid reasons for wanting to do this, and understanding them will give you a better grasp of Macintosh Pascal.

Displaying Program Results

First, there are times when you'll need to convert output produced by Macintosh Pascal programs into graphs or other pictorial forms. If a program already generates information for producing such graphics, it is more efficient to have the program generate the graphics as well than to use MacPaint to manually draw the graphs to depict the information. For example, a program that computed household expenses could use figures for the expenditures in several categories to generate a bar graph showing the relative sizes of the expenditures — allowing the user to see, for example, how much was being spent on clothing relative to the amount spent on food or charitable contributions.

Macintosh Pascal's graphics commands can be used to produce graphic representations quite easily. They allow the data produced by a Mac Pascal program to be transformed into bar graphs, pie charts, line graphs and other graphic output, using a relatively small number of easy-to-understand statements.

Impressing Your Friends

Another reason for using Pascal to create graphics, rather than merely using MacPaint, is that program graphics are so impressive and so easy to do. What used to be an arcane art has literally become child's play. (We're serious — our children do a much better job with Mac graphics than we do!) Using some easy programming tricks, it is possible to produce in a few seconds the kind of stunningly beautiful, animated graphic designs that would take a long time to create by hand (even on the Macintosh)! Several example programs in this chapter will show what we mean.

Gaining Insights into the Mac

A third reason for learning to program graphics on the Macintosh is that a great deal of what makes the Mac what it is — the way it arranges the screen, saves information on the disk, and permits manipulation of icons and cursors — is essentially graphic in nature. The better you understand how computer graphics work and what goes into them, the more you'll understand and appreciate the way the Mac works and why it does some of the things it does. It's the difference between merely listening to a Beethoven sonata on the stereo and being able to play the same sonata yourself.

And a Few Other Things . . .

There are many other reasons for learning to program graphics in Macintosh Pascal. You'll discover them for yourself when you begin to study the subject on your own, using the principles in this chapter and the details in your *Macintosh Pascal Reference Manual*.

However, there's one thing to keep in mind about programming graphics as opposed to drawing them with MacPaint. Programming graphics, like programming the solution to any other problem, is a step-by-step process, not to be rushed into or hurried through. We can sit down with MacPaint, begin to doodle, and possibly create something pleasing or even exciting. But if we're going to program in Pascal to create graphics, it helps to do some prior planning of the intended result.

Quick! Draw Me Something!

If you've perused your *Macintosh Pascal Reference Manual*, you've probably noticed a section called "QuickDraw Graphics". Let's focus our attention on some of the commands explained in that section of the manual.

ExploreDraw Program Example

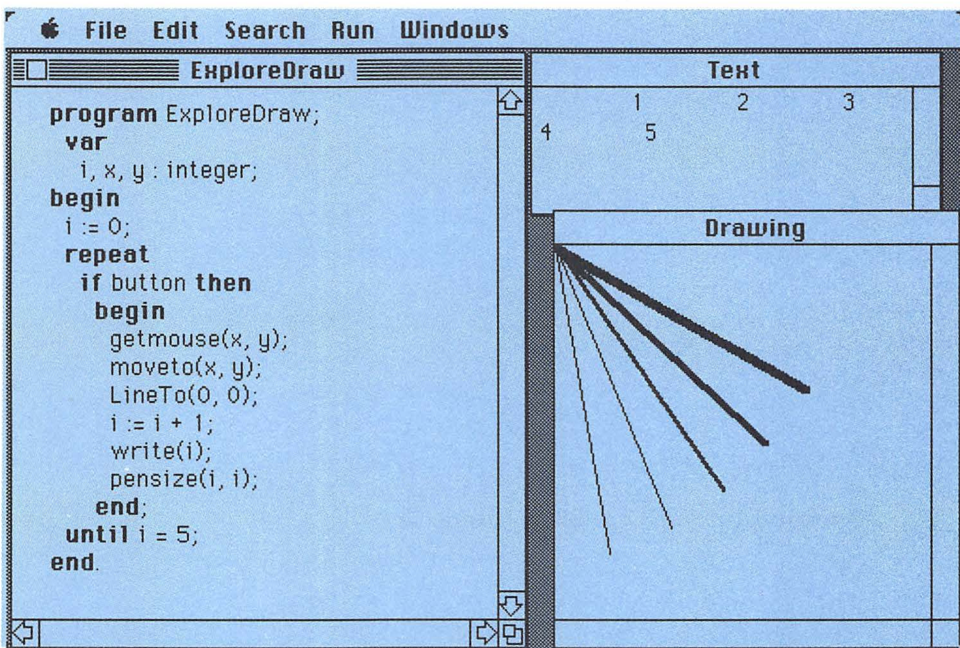
Type in the following program and save it under the name ExploreDraw (or some other name if you like).

```
program ExploreDraw;
 var
  i,x,y:integer;
 begin
  i:=0;
  repeat
   if button then
    begin
     GetMouse(x,y);
     MoveTo(x,y);
     LineTo(0,0);
     i:=i+1;
     Write(i);
     PenSize(i,i);
   end;
 until i=10;      ← Change this number to get more variety!
end.
```

Save the program on the disk, and then run it. When the program begins, position the mouse anywhere in the Drawing window and click it quickly (don't hold it down). If all goes well, two things will happen: The Text window will show the number 1, and a line will appear in the Drawing window, starting from where you clicked the mouse and ending in the upper left corner of the window. Repeat this five times (if you can before the program quits). The result should look something like Figure 6-1.

If the results differ from those shown in Figure 6-1, it is almost certainly because there are fewer than five lines drawn in the Drawing window. The numbers 1-5 appearing in the Text window may seem to indicate that the program drew five lines, but they aren't displayed. That happens because the seventh line of the program says "if button then". When you hold the button down even briefly, the program will check to see whether the button on the mouse is being pressed and, if the answer is yes, the program will draw another line in exactly the same position as the previous one — so you can't see it. This is not a serious problem, and you'll see ways to solve it when you know a little more. For now, just try to be a quicker clicker!

Figure 6-1. Results of Running ExploreDraw Program



This program is short and may seem a little trivial at first, but think how long it would take in MacPaint to: (1) click on the line drawing icon; (2) click on the appropriate size line with which to draw next; (3) position the mouse in the upper left corner of the window; (4) drag down to the place where you want the line to end; (5) release to put the line in place; (6) go back to step 2 and repeat the process four more times, each time using a little bigger pen. Besides that, you'd soon run out of pen sizes in MacPaint while, as we'll see soon, the pen sizes available in Macintosh Pascal are practically limitless.

A Closer Look at ExploreDraw

There are some new statements in the ExploreDraw program. They are: MoveTo(x,y), LineTo(0,0), and PenSize(i,i). Let's examine each of them briefly to gain an understanding of what they do and when to use them.

The MoveTo Statement

The tenth line of the program says:

```
MoveTo(x,y);
```

The MoveTo command instructs Macintosh Pascal to relocate the pen with which it draws lines to a particular location. The pen needs to be told where to start drawing a line and the MoveTo command is one way this is done. Two coordinates, x and y, provide the pen a position from which to begin to draw. In Chapter 5, these coordinates consisted of a pair of numbers defining a point inside a rectangular area. In this case the GetMouse function, which you met in Chapter 5, retrieves the coordinates of the current location of the mouse. The program puts the coordinates into the variables x and y and then tells the Macintosh's invisible drawing pen to move to that location and await its next instruction.

The variables x and y inside the parentheses following MoveTo are called *parameters*. A parameter is a value passed to a routine from the main program (it can also be passed from the routine to the main program).

Calling a routine from your program is a little like asking a coworker to do something for you. Say you want a letter typed. First, you need to know what routine to use: that's like knowing the appropriate person to ask (a secretary, not an electrician). Then you need to know what parameters to pass to the routine: that's knowing what to tell the secretary to do

(which letter to type). In this case, we call MoveTo to move the pen, and we tell it where to move it with the parameters x and y (which of course are given specific values before we call the routine).

The LineTo Statement

The next statement in the ExploreDraw program says:

```
LineTo(0,0);
```

This drawing command instructs the Mac's pen to draw something, in this case a straight line, from its present position (obtained with MoveTo) to a newly defined location, in this case the point defined by the numbers zero and zero. (You may have discovered from your experimenting with the mouse in Chapter 5 that Macintosh Pascal always defines the upper left corner of the Drawing window as point (0,0).)

When we Run the program, move the mouse to various locations inside the Drawing window, and click on the mouse (which causes the "if button" function to become true), the Mac draws a straight line from where the mouse is to the upper left corner of the Drawing window.

The PenSize Statement

The last new statement in the ExploreDraw program is the fourth line from the end of the program:

```
PenSize(i,i);
```

It is obvious from the instruction's name that we are establishing the size of the pen. It may be less obvious why the size has *two* values. The reason is that the Mac's pen has two dimensions: height and width. This fact enables us to define pens of different shapes — long, skinny pens; short, fat pens, and so on. In the present program, each pen is symmetrical; its height and width are the same. To make things more interesting, the program changes the size of the pen for each line drawn, so that the lines become increasingly darker and thicker as the program continues to run.

The first time through the program, the pen has a height and width of 1 (never mind one *what*; it's just a convenient unit of measure) because we set its size parameters to zero as we begin the main part of the program and then add 1 to it before we draw the next line.

But, you might say, wouldn't the pen have a size of (0,0) for its first line, since it draws that line *before* the variable i is set to the value of 1? Very observant! But the answer is "no". The Mac pen assumes a certain size if we don't tell it anything different. Since the program's first drawing instruction (the LineTo command we discussed earlier) appears before we give the pen any size, Mac uses this "default" pen size for its first line. The first line on the left side of the fan-shaped pattern in Figure 6-1 uses the default pen size, while the one just to its right uses a pen size of (1,1). Can you tell the difference? The default size to which the Mac automatically sets the pen, if not instructed differently, is apparently (1,2): twice as high as it is wide.

Modifying ExploreDraw

Now let's get back to programming the Mac's graphics! We called this program ExploreDraw so you'd get the idea that we want you to explore the drawing process with it. Let's look at some ways you might have some fun with the program.

Here's One Idea

The Mac doesn't always have to draw its lines from where the mouse is clicked to the upper left corner of the Drawing window. Let's change the program so that it always draws a line in the same direction, say 25 units long.

Simply edit the line that now says:

```
LineTo(0,0);
```

so that it reads instead:

```
LineTo(x + 25,y + 25);
```

Now Run the new program and see what kinds of effects can be created by positioning the mouse at various starting points. Have fun!

If you get tired of lines 25 units long, make some that are, say, 10 units long, and some 100 units long. How long a line can you draw without having it disappear? How short a line is visible to your eye?

Now Try This

After exploring the effects of changing the destination of the line, try playing with the PenSize command. You can get some startling effects by using different pen sizes.

First, change the line that now reads:

```
PenSize(i,i);
```

so that it reads instead:

```
PenSize(1,5);
```

Draw some lines with this "new" pen and examine the results. Play with different combinations of pen width and height to see what effects you can get and to determine for yourself how height and width are related to each other in the pen's shape.

Try making the pen's shape a function of the number of lines drawn but in a different way than in the previous example. Something like the following statement will do that:

```
PenSize(i + 5,i + 10);
```

You're on Your Own

Take some time to try different kinds of things with the MoveTo, LineTo, and PenSize parameters and see how much variety you can get out of using just the simple commands you've met so far. After doing some hands-on exploration, come back to the book and we'll expand your horizons a bit.

Random Art

One of the most enjoyable things to do with computers is to make them do things that are unexpected but still under our control. The results can be startling, and quite creative things can be done with them. In this section we'll take a look at the idea of randomness, and apply it to Macintosh graphics in a way we think you'll find intriguing.

Putting paint randomly on canvas can have interesting and even pleasing effects (as some modern artists seem to have discovered). We leave to philosophers the question of whether such unintentionally created effects are art. In any case, you can "train" your Macintosh to draw infinite varieties of lines and shapes at random for your amusement or that of your friends.

Randomness in Macintosh Pascal

Macintosh Pascal has a convenient little tool called a *random number generator*. If treated properly, it will produce random numbers within a range of values you specify. Here is the form of the command:

```
x: = Random mod y;
```

Don't worry for now about what the "mod" means; we'll explain that in Chapter 8. For now, just be aware that this program statement will produce a number picked at random from between zero and the number supplied following "mod" (but not including this number). Thus if you type:

```
x: = Random mod 10;
```

you will create a random number between 0 and 9. Since we usually don't want zero to be one of the random numbers generated, we most often set up a random number sequence to look like this:

```
x: = (Random mod 10) + 1;
```

This line will generate a random number between 1 and 10, which is the kind of thing we usually want a computer program to do.

The RandomArt Programs

In the next few pages we'll look at two closely related but different programs that generate pattern after pattern of lines to look at, exclaim over, impress friends with, and (perhaps) print for posterity. In the course of using these programs, you'll learn a little about randomness and the dramatic effects it can have. You'll also see an example of something Pascal graphics can do that MacPaint can't.

RandomArt1

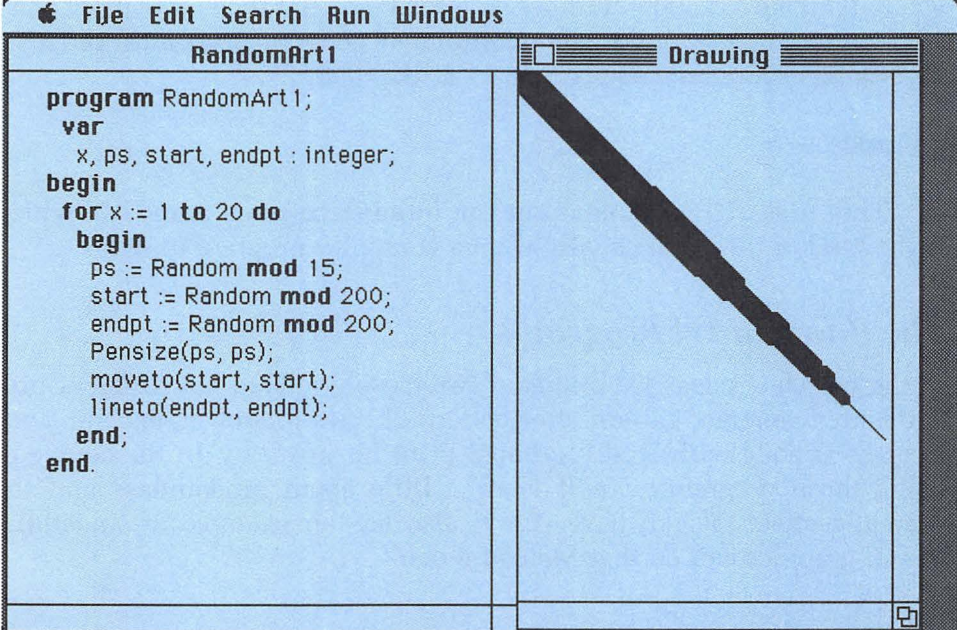
The first program is shown in Figure 6-2. The figure shows a listing of the program in the Program window and a sample run in the Drawing window. (We've erased the Text window and made the Drawing window larger; do the same if you wish.) Type in the program and save it as RandomArt1. Then Run the program a few times.

Undoubtedly you've noticed that the results of the program as displayed in the Drawing window are regular and symmetrical. The pattern is always from the lower right corner of the Drawing window to the upper left corner (with "corner" used very approximately here) and consists of many lines of different thicknesses, all overlaying each other. Some of the effects are interesting, but in general they are quite predictable.

RandomArt2

Now clear the Mac desktop and type in the next program. It looks very similar to the first random art program, but its effects are far less predictable. See if you can figure out why.

Figure 6-2. RandomArt2 Program and Results



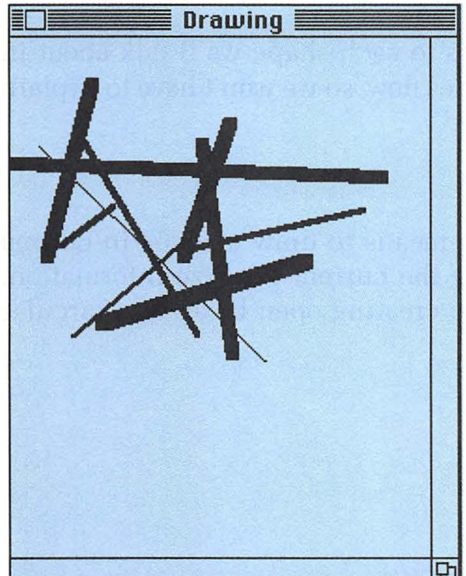

```
program RandomArt1;
  var
    x, ps, start, endpt : integer;
begin
  for x := 1 to 20 do
   begin
    ps := Random mod 15;
    start := Random mod 200;
    endpt := Random mod 200;
    Pensize(ps, ps);
    moveto(start, start);
    lineto(endpt, endpt);
   end;
end.
```

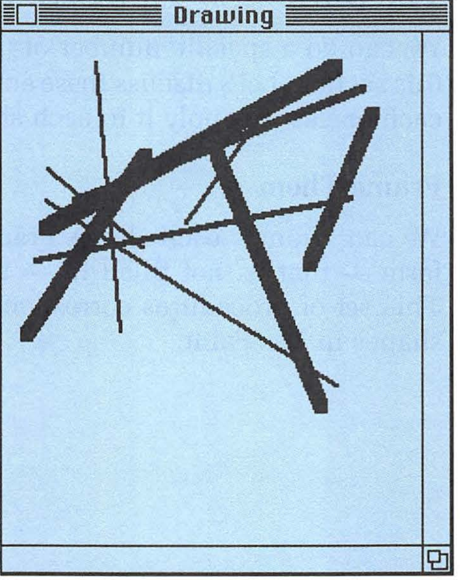
```
program RandomArt2;
  var
    x,ps,start1,start2,endpt1,endpt2:integer;
begin
  for x: = 1 to 20 do
    begin
      ps: = Random mod 10;
      start1: = Random mod 200;
      start2: = Random mod 200;
      endpt1: = Random mod 200;
      endpt2: = Random mod 200;
      PenSize(ps,ps);
      moveto(start1,start2);
      lineto(endpt1,endpt2);
    end;
end.
```

Save the program and then Run it a few times. Notice that the results are strikingly different from those of RandomArt1. Figure 6-3 shows the results we got when we tested the program for this book.

What causes the difference between the symmetry of RandomArt1 and the truly random look of RandomArt2? If you said it was because

Figure 6-3. RandomArt2 Run Results





RandomArt2 didn't require the starting or ending point to have a pair of identical coordinates, you're right. We set up two separate variables for the beginning position (start1 and start2) and for the ending position (endpt1 and endpt2). As a result, each line starts in a truly random place and ends in one, too. RandomArt1 used the same value for both coordinates of the beginning and ending points, which forced the drawing to take a 45-degree orientation. That's why all the drawings with that program are on an angle from southeast to northwest.

Now have some fun with RandomArt2 by changing things like PenSize, number of loops to execute, and range of starting and ending values. Here's an example:

```
start1:= Random mod 135;
start2:= Random mod 300;
```

We hope you'll find this program a lot of fun to experiment with.

Getting Shapelier with Our Art

You've seen the interesting and varied things that can be done with Macintosh Pascal's random number function, using only straight lines. Now it's time to go on to shapes: rectangles, circles, ovals, squares, and so on. Even greater graphic wonders will be revealed.

The Shapes of Things to Come

We can do a specific number of things to each shape we'll talk about in this section. Let's discuss these activities now, so we won't have to explain each one as we apply it to each shape.

Frame Them

We can "frame" each shape. Framing means to draw a shape in outline form — that is, not filled in — using the current PenSize information. This set of procedures corresponds to creating open boxes and circular shapes in MacPaint.

Paint Them

We can also *paint* the shapes — draw them and fill their interiors with a pattern. The Macintosh pen's current characteristics are used to perform this function. The pattern with which the pen draws can be changed from its normal solid black to shades of gray. All MacPaint patterns can be used. Whichever pattern is presently in use for the pen will be used to paint a shape.

Fill Them

This procedure is identical to painting except that we can tell Mac what pattern to use to fill the shapes without having to alter the characteristics of the pen.

Erase Them

We can, of course, *erase* shapes we've drawn — or parts of them. You'll discover erase procedures that create the illusion of motion. You will be able to paint a circle, then erase its center portion to create a ring. You can use a combination of painting and erasing to create a target pattern. You will also be able to erase whole portions of the Drawing window, or even the entire window, so you can start over again.

Invert Them

Inverting a shape looks like erasing it in some cases and like doing nothing to it in other cases. An "invert" command turns all black areas white and all white areas black. Thus, when a shape is painted and then inverted, it turns white and seems to "disappear" — that is, it blends into the white of the Drawing window. When a pattern that alternates black and white squares is inverted, however, the pattern will swap and seem to shift a little, but otherwise it will look very much the same.

Inverting a shape can have a noticeable and dramatic effect if we use complex patterns for a pen drawing. We can get a "shadow" effect by altering the basic gray patterns and schemes. You'll see some of these effects a little later.

Figure 6-4 shows the basic ideas of Frame, Paint, Invert, and Erase using two rectangles. The first shape, Figure 6-4a, results from executing this statement:

```
FrameRect(0,0,150,150);
```

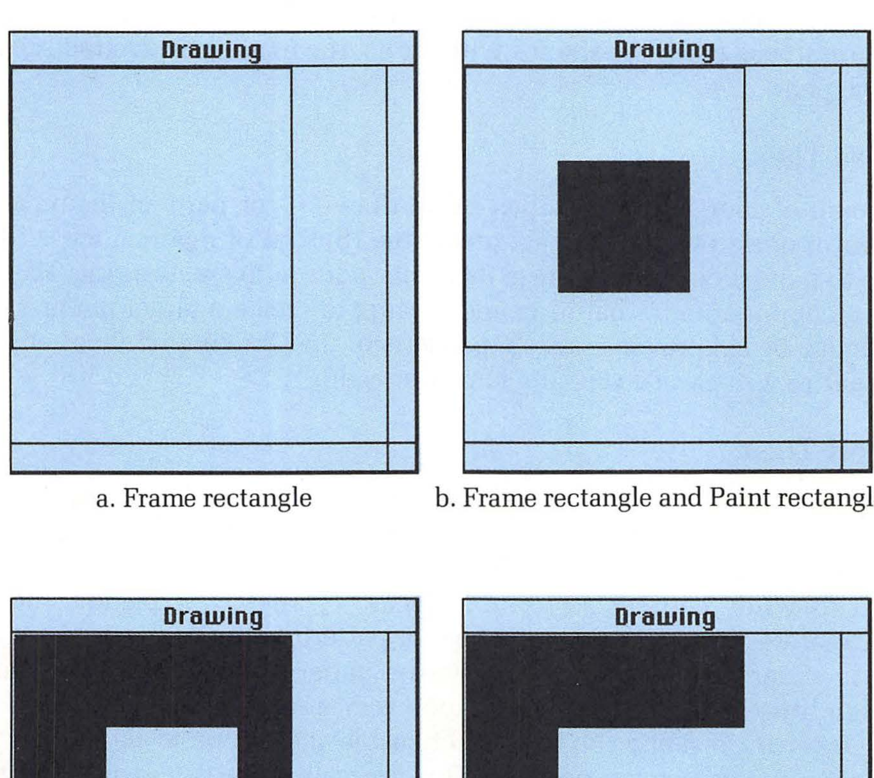
in the Instant window. We got the effect in Figure 6-4b by adding the line

```
PaintRect(50,50,120,120);
```

after the earlier line and running both of them. Then we typed

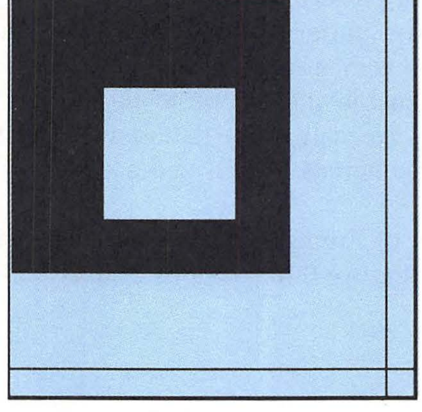
```
InvertRect(0,0,150,150);
```

Figure 6-4. Frame, Paint, Invert, and Erase Rectangles

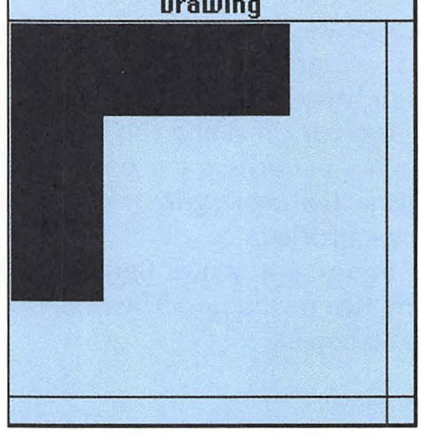


a. Frame rectangle

b. Frame rectangle and Paint rectangle



c. Invert the larger rectangle



d. Erase rectangle

to get the effect in Figure 6-4c. Notice that the large outside rectangle is now dark as if it were painted, while the smaller rectangle inside the larger one is white as if it had been framed. This is the general effect of an Invert command with any of the shapes. Finally, we typed

```
EraseRect(50,50,150,150);
```

to get the effect in Figure 6-4d. Notice that the small inside rectangle and the black "border" on its bottom and right sides have disappeared; they've been erased.

The Shapes We're In

This discussion will concentrate on the three main shapes available in Macintosh's QuickDraw Graphics routines: rectangles, round-cornered rectangles, and ovals. Each shape can be framed, painted, filled, erased, and inverted. Table 6-1 summarizes the commands used to accomplish these tasks with each shape.

The commands are self-explanatory once you know about the three shapes and the five things you can do with them. From this small set of commands come powerful graphic creation tools. Now let's explore some programs that use these shapes and commands.

Meet a Flasher

Type the following program, save it as Flasher, and Run it. It will make an oval shape appear and disappear, with brief pauses between each

Table 6-1. Commands Using Shapes in Macintosh Pascal

	Rectangle	*Round Rectangle*	*Oval*
Frame	FrameRect	FrameRoundRect	FrameOval
Paint	PaintRect	PaintRoundRect	PaintOval
Fill	FillRect	FillRoundRect	FillOval
Erase	EraseRect	EraseRoundRect	EraseOval
Invert	InvertRect	InvertRoundRect	InvertOval

appearance. This program shows the effect of the InvertOval when you invert an oval of the same size as the one you just created using PaintOval.

```
program Flasher;
 var
  x,y,z:integer;
 begin
  for x: = 1 to 10 do
   begin
    PaintOval(75,75,155,190);
    for y: = 1 to 50 do
     z: = z + 1;
    InvertOval(75,75,155,190);
    for y: = 1 to 50 do
     z: = z + 1;
   end;
 end.
```

Imagine painting an oval, inverting or erasing it, moving it slightly in some direction as you paint it the next time, and repeating the process. Can you see how this could give the effect of animation? In fact, that's how virtually all computer animation is done: a shape is created, erased, created again at a very nearby location, erased, and so on.

Feel free to experiment with Flasher. Try, for example, changing the shape from oval to rectangle or round-cornered rectangle. Relocate the shape inside the Drawing window. (Don't forget to change *both* the PaintOval and the InvertOval commands, or the results could be strange!) You may "lose" some or all of the shape if you position it partly outside the Drawing window. Recall from Chapter 5 that the window normally has the location (0,0) for its upper left corner (which doesn't change even when the window is resized or moved) and the point called (200,200) for its lower right corner (which *can* change as the window is resized or moved).

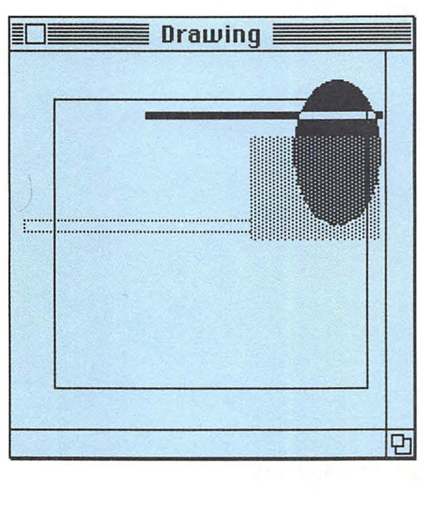
Macasso!

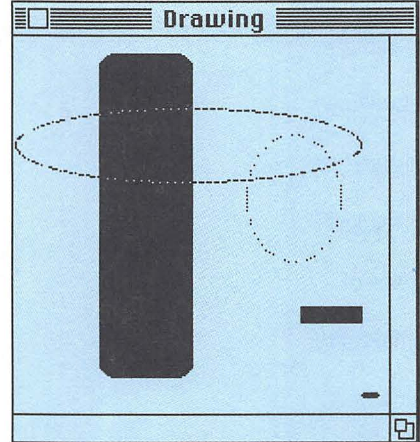
With apologies to the memory of Pablo Picasso, we're going to call our next "art" program Macasso. It combines randomness and shapes and also lets us renew acquaintance with a friend from Chapter 5, the Case statement. It introduces the concept of the pen pattern, which can be specified using the PenPat command. You may notice some other unfamiliar things, too; we'll explain them shortly. This is our longest program to date, so be sure to Save it before Running it, just in case.

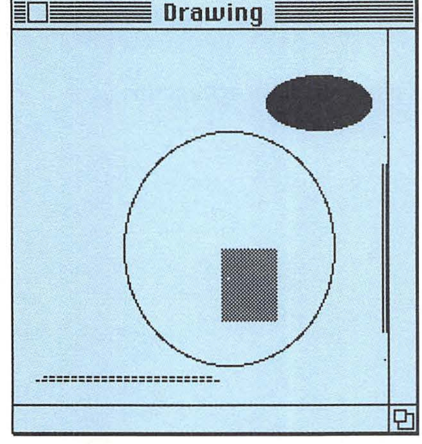
```
program Macasso;
var
  left,top,right,bottom,shape,x:integer;
  r:Rect;          ← We'll come back to this
begin
  for x: = 1 to 5 do
   begin
    left: = Random mod 200;
    top: = Random mod 200;
    right: = left + (Random mod (200-left)) + 1;          ← Ensures a visible shape is created
    bottom: = top + (Random  mod  (200-top)) + 1;
    Shape: = (Random mod 9) + 1;
    SetRect(r,top,left,bottom,right);          ← Yes, this is new, too
    case x of
     1:
       PenPat(gray);          ← This line displays as two but can be entered as one
     2:
       PenPat(black);
     3:
       PenPat(ltgray);
     4:
       PenPat(dkgray);
     5:
       PenPat(black);
    end;
    case Shape of
     1:
       FrameRect(r);
     2:
       FrameOval(r);
     3:
       FrameRoundrect(r,20,15);          ← Note the two extra parameters!
     4:
       PaintRect(r);
     5:
       PaintOval(r);
     6:
       PaintRoundRect(r,20,15);          ← All RoundRect operations have extra terms
     7:
       InvertRect(r);
     8:
       InvertOval(r);
     9:
       InvertRoundRect(R,20,15);
    end;
   end;
end.
```

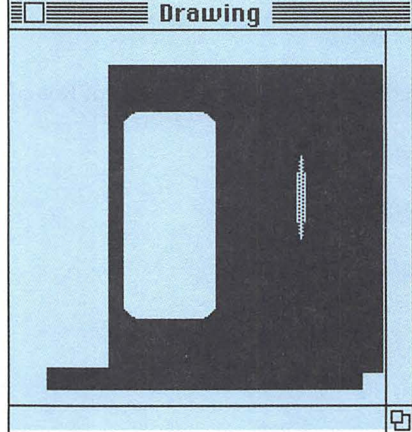
By running this program a number of times, you should get some quite artistic drawings. We recommend that you save the best for posterity in your personal Macasso gallery. When you get a picture you like, print the active window by holding down the [Shift], [⌘] and "4" keys simultaneously. Be sure to click in the Drawing window before doing so, or you'll end up with a printout of part of the program listing. Figure 6-5 displays a mini-gallery of some of the abstract art we created.

Figure 6-5. Mini-Gallery of Macasso Masterpieces









Now let's talk a little more about the Macasso program. The discussion will be easier to follow if you have a printed listing of the program, so we suggest that you make one before going on.

The Rect Variable Type

Notice the second line in the variable declaration section of the program. It defines a variable called r to be of type Rect. This variable type is used only in QuickDraw graphics. It is predefined by Macintosh Pascal. When a variable is declared to be of type Rect, it must be assigned four individual coordinate values in order to be used later in the program. That's what we do a little later with the SetRect statement.

The SetRect Statement

The SetRect statement, which appears just before the first Case statement, establishes a rectangle type variable called r to be positioned in the area defined by the variables Left, Top, Right and Bottom. SetRect doesn't result in the rectangle being displayed; it merely assigns the four boundary coordinates created earlier to a single variable. This keeps us from having to type things like:

```
FrameOval(left,top,right,bottom);
```

SetRect acts as a kind of two-dimensional assignment statement for our rectangle variable.

Setting Limits on Randomness

Since the Drawing window normally has a range of (0,0) to (200,200), a "mod 200" limit is set on all of the positioning variables created in this program. That keeps the program from drawing a shape outside the Drawing window, where we couldn't see it.

We also need to set limits on the relative positions of the parts of each shape in order to make sure the shape is visible. If we let a shape have, say, a left position of 50 and a right position of 32, the shape would be invisible because it would be "inside out" — in other words, the right edge would be farther left than the left edge! Under these circumstances, QuickDraw Graphics simply wouldn't draw anything we could see. If the top of the shape was at position 140 and the bottom was at 139, we would also get an invisible shape.

To keep this from happening, we define two of the shape coordinate variables in terms of others to which their relative position is important. This makes sure that we get a Left that is truly to the left of the Right position, and a Top that is actually higher in the window than the Bottom value. It wouldn't matter whether we defined Right in terms of Left or vice versa. In the Macasso program, we define Right to be determined by the value of Left plus some other values we calculate. Similarly, we define Bottom to be calculated by adding some values to the Top value. Using the value of a variable to define the value of another variable is a handy programming technique when the relationship between the variables is important in the program.

Pen Patterns

The next group of statements that introduces something new is the first set of Case statements. Here, the variable x counts the number of times to create and display a shape. It also determines which pen pattern to use. Macintosh Pascal has four predefined pen patterns: black, dark gray, light gray, and gray. We simply assign one of these pen patterns to each value of the counter variable x so that each drawing will include each pen pattern and will have two shapes that use the black pen pattern.

Incidentally, you can define your own pen patterns using an advanced technique covered in the Macintosh Pascal *Reference Manual*. You can give these patterns names and use them in PenPat statements to create all kinds of intriguing special effects. When you're comfortable with QuickDraw Graphics, you might enjoy a spell of pattern creating.

Creating the Shapes Themselves

The last part of the Macasso program creates and displays the shapes, using the value in the randomly generated variable called Shape to ensure that a variety of shapes is generated. Each shape will be drawn, painted, or inverted using the pen pattern in effect for that particular loop through the program.

Notice that three lines use two values besides that of the rectangle variable r and that all three of these lines involve round-cornered rectangles (RoundRect). In order to create a round-cornered rectangular shape, QuickDraw Graphics needs to know how round to make the corners, and these two values give it that information.

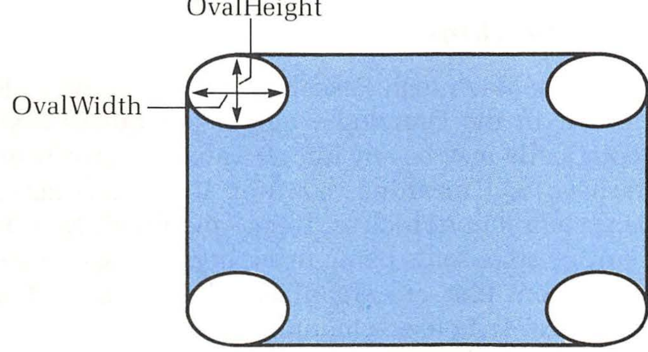
Figure 6-6 depicts the information QuickDraw Graphics needs. Notice that the rounded portion of each corner can be thought of as a small oval. Since all ovals have a width and a height, we can tell QuickDraw Graphics how much to round the corners of our round rectangles by giving it the width and height of these ovals. Thus, to define a round-cornered rectangle, we give Macintosh Pascal six parameters (pieces of information): four to define the boundary coordinates, one for the width of the oval to use for rounding corners, and one for the height of that oval. (We'll explain more about parameters in the next chapter.)

Do Your Own Thing

Now that you understand the Macasso program, have some fun modifying various aspects of it to see what happens. You'll discover dozens of things to do, but just to get the creative juices flowing, we'll suggest three:

1. Change the number of shapes created by the program. (For *more* than five loops, modify the first Case statement so that it has a pattern for x values of x greater than 5.)
2. Eliminate small shapes by using a value greater than 1 to add to the Left and Top variables to create the Right and Bottom variables. (The fact that some shapes are essentially 1 by 1 explains why sometimes it doesn't *appear* that Macasso drew five shapes. It really did; you just can't see them.)
3. Shift the pen patterns around or use one of them more than once or twice in the list to see whether the resulting artwork is more pleasing.

Figure 6-6. Width, Height Parameters for RoundRect Shapes



Adding Writing to Your Artwork

The last aspect of QuickDraw Graphics we'll examine in this chapter is that of displaying text in the Drawing window. There are times when you might want, for example, to label a piece of artwork or indicate the values represented by a graph. These functions require mixing graphics and text in the same window. Macintosh's QuickDraw Graphics library incorporates a number of text-generation and manipulation commands. Let's look at a few of them.

Type the following program into the Program window and Save it on disk.

```
program TextWork;
 var
  Font,Size:integer;
  Line:string;
begin
 Writeln('Enter the font number: ');
 Readln(Font);
 Writeln('Enter the size in points: ');
 Readln(Size);
 Writeln('What do you want typed? ');
 Readln(Line);
 FrameRect(25,0,65,150);
 MoveTo(2,50);
 TextFont(Font);
 TextSize(Size);
 DrawString(Line);
end.
```

This program introduces three new statements: TextFont, TextSize, and DrawString (which has nothing to do with closing burlap bags or sweatpants). Let's look briefly at these three statements.

Font Selection with Textfont

The Textfont statement tells Macintosh Pascal which font (type style) to use when displaying text in the Drawing window. It requires a single parameter, which theoretically may be any integer value. Relatively few of the possible integers have real meaning, however. If you use zero, the program will use the system font (which is, unless modified, New York). Other values will result in other fonts being used, depending on how the fonts in the System and Font files are organized. (If this doesn't make sense, take a look at the MacWrite user's manual.)

Figure 6-7 shows how one Mac used font number 5 to write its nickname. This font is called Venice.

How Big Are Your Letters?

Using the TextSize statement, you can set characters in the Drawing window to be as small or as big as you like. There are, of course, two practical limits: If letters are too small, they are indistinguishable from one another; if too large, they won't all fit inside the Drawing window.

Size of characters is given in *points*. Technically, a point is about 1/72 of an inch. Thus, 9-point type is 9/72 or 1/8 inch high, 24-point type is 1/3 inch high, and so on.

Draw on the Result

After you've selected the size and font for your text — or simply left them up to Mac Pascal by omitting their specifications from your program — use the DrawString statement to *draw* the *string* in the Drawing window.

Figure 6-7. Text in Drawing Window Using Venice Font


```
program Textwork;
  var
    Font, Size : Integer;
    Line : string;
    Letter : Char;
begin
  Writeln('Enter the font number: ');
  Readln(Font);
  Writeln('Enter the size in points: ');
  Readln(Size);
  Writeln('What do you want typed? ');
  Readln(Line);
  Framerect(25, 0, 65, 150);
  Moveto(2, 50);
  Textfont(Font);
  Textsize(Size);
  Drawstring(Line);
end.
```

Since this is a drawing, even though it looks like text, you must use a MoveTo command to position the Mac's pen where desired.

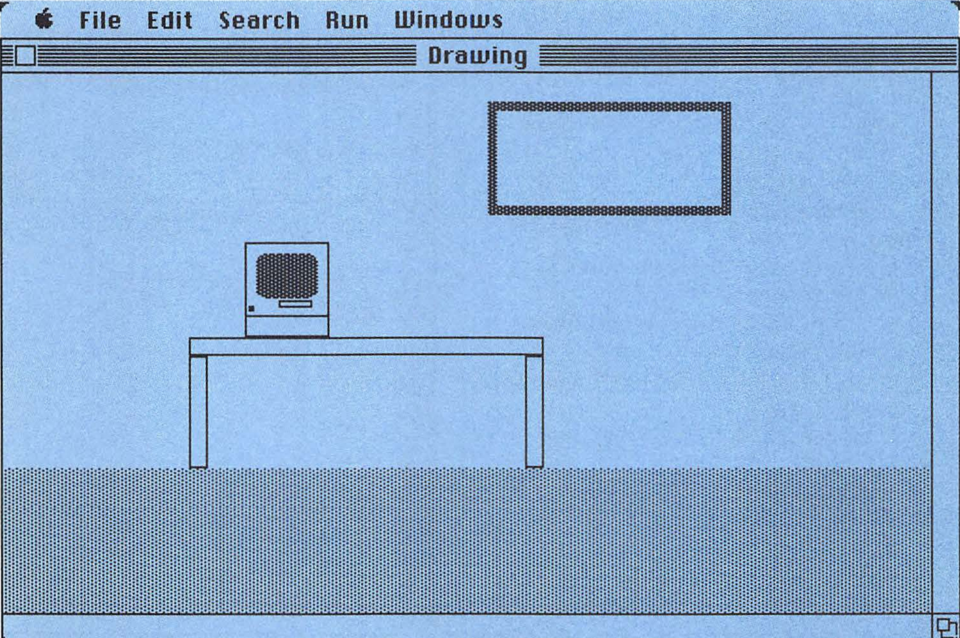
Keep in mind that DrawString doesn't know about window borders, carriage returns, and things like that. It will blithely print as long as it has text to draw. If you want to make *practical* use of DrawString, do some experimenting to find out how to position it correctly.

Getting the Picture

As a fitting end to this chapter on QuickDraw Graphics, let's play with a program that produces a picture having some complexity and containing objects that someone else can recognize — something like Figure 6-8. This program will introduce one more new idea: that of relocating and resizing the Drawing window.

You'll notice that our program includes material between curly brackets on several of the lines. As you may remember, such things are *comments* used to tell people what's going on in a program. Macintosh Pascal will ignore anything between curly brackets when it executes a program, although it *will* display or print the material between the brackets when you examine the program.

Figure 6-8. Drawing of MacStillLife



```
program MacStillLife;
 var
   r:rect;
begin
   HideAll;   {Erase screen}
   SetRect(r,0,38,511,341);   {Set size of window}
   SetDrawingRect(r);   {Apply size to drawing window}
   ShowDrawing;
   FrameRect(150,100,210,110); {Left leg of table}
   FrameRect(150,280,210,290); {Right leg of table}
   FrameRect(140,100,150,290);{Table Top}
   FrameRect(89,130,140,175);{Outline of Macintosh}
   MoveTo(130,128);
   LineTo(174,128);{Protrusion below Mac disk drive}
   FrameRect(120,148,124,166);{Disk drive slot}
   FillRoundRect(95,136,119,169,15,10,dkgray);{Mac display screen}
   PaintOval(123,132,126,135);{Apple logo spot}
   PenSize(100,100);
   MoveTo(0,210);
   PenPat(ltgray);
   LineTo(500,210);{floor}
   PenPat(dkgray);
   PenSize(5,5);
   FrameRect(15,260,75,390);{Picture frame on wall}
   PenNormal
end.
```

Running the Program and Returning to "Normal"

Once the program is typed in, by all means Save it! Don't risk having to retype such a long program. Note that the program erases the Program window when it runs. It is easy to get the listing back, of course, but sometimes people get frightened when they see their programs disappear.

Run the program and observe what happens. When the program finishes, the screen will contain only a large Drawing window with your picture in it. After you've admired the art work, get the program itself back on the display so we can look at a couple of things. To do so, either simply pull down the Windows menu and click on the first item or use the Quit option on the File menu, return to the beginning of Mac Pascal and restart. Then, load the program into the Mac's memory and you'll be all set.

Moving Windows Around

Macintosh Pascal doesn't let us create new windows easily, but it *does* permit us to manipulate the windows we have. The first part of the MacStillLife program shows how to do this with the Drawing window. First, the HideAll command, as its name suggests, hides all of the windows. Next, the SetDrawingRect(r) statement assigns the rectangular boundaries defined by the variable r (which is declared to be of type Rect) to the Drawing window. The ShowDrawing statement then redisplays the Drawing window in its new coordinate position, where it occupies virtually the whole screen.

Drawing the Picture

From this point, drawing the picture is very straightforward. We use a bunch of FrameRect commands to create the table and the computer, draw a floor under the table, and hang an empty picture frame on the wall behind the Mac. When you ran the program, you saw how quickly it could do all of these things. Equally important, it draws more precisely than even a fairly good artist can do with MacPaint, because the beginning and ending points of each line and the boundaries of each form are set in the program.

Returning to Normal

The end of the program contains a statement using the command PenNormal. This command restores the pen to "normal" condition, which is size (1,1) and color black. Whenever you change a standard value in a program, it's good programming practice to return to normal settings before ending the program, so that the results of subsequent programs will be predictable.

Adding Your Personality

If you're as excited about Mac graphics as we hope you are by now, your head should be brimming with ideas for enhancing and improving MacStillLife. Put some words or some Macasso-like artwork in the picture frame. Draw a textured right wall to give a feeling of three-dimensionality to the drawing. Move the Mac. Add a printer . . . a disk drive . . . a mouse and tail. The possibilities are endless.

We've only scratched the surface of Macintosh Pascal graphics in this chapter. In Chapter 14, "Advanced Mac Graphics", we'll take a look at a few more of the powerful routines for graphic design in Mac Pascal. The Macintosh Pascal *Reference Manual* provides a complete list and explanation of the impressive array of graphics commands available on the Mac.

Summary

We've explored a lot of artistic territory in this chapter, and you've learned a great deal about ways of using Macintosh Pascal to program QuickDraw Graphics for exciting effects.

You've learned to use the MoveTo and LineTo commands to draw straight lines. You've learned to use the PenSize and PenPat commands to change the size and pattern of the graphics "pen," and you can use the PenNormal command to put the pen back to its default settings when you're through.

You've learned how to generate random numbers with Random and use them in your art. You've also learned how to set limits on such numbers so they'll do what *you* want them to do, not whatever weird thing comes into their randomized little heads.

You've learned how to Frame, Paint, Fill, Erase, and Invert ovals, rectangles, and round-cornered rectangles (known as Oval, Rect, and RoundRect to their Macintosh Pascal friends). You've seen programs that let the Mac combine these shapes and activities in various interesting and sometimes startling ways.

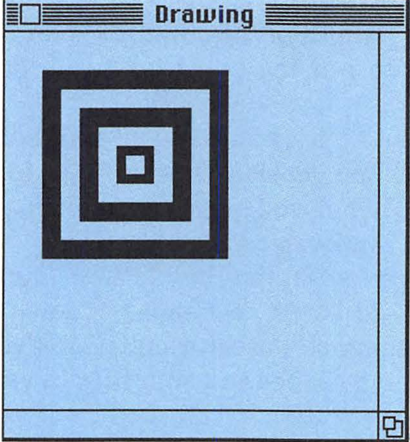
You've learned to add text to the graphics in the Drawing window and determine the text's size and appearance using TextFont, TextSize, and DrawString statements.

You've learned that Rect (rectangle) can be a variable type. You've learned how to use it in the SetRect statement to set up a rectangle in a particular place. You've learned how to apply SetRect to a special kind of rectangle, the Drawing window, and to use this statement along with others involving the HideAll, ShowRect, SetDrawingRect, and ShowDrawing commands to change the size and location of the Drawing window.

By now you're well on your way to being an artist with Macintosh Pascal!

Mac-r-cises

1. Write a program that paints an oval shape, draws a smaller oval shape inside of the larger one, and then causes the smaller oval to appear and disappear, giving the impression of a "talking" oval. (Later you might want to enhance this with things like eyes, perhaps even eyes that blink, and cute sayings that appear in the Text window when the mouth moves. But for your Mac-r-cise, just do the basic program.)
2. Create a series of nested rectangles that alternate between black and white as they go from large to progressively smaller ones, something like this:



   Hint: this is not as difficult as it may seem at first, but it is a good example of the need for prior planning that we've mentioned before.
3. (Difficult) Write a program that positions a small black ball at the top of the Drawing window, drops it straight to the bottom of the window, and bounces it back up to where it started.

7

Some Fun Things to Do with Letters and Words

When you finish this chapter, you'll know

- **What a built-in procedure in Pascal is**
- **How to get information about string variables**
- **How to combine two or more strings**
- **How to extract information from strings**
- **How to change strings by deletion and insertion**

One thing that has made Pascal an important programming language is the easy way it lets its users (that's us) manipulate information or data. In the next two chapters, we'll look at some of these ways of handling data, beginning with alphanumeric data in this chapter and moving to numerical data in Chapter 8. These two chapters will also teach you about built-in "procedures" and "functions".

What's a String Again?

Many programming tasks involve manipulation of *strings*. Strings, in computer jargon, can be characters, letters, words, sentences, paragraphs, or even (theoretically) whole articles or books. When we use a word processing program like MacWrite, for example, we are manipulating strings. Similarly, building a file of names and addresses for later recall, sorting names and addresses, label printing, and similar common business activities involve letters and words, and therefore use string manipulation.

Just to refresh your memory, let's review the characteristics of a string in Macintosh Pascal. First, a string is a variable declared to be of type String and, optionally, given a maximum length in number of characters. The value of a string variable is the string's current contents. A string can also be a constant defined to be a string by virtue of the fact that its contents are enclosed inside single quotation marks.

Second, Pascal always treats the contents of a string as letters (or at least non-numbers), even if they sometimes look like numbers to us. Thus, a string can have the value "abcdefghijklmnopqrstuvwxyz" or the value "0123456789" or even "a0984bvcwbxtk99". But Pascal could not multiply a string by 2, for example, even if the string contained only numbers. Thus, if a string variable called "string1" contains the value "234", trying to say something like Writeln(2*string1); would be as meaningless as saying, in English (as opposed to Pascal), "Two times doorknob".

The elements of a string are usually called "characters" or sometimes "alphanumeric characters" to indicate that they can be letters, numbers, or other symbols (like punctuation marks).

Types of String Manipulation Procedures

A number of procedures for handling strings are built into Macintosh Pascal, and we'll take a look at all of them in this chapter. They can be subdivided into the following categories:

1. Procedures for getting information about strings and their contents
2. Procedures that compare strings
3. Procedures that create new strings from old strings

Information-Gathering Procedures

Procedures to gather information from a string include procedures for finding the length of a string and for picking out part of a string. Let's look at length first.

Finding the Length of a String

We quite often want to know how many characters a string contains (so that we can, for example, tell a program to print the string and make it appear centered on a page or screen). How can we find out this information

— that is, the length of the string? This turns out not to be as easy as it might seem at first glance. Fortunately, Macintosh Pascal has provided us with a built-in procedure for doing this job. Logically enough, the procedure is named Length. Its use is shown in the following example:

```
program StringLength;
  var
    a:string;
begin
  Writeln('Input a string: ');
  Readln(a);
  Writeln('That string has ',length(a),' characters in it.');
end.
```

Type in the program, Save it, and Run it, entering strings of various sizes and checking to satisfy yourself that the procedure called Length does in fact calculate the length of any string entered and faithfully report the number. Note that the Length procedure counts everything enclosed between the single quotes in a string or, alternately, everything typed into a Readln statement, *including spaces*. This is likely to be confusing at first. If someone told you to count the number of letters in the sentence "Go count the number of letters in that sentence", you'd probably come up with 39, but to Macintosh Pascal, the Length of that sentence is 47. In fact, the Length procedure counts *every* character — letter, number, punctuation mark, space, or other symbol — when it evaluates a string.

Getting the String to the Procedure

Notice that the Length procedure requires the word *length* to be followed by a left parenthesis, the string being evaluated, and then a right parenthesis. Virtually all procedures — whether built-in or created by programmers — require one or more values in order to run. These values are called the *parameters* of the procedure. The process of getting them to the procedure from the program is referred to as *passing* the parameters.

The parameters are enclosed in parentheses following the name of the procedure.

Leaving out the parameters that a procedure expects can result in all kinds of trouble. For instance, if the Macintosh saw a line of Pascal code that said

```
Writeln ('The length is ' Length);
```

it wouldn't know what the word *Length* meant. Macintosh Pascal will not recognize this word as the name of a built-in procedure, because the procedure it knows by that name is always associated with a parameter. Instead, it will assume that a *variable* named Length is meant. If in fact you've defined such a variable, the program will simply print its current value here and proceed merrily along. If you haven't, you'll get an error message and the program will stop in its tracks.

The parameter passed to any built-in function used in this chapter must be of type String. (This is not necessarily true of procedures we'll find in later chapters.) The passage of any other type of parameter to the Length procedure results in an error announced by a Bug box that looks like this:

An incompatibility between types has been found.

Finding a String in a String

We frequently want to find out whether a string contains a certain set of characters within it. For example, if we are looking through a file for the name Aron Nimzovich, but we suspect that this name may be spelled wrong (not too remote a possibility), we might want the program to find any name containing the letters *Nim*. This would probably be the person we're looking for.

The built-in Macintosh Pascal procedure for finding a string inside another string is called Pos (for position). As the name implies, Pos tells both *whether* a string exists inside another string and, if it does, *where* it is inside the larger string (in other words, its position).

Using the Pos Procedure

The following sample program will ask for two strings and then tell where, if at all, the first string can be found inside the second. Type it in, Save it, Run it with strings of varying lengths up to the maximum of 255, and observe the results carefully. Try using a one-character string as the string to be located and (separately) as the string in which the other is to be located. What is the result?

```
program FindString;
  var
    a,b:string;
begin
  Writeln('Input the base string: ');        ← Not "bass" string!
  Readln(a);
  Writeln('What string should I find? ');
  Readln(b);
  Writeln('The second string,', b,'is at position ',Pos (b,a),'.');
end.
```

Double Parameter Passing

The Pos procedure requires *two* parameters, the first containing the string to be found and the second containing the string in which the first string is to be located. Many Macintosh Pascal procedures require multiple parameters; when that's the case, the parameters usually must be passed to the procedure in a prescribed, definite order. Each parameter is separated from the rest of the statement by a comma. The program you just typed, for example, told Pos to find the string designated by variable b inside the string designated by variable a.

What happens if you tell Pos to look for a string that does not occur *exactly* as you list it — say, it has a lowercase letter where you listed a capital? Try it. You should get a response of "0", which means that the string isn't present. In this respect, as in many others, computers are very literal-minded.

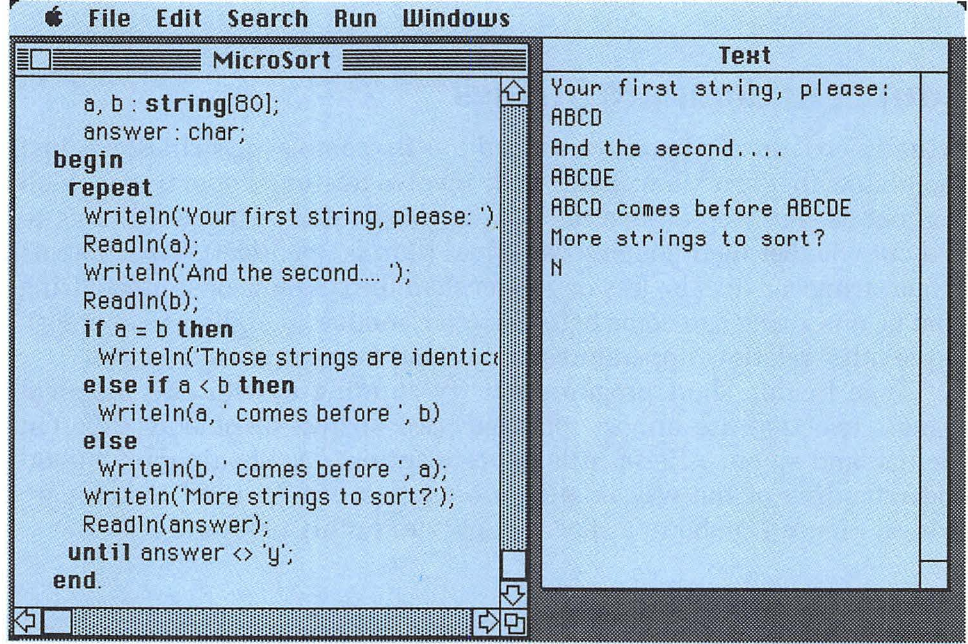
Procedures to Compare Strings

Actually, you already know the procedures for comparing strings; you just don't know that you know them. They involve relational operators, which you met earlier. We can use the relational operator = on two strings to find out whether their contents are equal (that is, identical). The contents of one string can also be less or greater than the contents of another string (that is, one string can come before or after another in alphabetical order), so the other relational operators can be used for comparison as well.

Type in this short program and try running it with two identical strings, two that are almost identical, two strings of notably different lengths, and so on. After a little experimenting, you should gain a good understanding of the way relational operators can be used to compare strings. Figure 7-1 shows a short sample run of this program.

```
program MicroSort;
 var
  a,b:string;
  answer:char;
begin
 repeat
  Writeln('Your first string, please: ');
  Readln(a);
  Writeln('And the second.... ');
  Readln(b);
  if a = b then
    Writeln('Those strings are identical!')
  else if a<b then
    Writeln(a,' comes before ',b)
  else
    Writeln(b,' comes before ',a);
  Writeln('More strings to sort?');
  Readln(answer);
 until answer<>'y';
end.
```

Figure 7-1. Sample Run of MicroSort Program



New Strings from Old

There are several ways to make a new string from one or more old strings, and Pascal has a built-in procedure for each of them. We can:

1. stick two strings together;
2. extract a piece of one string to form another;
3. delete characters from a string;
4. insert new characters into a string;
5. or various combinations of the above.

String manipulation procedures can accomplish some very complex things. Word processor functions such as search-and-replace work by combining such procedures (either built-in or created by the program) in various ways.

Combining Strings

To combine two (or more) strings in Macintosh Pascal, we use a procedure called Concat (the name comes from *concatenation*, the technical term for hooking two or more things together). The following small program demonstrates Concat.

```
program StickStrings;
 var
  a,b:string;
  answer:char;
begin
 a:=' ';        ← Note blank space between single quotes
 repeat
  Writeln('Enter a string, please: ');
  Readln(b);
  a:=Concat(a,b);
  Writeln('So far we have: ',a);
 until b='*';
end.
```

The program keeps adding new strings to the one created by adding together other strings until the user types an asterisk *or* until the 255-character limit imposed on the length of strings by Macintosh Pascal is exceeded.

String concatenation can be used to build meaningful messages, as the following example shows. It also demonstrates how to insert punctuation into collections of items as they are gathered and printed. (A For loop

could get rid of the extra comma dangling at the end — can you figure out how?)

```
program SelfImage;
var
a,trait:string[80];
n:integer;
begin
 Writel('Give me three words that describe things you like about
 yourself.');
 for n: = 1 to 3 do
  begin
   Write('Trait #',n,':');
   Readln(trait);
   a: = Concat(a,trait);
   a: = Concat(a,', ');         ← Adds comma and blank after each trait
 end;
 Writeln('Aren''t you glad that you''re ',a,'?');
end.
```

Figure 7-2 shows a sample run of this program (with the author's customary modesty displayed for all to see).

Extracting Sections of a String

The built-in Macintosh Pascal procedure called Copy takes one or more consecutive characters from one string and places them into another. The Copy function needs to be told three things in the parameters passed to it:

1. The name of the string to be used (or the string itself, enclosed in single quotation marks);
2. How many characters into the string to begin the copy (that is, the starting point of the copy);
3. How many characters from that point to copy (the length of the string to be copied).

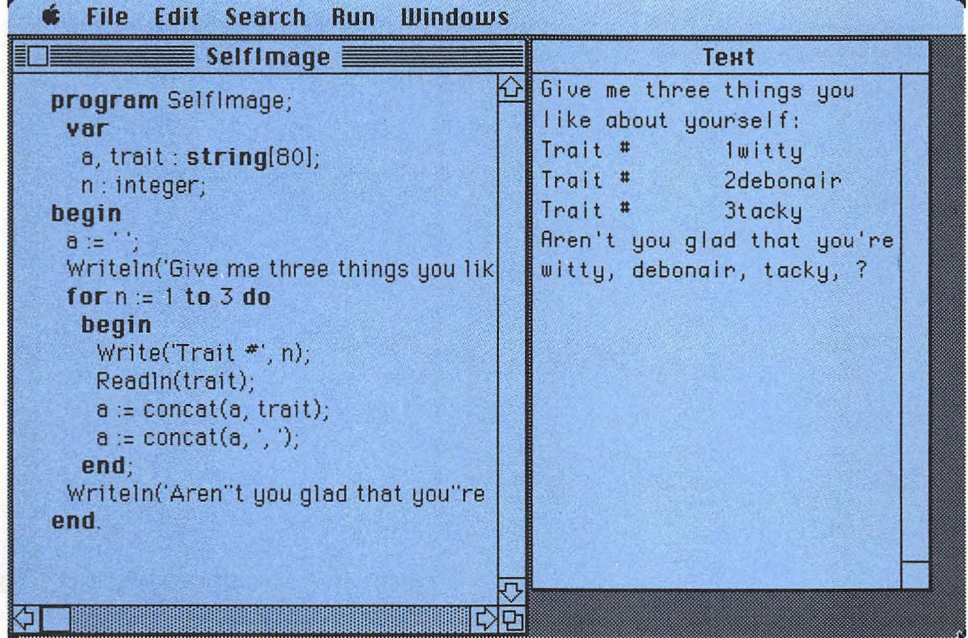
Given this information, Copy will *copy* the designated characters in the designated string and do with them whatever the Pascal statement containing copy directs (for example, assign them to a variable, print them, or concatenate them to some other variable).

The following sample program demonstrates two important principles in manipulating strings with Macintosh Pascal: the use of the Pos procedure to find the place in a string where something is to be done, and the fact that Copying from a string leaves the original string untouched.

```
program Extract1;
  var
    a,b:string;
    n:integer;
  begin
    a: = 'The Macintosh is a superb computer.';
    n: = pos('Mac',a);        ← Locates the start of the string 'Mac'
    b: = copy(a,n,3);         ← Takes three characters beginning at nth position
    Writeln('Let''s hear it for ',b,'!');
    Writeln(a);
end.
```

There's no need to save this program, but run it once and then look at the two strings involved: a and b. When the variable a is initialized to contain the sentence, "The Macintosh is a superb computer." and the Pos procedure is used, the variable n is set to the value 5, because the string "Mac" begins at the fifth position of the string. Then Copy takes the fifth, sixth, and seventh letters of string a and puts them into string b. The last line of the program repeats the value in a, just to prove that we haven't altered its contents, by making a copy of some of its characters.

Figure 7-2. Modesty and the SelfImage Program

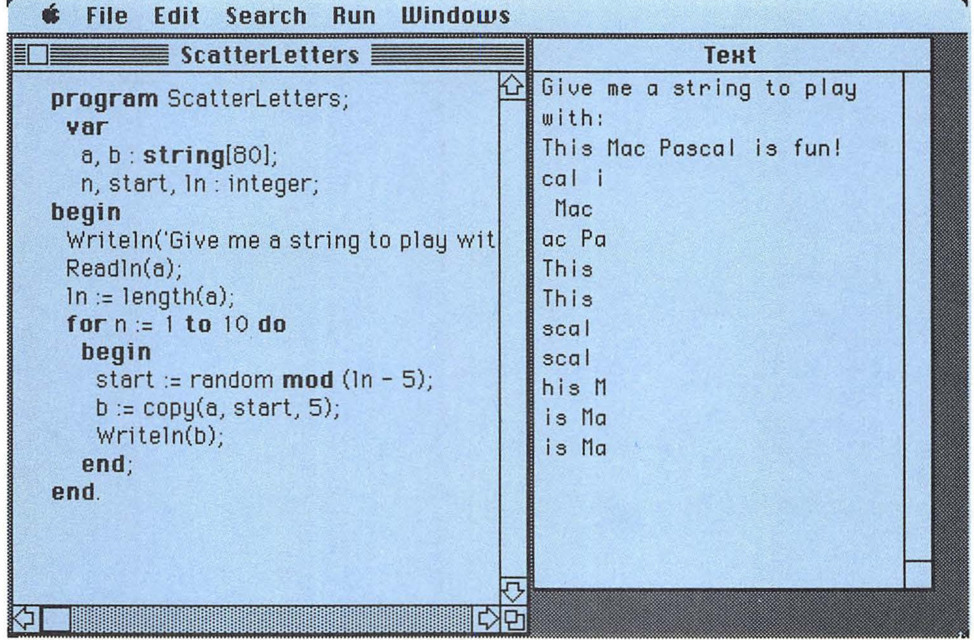


Breaking a String into Bits

The next sample program plays havoc with a string you type in, separating it into ten blocks of five characters each, completely at random. The effect can be interesting. Figure 7-3 shows a sample run of the program.

```
program ScatterLetters;
 var
  a,b:string;
  n,start,ln:integer;
 begin
  Writeln('Give me a string to play with: ');
  Readln(a);
  ln: = length(a);        ← To be sure we don't go too far!
  for n: = 1 to 10 do
   begin
    start: = random mod (ln-5)      ← Picks a random starting place
    b: = copy(a,start,5);
    Writeln(b);
  end;
end.
```

Figure 7-3. Sample Run of ScatterLetters Program



From here, we bet you'd find it easy to come up with a general Copy program that asks the user for a string to work with, a start position, and an end position, checks to be sure everything's in order, and then prints the resulting copied string.

Getting Rid of Part of a String: Omit and Delete

Sometimes you'll decide that a sentence you've typed into your Mac contains unwanted letters or words, either because of a typographical error or because you've decided to edit something out of a passage to make it sound better. To get rid of the extra materials, you could use the functions you've already learned, following these simple steps:

1. Use Pos to find the string to be deleted within the passage you're working with.
2. Use Copy to extract the characters *up to* the position of the characters to be deleted.
3. Use Copy to extract the characters *following* the string to be removed.
4. Use Concat to combine the two strings created with Copy.

Whew! If we had to do that every time we wanted to delete some characters from a string, we'd leave the dumb things there! Fortunately, Macintosh Pascal has a built-in procedure called Omit that creates a string made up of the string it is working on *minus* the characters we specify by starting position and length. Omit requires the same three parameters as Copy: string or string name, starting position, and length of string to delete.

Let's take a look at a program that looks for the word *the* in a sentence and deletes it along with its following space.

```
program CatchTheThe;
 var
  a,b:string;
  n:integer;
 begin
  n:=1;
  Writeln('Give me the letter, please.');
  Readln(a);
  while n<>0 do
```

```
    begin
      n: = pos('the',a);
      if n<>0 then
      a: = omit(a,n,4);
    end;
  Writeln('This is what I sent: ');
  Writeln(a);
end.
```

The program isn't foolproof. If, for example, a "the" in the sentence begins with a capital T, the program will leave the word alone. Similarly, a "the" followed by punctuation would cause the word and punctuation to be deleted but leave the following space(s) intact.

As it is written, the program won't differentiate between the *word* "the" and the combination of letters "the" as they appear in the words theater and leather. In fact, in these latter cases, the program would delete four characters and leave us with the words "ter" and "lea" — clearly not what we intended. We could get around that problem by using the four-character combination "the " (with a blank at the end) to search for, but even *that* solution won't catch the word "the" followed by punctuation and would have an undesirable result on words like "absinthe" which would become "absin". As you can see, being precise about such editing is not an easy task and is one of the reasons word processing programs are so large and complicated.

One final point. Omit, like the other string commands we've discussed, is really a function rather than a procedure. (We'll explain the difference between the two in Chapter 9.) Macintosh Pascal also has a procedure which does the same thing as Omit. It's called Delete, and it's used exactly the same way as Omit except that it doesn't return a result without being instructed to do so. You'll see what this distinction means in Chapter 9.

Inserting New Material

The other half of editing — that grubby but necessary job that follows writing — is adding words or letters that somehow got left out the first time around. Again, it would be *possible* to use the Pos, Copy, and Concat functions to insert material, but the process would be very cumbersome. Instead, Macintosh Pascal offers a function called Include and a corresponding procedure called Insert. We'll concentrate on Include because it is easier to work with.

Like Copy and Omit, Include requires three parameters. The three parameters differ slightly from those required for the other two functions,

however. Include needs to know the string to work with, the string (or name of string) to insert, and the position in the existing string at which the new information is to be placed. It doesn't need to know the *length* of the insertion because it simply inserts the whole second string into the designated position.

Combining Omit and Include to Replace

Let's combine Omit and Include into a useful little program. The program will ask for a string to work with, a string to search for, and a string to replace it with, after which it will print the modified string.

```
program ReplaceIt;
var
   sentence,right,wrong:string;
   n:integer;
begin
  n: = 1;
  Writeln('What''s the beginning string?');
  Readln(sentence);
  Writeln('What to find?');
  Readln(wrong);
  Writeln('Replace with what?');
  Readln(right);
  while n<>0 do
    begin
      n: = pos(wrong,sentence);
      if n<>0 then
        begin
          sentence: = omit(sentence,n,length(wrong));        ← Note parens!
          sentence: = include(right,sentence,n);
        end;
      end;
    Writeln('The changed sentence reads as follows:');
    Writeln(sentence);
end.
```

Type the program into your Mac, Save it, and Run it a few times. Try different combinations. You'll probably discover that if you put in a string that occurs both as a stand-alone word and as a combination of letters within a word, both occurrences are changed — as happens in Figure 7-4, where the *is* in *This* was replaced inadvertently. To avoid this problem, begin and end your answer to "What to find?" with a blank (space). If you do that, the program will replace your requested word only when it finds it as a stand-alone word.

Problems with ReplaceIt

Running the ReplaceIt program produces some other problems that probably wouldn't be obvious to you at this stage of your Macintosh Pascal adventures. A quick summary of the major problems *we* found when testing the program follows:

1. Entering a string to be found that is longer than the string in which we are searching can cause the system to take a sudden vacation. This can be remedied only by resetting the system with the Programmer's Key.
2. Entering a string longer than 255 characters *or* a replacement string that, when combined with the modified original sentence, is longer than 255 characters produces an error condition from which recovery is self-explanatory.

Figure 7-4. Inadvertent Replacement of a String

```
File  Edit  Search  Run  Windows

ReplaceIt
  Writeln('What''s the beginning string'
  Readln(sentence);
  Writeln('What to find?');
  Readln(wrong);
  Writeln('Replace with what?');
  Readln(right);
  while n <> 0 do
   begin
    n := pos(wrong, sentence);
    if n <> 0 then
     begin
      sentence := omit(sentence, n, le
      sentence := include(right, sente
     end;
   end;
  Writeln('The changed sentence reads
  Writeln(sentence);
 end.
```

```
Text
What's the beginning
string?
This is a test.
What to find?
is
Replace with what?
was
The changed sentence reads
as follows:
Thwas was a test.
```

One Other Way to Access Part of a String

Our discussion of strings would be less than complete if we didn't mention in passing one more way to get information out of a string. This idea will require you to accept an idea which we will not fully explain until Chapter 10.

A string can be thought of as a special type of *array* (there's the idea we mentioned). We can get information out of an array by using what is referred to as "indexing". For example, if we have the string: "Mac Pascal is a super language!" assigned to the variable Saying, and we want to find out what the tenth character of the string is, we can write a line of Pascal code like this:

```
newstring:=saying[10];
```

The number 10 in square brackets tells Macintosh Pascal to extract the tenth character from the string array whose name we have given. In this case, the tenth character — which happens to be the letter *l* — is assigned to a variable called NewString, which must, obviously, be of type String or Char. Indexing is not as powerful or flexible as the Copy function, but there will be times when it is the best way to handle a string extraction or comparison.

Summary

In this chapter, you've learned a lot of things to do with strings besides play cat's cradle with them! By now you know how to use the following built-in Macintosh Pascal procedures and functions with strings:

Length to find out how long a string is
Pos to locate a particular set of letters within a string
Concat to stick two strings together
Copy to extract characters from a string
Omit to delete characters from a string
Include to insert characters into a string
Indexing with brackets to extract a sub-string

You also learned how to combine some of these functions and procedures in useful programs.

The next chapter will demonstrate built-in Mac Pascal procedures and functions designed to work with numbers rather than strings.

Mac-r-cises

1. Rewrite the CatchTheThe program so that it asks what word you'd like to delete and then deletes *all* occurrences of that word.
2. A palindrome is a word or phrase that reads the same forward and backward. One of the most famous palindromes is "Madam, I'm Adam." (Note that punctuation is ignored in deciding whether a particular string is a palindrome.) Write a program that accepts any string as input and produces a palindrome by attaching to the end of the string that has been entered the same string in reverse order. (For the moment, don't be concerned with punctuation.)
3. Now write a program that accepts a string as input and, ignoring punctuation, tells whether the string is a palindrome or not.
4. (Difficult) Write a program that accepts a character, adds it to a word, continues building that word until it gets to be 80 characters long, prints the word at that point, and then begins accepting input again. (Hint: The program will need to know when to end the process of accepting letters.)

8

Can Numbers Be So Much Fun?

When you finish this chapter, you'll know

- **How to do math seven ways in Pascal**
- **How to determine whether a number is odd or even**
- **More about the way random numbers work**

The last chapter focused on the built-in procedures and functions in Macintosh Pascal that enable us to manipulate string data. This chapter will look at these procedures' close relatives, the built-in mathematical procedures and functions in Mac Pascal. We are trying hard to avoid too much emphasis on mathematics in this book, but we would be doing the Pascal language an injustice if we didn't spend some time on the powerful number-handling methods it makes available.

This chapter will make extensive use of the Instant window, which you met earlier. You can execute all math functions there and see the results immediately.

Types of Numbers in Pascal

Before digging into mathematical procedures and functions, let's get a handle on the various types of number representations available in Macintosh Pascal. This is important because some procedures and functions in this chapter will work only on variables or values of a specific type or types. Others return results only of a specific type. You need to know what the results mean in order to perform further calculations with them.

Integers

Integers are old friends by now. As you probably remember, an integer variable can hold any value from –32767 to +32767. Its value must be a *whole* number; it cannot include any decimals or fractional parts. The following are all valid integers in Macintosh Pascal: 42, 18943, 2, 0, 32766, –11, –32, –19765. On the other hand, these are not: 3.1415, –11.11, 3., 98.6, –2001.1, 11., and 16e5. (If that last item looks a little confusing, please be patient for a few moments until we get to the *Real* numbers.)

All the examples were probably clear except for the value "11." — which *cannot* be represented using integer variables. It appears to be a whole number, doesn't it? The problem is, Mac Pascal treats a number with a decimal point in a special way, even if nothing or a zero (or zeros) follows it. The decimal itself creates a fractional part.

Real Numbers

You may also remember "real" numbers. In Macintosh Pascal, a real number contains a decimal point with at least one digit before and one digit after it. In other words, a real number contains a fractional part. Because of the way numbers are stored in the Mac's memory, a real number can hold values much larger or smaller than an integer can hold. To understand the range of values involved, though, we need to take a side excursion into the subject of exponential notation.

Numbers with Three Parts

Numbers declared to be "real" are stored in the Mac's memory in such a way that they are displayed on the screen in three parts, as shown in Figure 8-1. (We'll explain the meaning of "mantissa" and "exponent" in a few moments.)

To see a few real numbers displayed on the screen, so you can get used to their appearance, type in this simple program and Run it. (No need to Save it, but we will use it once more, so don't erase it.)

Figure 8-1. How Real Numbers Are Stored and Displayed

6 e + 4

Mantissa Exponent

```
program ShowReals;
var
  x:Real;
begin
  Writeln('Gimme a number: ');
  Readln(x);
  Writeln(x);
end.
```

Try running the program with the data in Column 1 of Table 8-1 as test information. The Mac screen should show the number as in the second column of the table. The third column tells the full way to say the number as displayed.

The Moving Decimal

Now that we've looked at some examples, let's find out what they mean. Look at number 111.0 first. When displayed as a type Real variable, it appears as 1.1e+2. Its *mantissa* is 1.1 and its *exponent* is 2. This way of displaying numbers is sometimes called "scientific notation". We can quickly determine the value of a number displayed in scientific notation by moving the decimal point inside the mantissa the number of positions indicated by the exponent. Here, 111.0 displays as 1.1e+2. If we take 1.1 and move the decimal point two places to the right (+2) we'll get 110. (The 111.0 became 110 because of rounding.) Similarly, 3.4e+4 results in a value of 34000 after we move the decimal point four places (+4) to the right.

Numbers smaller than one are handled similarly, except that the exponent is negative and the decimal point gets moved to the left. Thus,

Table 8-1. Sample Data for ShowReals Program

Enter this:	*See this:*	*Say this:*
1	1.0e+0	One times 10 to the zero power
3.	3.0e+0	Three times 10 to the zero power
111.0	1.1e+2	1.1 times 10 squared
1234.56	1.2e+3	1.2 times 10 cubed
33999	3.4e+4	3.4 times 10 to the fourth
.045	4.5e−2	4.5 times 10 to the minus two

the value .045 from Table 8-1 is expressed in scientific notation as 4.5e − 2. If we move the decimal point two places to the left (−2) of its present position, we restore the original value of .045.

Other Types

The vast majority of numbers likely to be involved in your Pascal programming on the Mac will fit into the data types Integer and Real. Macintosh Pascal does, however, make other data types available. Here are a few: Longint is a variation of Integer with a wider range of values; Double is variation of Real with a wider range of precision; and Extended goes one step further in terms of the precision of Real values that it can represent.

All these can be useful if an application, for example, calls for great precision in very large or small numbers. On the whole, though, they're too esoteric for our purposes, and we'll ignore them for the rest of this book. To learn more about them, check your *Macintosh Pascal Reference Manual*.

Displaying Numbers

You may have noticed that we seem to have a dilemma when it comes to dealing with numbers in Macintosh Pascal. If we use Integer variables and values, we are limited to whole numbers — fine for counting things like the number of chairs in the room, the number of Macintosh computers you'd like to own, and so on, but nearly worthless for tasks like averaging numbers, measuring weights or distances, or dealing with money. On the other hand, if we use Real values, we seem to be stuck with strange-looking numbers and a lack of precision in the display to boot. If we are paid, say, $8.73 per hour, Mac Pascal would translate that to 8.7e + 0. Not only is this result funny-looking, it's also apparently wrong; we're being robbed of three cents an hour!

The problem turns out to be in the way Mac Pascal *displays* numbers rather than in the way it *remembers* them.

That's More Like It!

Go back to the Mac and find the line in the program ShowReals that displays the value of x. Just below that line, add the following new line:

```
Writeln (x:3:2);
```

Now Run the program again, using data from Table 8-1 and adding some of your own. The program will first display numbers in the scientific notation we saw before, but below that it will display them with a neatly placed decimal point — the way we *expect* a well-behaved computer to do!

Size and Decimals

We can control the format of the display of numerical value on the Mac screen by adding two values to a Write or Writeln statement. Each value is preceded by a colon. The first value tells Macintosh Pascal the *minimum* number of characters to display for any number. The number must be at least 1, but it can be quite large. The second value tells Mac Pascal how many digits to put after the decimal point. This second value is optional. If it's left out, Mac Pascal does not put in any decimal points, which is what we want when we format integers for output but not usually what we want when we format real numbers for output.

Experiment with different values by changing the numbers following the colons in the program ShowReals. You'll discover that changing the first value affects little except the positioning of the number inside the Text window, while altering the number of decimal digits to display affects the way the value appears. Note, by the way, that Mac Pascal rounds numbers as needed: the value 4.939 displayed with two decimal places comes out 4.94, not 4.93.

Getting Information about Numbers

In the last chapter, you saw that Macintosh Pascal includes built-in functions and procedures that enable us to examine a string and extract information about it. Mac Pascal offers other built-in functions that do the same kind of thing with numbers. Two of the most useful are the functions Odd and Abs. Let's take a look at these.

Odds Are

Set up the Instant window and type in the following line of Mac Pascal code:

```
Writeln(odd (3));
```

Now click on the Do It button and see what happens in the Text window. If all goes as expected, the word *True* will appear there. Change the number to any other *integer* and Do It again.

The Odd function will return a "True" result if the number given as a parameter is odd; otherwise it returns "False". As you may remember, values that return the results true and false are called *boolean*. Because it returns a boolean value, we can use the Odd function in a program statement or expression that does conditional processing.

Are You ABSolutely Sure?

Another characteristic of numbers that Mac Pascal can reveal is their *absolute* value. The easiest way to understand the function that does this, Abs, is to work with it a little. Clear the Instant window and type in the following:

```
Writeln(abs(56));
Writeln(abs(-56));
```

Now click on the Do It button and observe the Text window. The Abs function gave the same result for both 56 and –56 because its purpose is to cause the program to ignore the sign in front of a number and return the number's *absolute* value. The value is called "absolute" because it is not relative to a particular reference point at which positive and negative numbers are separated from one another.

Creating Numbers at Random

In Chapter 6, we experimented with the Random function in creating artwork to demonstrate Mac's QuickDraw Graphics routines. We used a statement like this

```
Start: = random mod 200;
```

to generate a random number between 0 and 199. Take another look at Random and Mod and try to figure out *why* this statement works as it does.

Random Alone

By itself, the Random function will produce a value between −32767 and +32767, the range of valid integers in Macintosh Pascal. You can confirm this by using the Instant window and clicking on the Do It button a few times or by writing a small program with a For loop to generate random numbers.

When a random number in that range is useful in a program, you can simply write a statement like this:

```
X: = random;
```

or display a random number using Write or Writeln, like this:

```
Writeln(random)
```

Most of the time, however, programs need a random number within a smaller range. That's where Random's stylish friend Mod comes in.

Random with Mod

The Mod function by itself produces the remainder of a division problem. To see this on the Mac, open the Instant window again, clear it if necessary, and type in the following lines:

```
Writeln(87 mod 9);
Writeln(672 mod 6);
```

Now click on the Do It button in the Instant window, and you'll see the numbers 6 and 0 in the Text window. That's because 87 divided by 9 produces 9 with a remainder of 6, and dividing 672 by 6 produces 112 with no remainder. Note that Mod *always* produces positive values.

Now let's combine Random, which generates a number between −32767 and +32767, with the Mod operator, which creates a remainder. Suppose we want to generate a number between zero and 5. Random gives us the number 21398. If we divide that by 5, we get 4279 with a remainder of 3. Thus the statement

```
x: = random mod 5;
```

in this case gives x the value 3. This approach will always produce a number *less than* the number following the Mod function because the

remainder of a division problem must always be less than the divisor. If we want a range of values from 1 to 5 instead of 0 to 4, we just add 1 to the result of the Random mod combination.

Let's Check the Odds

Another example program can bring together most of what we've discussed so far. Type in the OddCheck program that follows and Save it on your Mac disk. Then run it a couple of times before going on with the discussion.

```
program OddCheck;
 var
  x,range,count,numodd,numeven:integer;
begin
 Numodd: = 0;
 Numeven: = 0;
 Writeln('This program generates numbers');
 Writeln('at random in a range you specify');
 Writeln('and keeps track of how many odd and');
 Writeln('how many even numbers are created.');
 Writeln;
 Writeln('What is the highest number you want?');
 Readln(range);
 Writeln('How many numbers should be generated?');
 Readln(count);
 for x: = 1 to Count do
   begin
    if (Odd (random mod range))then
     Numodd: = numodd + 1
    else
     Numeven: = numeven + 1;
 end;
 Writeln('I got ',Numodd :1 , ' odd numbers');
 Writeln('and ',Numeven :1 , ' even numbers.');
end.
```

As indicated in the instructions that will print in the Mac's Text window, this program generates random numbers and keeps track of the totals of even and odd numbers it creates. Run the program a few times using very large Count values, and you'll notice that the distribution of odd and even numbers tends to be about evenly divided. That's a sign that Random generates truly random numbers. In theory, if we create enough such numbers, we should find out that odd and even numbers are

each generated exactly half of the time. Given the imprecision of such things, though, the ratio will always be a little off in one direction or another. The fewer numbers we ask the program to create, the less likely we are to see a 50-50 split.

Calculating with Numbers

We can perform a wide range of calculations with numbers in Macintosh Pascal, using no less than fifteen predefined functions and procedures. The remainder of this chapter will take a look at just a few.

Squares and Roots

Do you remember struggling through the complicated process of manually calculating square roots in high school math class? If so, you'll be delighted to know that Mac Pascal calculates square roots in a simple one-step process. To see this work, pick a number you'd like to know the square root of. (Even if you can't think why on earth you'd want to know the square root of any number, humor us for a moment and think of a number anyway.) Now put this statement in the Instant window

```
Writeln(sqrt(53):1:3);
```

using *your* number in place of the 53. Now Do It and observe the result. Voila! Instant square roots.

Mac Pascal also offers a companion function to Sqrt. It's called Sqr, and it will *square* (multiply by itself) any number the program gives it. To test Sqr, type the following in the Instant window, using the *answer* from your last run using the Sqrt function. Do It and see if the result matches the number you started with.

```
Writeln(sqr(7.28):1 :3);
```

Squaring the square root of a number should give you the number you started with. However, there will probably be a slight discrepancy. For example, we used 53 for our first Sqrt problem and got an answer of 7.280. But when we used 7.280 as the number to be squared by the Sqr function, the answer came up 52.998. A rounding-off process takes place when we display a number using the colon values (:1 :3) which clearly reduces the precision of the final answer. The square root of 53, carried to eight decimal places instead of three as we ordered it, is 7.28010989.

In fact, the Macintosh stores numbers with such great precision that if one statement takes the square root of a number and another statement squares the answer, the exact number will be returned every time. Try the following in the Instant window, and you'll see what we mean:

```
Writeln(sqr(sqrt(53)) :1 :3);
```

Arithmetic Operators

Remember the four arithmetic operators used by Macintosh Pascal — + (addition), – (subtraction), / (division), and * (multiplication)? To use them properly, you need to understand *precedence*.

Precedence

Precedence in Macintosh Pascal can be illustrated by a simple problem. Suppose you want to count the number of tomato plants you plan to put in your garden this season. You have eleven flats of tomatoes, each containing twelve plants. A generous neighbor then gives you seven more flats. On a calculator, you'd punch the 11, the + key, the 7 key, the * key, and finally the 12 to find out how many tomato plants you now have. But that approach in Mac Pascal will produce an incorrect answer. To prove it, use the Instant window again:

```
Writeln(11 + 7*12);
```

Do It to this expression and you'll get the answer 95, not 216 as expected. Why did that happen?

Mac Pascal needs to know the order of mathematical operations in a string of arithmetic like the one we just gave it. It always performs multiplication and division *before* addition and subtraction. (It will perform multiplication and division, as encountered, from left to right.)

You may also need to know someday that arithmetic functions have precedence over logical operators (<, >, <>, =, <= and >=).

Math Quiz Program

Now let's put what you've been learning in this chapter to practical use in a multiplication table quiz program. The program listing is given below. Type it into the Program window, Check the syntax, Save it on the disk, and then Run it.

```
program MultTables;
 var
  Howmany,bignum,x,factor1,factor2:integer;
  Ans:integer;
  Name:string;
 begin
  Writeln('What''s your name?');
  Readln(name);
  Writeln('How many problems, ',name,'?');
  Readln(howmany);
  Writeln('What''s the biggest number');
  Writeln('I should use?');
  Readln(bignum);
  for x: = 1 to howmany do
     begin
      Factor1: = (Random mod Bignum);
      Factor2: = (Random mod Bignum);
      Writeln('How much is: ');
      Writeln(Factor1 :4);
      Writeln('x',Factor2 :4);
      Writeln('?');
      Readln(ans);
      if Ans = (factor1 * factor2) then
       Writeln('Right, ',Name,'!')
      else
       Writeln('Sorry, ',Name,'.');
    end;
end.
```

Be sure to save this program. You'll be asked to enhance it as part of the Mac-r-cises at the end of this chapter.

Other Calculations

We've discussed seven of Mac Pascal's fifteen built-in arithmetic operators and functions in some detail. The other eight involve complex mathematics, and are beyond the scope of this book. We will just mention them briefly before we close our consideration of numbers in Macintosh Pascal. If some (or all) of the operators and functions don't make any sense to you, don't worry about it. If you've gotten along this far in life without understanding or using them, chances are you won't need them desperately in the next week or two.

Trigonometric Functions

If you do advanced math that requires trigonometric functions, you'll be interested to know that Macintosh Pascal has built-in functions to calculate the sine (Sin), cosine (Cos) and arctangent (Arctan) of any angle it is given. Mac Pascal assumes that the angle parameters you give it are expressed in radians.

Logarithms

The natural logarithm (log to the base "e") of a number can be calculated in Mac Pascal by the expression

```
x: = ln(value);
```

where any Integer or Real numeric value can be used where the term "value" appears.

The Exp function is the inverse of the Ln function. Exp returns the exponent, in the base "e", of the value given as a parameter. In other words, the expression

```
Writeln(exp(13.498));
```

will print the power to which the natural log base "e" would have to be raised to give the answer 13.498.

Fixed-Point Arithmetic

A specialized number type in Mac Pascal is called "fixed point". You're not likely to run into it in everyday life, so don't worry about it. Fixed-point numbers are something like a cross between integers, which have no decimal points, and floating-point numbers, which have a decimal point with a variable position (that is, it "floats" around inside the number) depending on the value represented. A fixed-point number lets us represent values with decimal or fractional parts but without the mantissa-exponent tracking associated with floating-point numbers.

In case you *are* interested in manipulating such data, you should know that Macintosh Pascal includes commands for dividing (FixRatio), multiplying (FixMul), and rounding off (FixRound) such numbers. There is more information on this topic in the *Macintosh Pascal Reference Manual*.

This Is SANE?

A final point is worth noting. Apple has adopted an emerging national standard for the manipulation of numeric data and given it the acronym SANE (Standard Apple Numeric Environment). Macintosh Pascal contains a host of numeric manipulation procedures and functions from the SANE Library that permit such operations as:

1. converting numbers from one base system to another (for example, decimal to hexadecimal)
2. converting between numbers and strings
3. performing exponentiation to include a broad range of exponential values
4. calculating compound interest and annuity values automatically

If any of this sounds intriguing, feel free to experiment with the various functions available. Be warned that they won't work in the Instant window, however. To use them in a program, insert a statement that says

```
uses SANE;
```

right after the first line of the program (the one that gives the program a name). This instructs Mac Pascal to use procedures from a special part of its disk memory called a "library".

Complex Calculation Made SANE

Just for the fun of it, let's see how one of the built-in SANE functions can greatly simplify a relatively complex but commonly needed calculation: the future value of an annuity.

An annuity is a regular payment made into a bank account, insurance policy, mutual fund or some other such repository that accrues interest. The question we will try to answer is this: "If I put $150 per month into the bank for the next 5 years and the money draws 10.5% interest, how much will I have at the end of the five years?"

Here is the formula for the calculation:

$$FV = PMT\ (1 + R)n \quad \frac{1 - (1 + r)^{(-n)}}{r}$$

where FV is the future value, PMT is the regular payment, r is the interest rate, and n is the number of periods in which payments are made.

The SANE library includes two functions that are useful for this problem. One, called Compound, calculates compounded interest, and the other, Annuity, calculates the value of an annuity. The Compound function translates to the formula $(1 + r)^n$. The Annuity function is equivalent to the other portion of the formula. Therefore, to calculate the future value of an annuity with Macintosh Pascal, the following simple program will suffice:

```
program FutureValue;
uses SANE;
var
   fv,pmt,r,n:real;
begin
   Writeln('Enter the size of the regular payment: ');
   Readln(pmt);
   Writeln('The monthly interest rate is...');
   Readln(r);
   Writeln('How many payments?');
   Readln(n);
   Writeln('That schedule will result in');
   fv: = pmt*compound(r,n)*annuity(r,n);
   Writeln('a fund totaling $',fv:1:2,'.');
end.
```

The *Macintosh Pascal Technical Appendix* includes a major discussion of SANE.

Summary

This chapter has taught you how to do arithmetic using seven of the fifteen built-in mathematical functions in Macintosh Pascal. You've learned how to handle and display several types of numbers. You've learned how to determine whether a number is odd or even and how to determine the absolute value of a number. You've delved more deeply into the mystery of random number generation to understand how and why it works in Mac Pascal programs. Finally, you've become aware that there are many other useful mathematical functions available in Mac Pascal if you need them.

Mac-r-cises

1. The coin-tossing program has been used for years to teach people about computers. In this program, the computer simulates the flipping of a coin a predefined number of times and keeps track of how many times it comes up with "heads" and how many times it comes up with "tails". Write such a program. Have it allow the user to tell the computer how many times to toss the coin.
2. Make two modifications to the MultTables program you worked on earlier in this chapter. First, add a scorekeeping function so that the computer keeps track of the number the student gets right and at the end of the problem sequence prints out a percentage score for the session. Second, refine the program so that the student has three tries to get the answer right before the program gives feedback and updates the score.
3. A procedure that simulates rolling dice is frequently needed in computer simulations of games. Write a program that will keep rolling two dice and producing random results until users indicate by some means that they don't need any more dice rolls.
4. (Difficult) Write a program that takes as inputs the three coefficients in a quadratic equation and produces the solutions to the equation. Keep the number of lines to a minimum; use parentheses to consolidate the calculations as much as possible.

9

Making Your Own Procedures and Functions

When you finish this chapter, you'll know

- **What a procedure is**
- **How to define a procedure**
- **What structure procedures have**
- **How procedures use variables and parameters**
- **The difference between procedures and functions**

The past three chapters covered many of Macintosh Pascal's built-in procedures and functions, and later in the book we'll look at another set of them. But why be limited to what Mac Pascal gives you, generous though that is? This chapter will teach you how to design and create your own procedures and functions.

What Is a Procedure?

Procedures may be thought of as little programs (though sometimes they are not so little!). A procedure is a collection of Pascal statements that carries out a specific task.

To use a procedure in a program, you simply "call" it by putting its name in the appropriate part of the program. For example, the built-in procedure Writeln is invoked merely by typing the word Writeln. Generally (though not always) Writeln is provided with a parameter consisting of the information to be displayed on the screen (or other output device) and perhaps some data about the format of the display or printout.

What Is a Function?

A function may also be thought of as a little program (functions tend to be shorter than procedures). Like procedures, functions are collections of Pascal statements that carry out specific tasks.

So what's the difference between procedures and functions? Simply this: functions "return" values to the program lines that call them, while procedures normally do not. A function almost always carries out one or more processing steps on one or more pieces of information. It then returns to the program, as a result of that processing, one value that the program can use in some way. A function almost never alters the value of any other variable in the program.

A procedure, on the other hand, often works with several pieces of information and perhaps modifies all of them. It will return a value for the main program to use only if the program requires it to do so.

If all this talk about "returning" values seems confusing, don't worry. In a few more pages, after we've worked through an example or two, it should become perfectly clear.

Examples of Procedures and Functions

Let's look at two built-in procedures and functions you're already acquainted with to get a better understanding of the purposes of procedures and functions and of the differences between them.

Sqrt, which you met in the last chapter, provides a good example of a function. It is given a parameter and returns the value of the square root of that parameter. If we write in a Pascal program

```
x: = sqrt(16);
```

we expect the variable x to have a value of 4 when we're done. This is the value Sqrt has returned to the program. This value can then, for example, be assigned to a variable, further manipulated, displayed, or stored somewhere.

It would, however, make no sense to write in Macintosh Pascal:

```
x: = Writeln('Hello, there, procedure!');
```

Trying this will produce an error, because Writeln is a *procedure* and therefore is not designed to return a value to the program. Since there is no "result" of the Writeln procedure, attempting to assign this nonresult

to a variable causes Mac Pascal to have a distinct pain, about which it notifies us in no uncertain terms.

Procedures Can Return Values

Having said this, we should point out that it is perfectly *legal* for a procedure to return a value to the program. In fact, this is a fairly common use of procedures. Unless a procedure is explicitly told to return a value to the program, however, it will not do so. A function, on the other hand, simply can't help itself; its role in life is to return a value.

When a procedure does return a value, the value is not the name of the procedure and can't be used as such. On the other hand, a line like this

```
if sqrt(x)>0 then ...
```

means something. The *function* Sqrt returns a value that can be substituted in the place in a Macintosh Pascal program from which the function is called. Here, the program line calls the function Sqrt, which returns a value that can in turn be tested directly. Note that the value is not assigned to a variable in this case.

Trying to use a *procedure* name in such a situation would produce an error. We may, however, define a procedure that assigns a result to a *variable*, and then, in our main program, we may test the name of the variable for certain conditions. For example, let's say a procedure called CountWords counts (strangely enough) the number of words in an input text stream. Assume that the procedure assigns a value to a variable called TotalWords. We can call the procedure and use the result, as in:

```
CountWords(text);
if TotalWords = 0 then ...
```

Note, however, that we could *not* write

```
if (CountWords(text)) = 0 then ...
```

unless we defined CountWords as a function rather than a procedure.

When Should You Use Procedures and Functions?

You should seriously consider using a procedure or function (choosing the appropriate form based on what you want to do with the result) whenever you encounter one or more of the following conditions:

1. The operation involved will be used with many different variables at various times (as with the Sqrt function which will be used with any integer value for which we wish to find the square root).
2. The processing will be used at more than one point in a program (for example, Readln or Writeln which are used repeatedly in many programs).
3. The process being defined may be used by more than one program.

In other words, functions and procedures are handy shortcuts that you should use whenever you want to avoid writing the same lines of code over and over.

A Grapeful Program

Let's look at a short graphics program that takes advantage of procedures in Macintosh Pascal in a typical way. Type in the following program and, if you like it, save it on the disk.

```
program Grapes;
 var
  x,y,y1,count,num:integer;
 procedure row(num:integer);
  var
   count:integer
 begin
  y1:=y;         ← Save original value for later
  repeat
   for count:=1 to num do
    begin
     PaintOval(x,y,x+10,y+10);
     y:=y+10;
  end;
```

```
    num:=num-1;         ← Each row gets smaller by one grape
    x:=x+10;            ← Move starting position of each row down a bit
    y:=y1+5;            ← Indent first grape on row a little to the right
    y1:=y;              ← Save new position for use next time
  until num=0;
  end;
begin {main program}
  repeat
    Writeln('Point to starting location');
    repeat
      GetMouse(y,x);
    until button;
    Writeln('How many grapes?');
    Readln(num);
    if num>0 then
      row(num);
  until num=0;
end.
```

Run this program. It will ask for a starting location for a bunch of grapes. Position the crosshair where you want the upper left corner of the bunch to be and click the button on the mouse. (Notice the "until button"?) Now the program will ask how many grapes should be on the top row of the bunch. Enter a number and press the Return key. A bunch of grapes like those shown in Figure 9-1 will appear, and the program will again ask for a starting location. Continue until you've drawn enough grapes to fill your jug or your Drawing window. Then answer with a 0 when asked how many grapes to draw, and the program will end.

The Row Procedure

Notice that a procedure called Row is used in this program to draw several rows in each bunch of grapes. (The number of rows depends on the number of grapes you tell the program to place on the first row.) The procedure is repeated for each row to be drawn in the bunch. The main program calls Row several times — as many times as the user likes. The use of Row in this program is a classic use for a procedure.

Note, too, that the procedure is labeled as such and is followed by the procedure's name ("Row") and some information about the parameter(s) that are to be passed to it. In this case, row needs only one parameter, an integer value it calls "num". An error condition would result if a user attempted to try to call the procedure with no parameter, two parameters, or with a non-integer parameter.

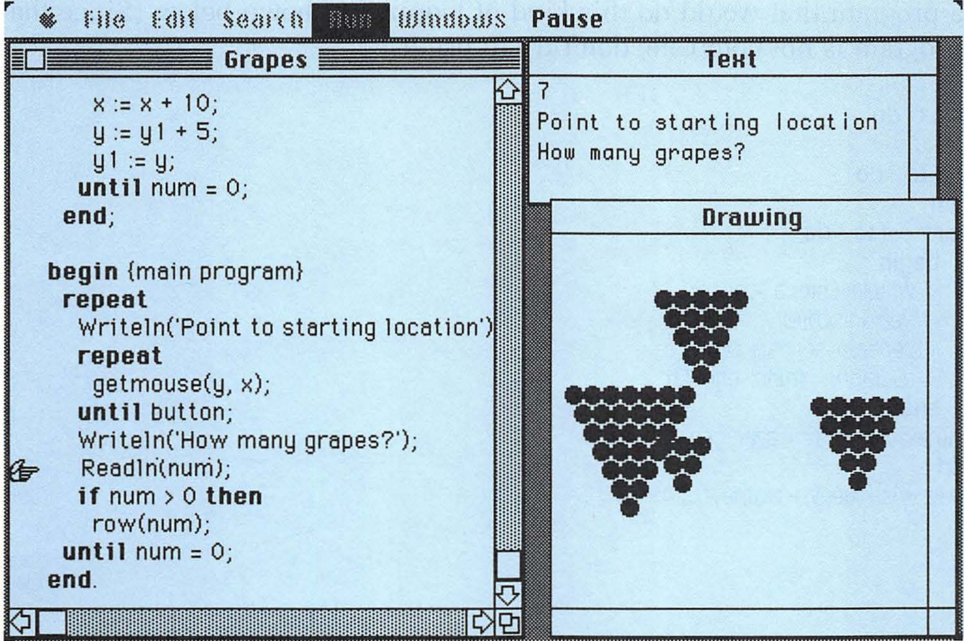
Procedures to Simplify Program Design

Procedures make programming easy. They permit the creation of programs that are easy to design and to understand. Effective and useful Pascal programs usually use several procedures. Some are included within other procedures. Some call other procedures. This kind of approach produces two fortunate outcomes: First, the main program is always the last thing in the program listing; and second, it is often quite short and, as a result, very readable.

Use procedures whenever you can. If a program does the same thing several times in several different places, consider that process a good candidate for conversion to a procedure, which can then be called whenever it is needed.

For example, suppose you're the secretary of a bowling league and you want to use the Mac's brainpower to calculate weekly averages for each bowler, each five-member team, and the league. Each bowler bowls

Figure 9-1. Sample Run of Grapes Program



three games, and there are twelve teams in the league. You *could* design a solution that required the following steps:

1. Enter each bowler's three individual game scores into a program called BowlerAverager and record the resulting three-game total and average on a piece of paper for later use.
2. Enter each team member's three-game total into a program called TeamAverager and record the resulting team average on a piece of paper for later use.
3. Enter each team's three-game, five-bowler total into a program called LeagueAverager.
4. Type up the weekly report.

Even this primitive method is preferable to using a calculator, let alone pencil and paper — but it still requires a lot of manual work. And isn't the whole point of having a computer the ability to avoid this sort of boring, time-consuming task?

Loops Inside Loops Inside Loops

One approach to this problem might use a series of nested For loops. A loop for each bowler would run three times. A loop outside that for each team would run five times. A final outside loop for the entire league would run twelve times. The three loops would look just about the same. Part of a program that would do this kind of looping is shown below. (Since the program is not complete, don't try to use it.)

```
for x: = 1 to 12 do
 begin
  for y: = 1 to 5 do
    begin
      for z: = 1 to 3 do
        begin
          Writeln('Enter a score: ');
          Readln(score);
          series: = series + score;
          average: = trunc(series/3);
        end;
      teamav: = teamav + average;
      end;
  leagueav: = leagueav + teamav;
  end;
end.
```

The organization of this little program segment is tough to follow. Just matching up the statements in the *for* structures becomes complex enough without also worrying about what is happening inside each program loop.

This partial program makes it clear why nested loops are usually inefficient and confusing ways of doing things in Pascal. Use them sparingly.

I've Already Programmed That This Week!

If you're going to use the same program statement group repeatedly, either within a program or in several different programs, by all means turn it into a procedure. For example, the bowling program must calculate the average of 3 or more numbers *73 times*. Setting up this program would be long and boring indeed if it required typing the same lines of code 73 times! And then, if you decide to write a program to compute your favorite stock's average closing price for the past 15 weeks, you would have to write what are basically the same lines of code all over again.

Procedures save tremendous amounts of programming time by supplying a store of often-used solutions that can be called into programs whenever they are needed. For example, let's solve part of our bowling-secretary problem with a program that uses procedures. The program asks for each bowler's three game scores for the week, prints out each bowler's average, and then calculates the team's average from that information. Notice what a workout the Averager procedure gets!

```
program BowlingSec;
 var
   total,scores,av,game,bowler,teamtot,score:integer;
 procedure Averager(total,scores:integer);
 begin
  ave: = trunc(total/scores);
 end; {averager}
begin {main program}
 total: = 0;
 teamtot: = 0;
for bowler: = 1 to 5 do
 begin
  for game: = 1 to 3 do
   begin
    Write('Score for game ',game:1,':');
    Readln(score);
    total: = total + score;
  end; {obtaining scores}
```

```
    teamtot: = teamtot + total;
    averager(total,3); {notice we pass total score and number of scores}
    page; {clears the Text window for appearance}
    Writeln('Bowler ',bowler:1:1,' average = ',ave);
    total: = 0; {must zero out score before next bowler's scores are input}
  end; {all bowlers' scores are in}
  Averager(teamtot,5); {team average = average three-game set for team}
  Writeln('Team Average = ',av);
end.
```

Note the Trunc function in the third line of the program. It converts a real number to an integer, which in this case has the effect of rounding it off.

Once you understand procedures, you'll find this program far easier to understand than the nested loop approach shown earlier.

Structure of Procedures

All the procedures used in this program have a basically similar structure. Like all Macintosh Pascal procedures, they follow certain simple rules, which can be summarized as follows:

1. All definitions of procedures start with the word *PROCEDURE*.
2. All procedures in a Macintosh Pascal program must normally be named and defined *before* being used for the first time.
3. Once defined, procedures may be used simply by entering their names into the program at appropriate points, along with any parameters required for them to know what to do.

This is the basic format for a procedure. Very complex procedures can include all of the commands and statements you've learned so far and many others, too. Procedures can even contain other procedures.

Commenting on Procedures

Remember about comments, those lines contained inside curly brackets? You probably noticed that the last program used a lot of them. Comments can be very helpful in keeping track of the structure of a program. They can show, for example, where each procedure begins and ends and where the main program begins. (The main program *always* appears as the last thing in the listing.) As we've said before, liberal use of comments can

make Macintosh Pascal programs easier to figure out later — both for other people who might be interested in how the program works and for the programmer, who perhaps could not otherwise remember what the structures are.

Allowing Data Reentry

"Error checking" is a useful feature to include in programs. As you know, Mac Pascal does some error-checking on its own, producing Bug boxes when it finds something it doesn't like. But what if a program requests numbers in a certain range? How can it check to see that a number the user enters is in the proper range? Without procedures, this would be virtually impossible.

To prove that, let's take a look at what might *seem* like a solution to this problem. Don't enter the following program segment; just look at it as you follow the discussion.

The program is supposed to ask the user to type in 300 numbers between 1 and 10, check to be sure they *are* between 1 and 10, and then finally (though not shown here) do something with those numbers.

```
begin
  for x: = 1 to 300 do
    begin
      Writeln('Enter a number from 1 to 10: ');
      Readln(number);
    if number<1 or number>10 then Writeln ('Wrong!') else
    tot: = tot + number;
    end;
  *
  *
  *
end.
```

This program segment works fine except that it ends up with fewer than 300 numbers if the user puts in any values outside the range 1 to 10. To be sure, the user will be *told* (none too politely) that he or she has not done what was requested, but the If...then...else statement has no way of going back to the Readln statement to have the user enter the number a second (or third or forty-ninth) time. Therefore, the results of the program won't be as expected.

To show you how this problem could be solved using a simple procedure, look at the following portion of a program (again, this is not a complete program, so don't try to type it in and use it...it won't work!):

```
procedure CheckNumber;
    if number<1 or number>10 then Writeln ('Wrong!') else do
            begin
                x:=x+1;
                tot:=tot+number;
            end;
end; {CheckNumber}
*
*
*
begin {main}
    x:=0;
    repeat
        Writeln('Enter a number from 1 to 10: ');
        Readln(number);
        CheckNumber;
    until x:=300;
end.
```

Notice here that the value of the counter variable "x" won't be incremented each time a number is input; it is incremented *only* when the value entered is in the proper range — i.e., between 1 and 10 — as determined by the procedure CheckNumber.

"Scoping" with Procedures

Did you ever wonder whether a procedure, since it looks so much like a program, can have its own internal (private) variables? Well, if you have, congratulations! You are on your way to becoming a first-rate Pascal programmer. (If the thought *didn't* occur to you, don't get upset; we may be keeping you so busy learning new material that you don't have time to ask questions.)

The answer to this question is an emphatic "Yes". Any procedure can access information that is available to it and to no other part of the program; or it can contain and use information given to it by the program or procedure which called it into action. The former pieces of information are called "local variables", and the latter are called "parameters". Let's take a look at both.

Local Variables in Procedures

Any procedure can contain within itself a variable declaration section just like those we've been using in our full-sized programs For example, the first part of a procedure for the bowling secretary's program, intended to add three numbers and produce an average, could look like this:

```
procedure Average3Scores;
 var
  score1,score2,score3,average: integer;
 begin
*
*
*
```

This format defines the variables Score1, Score2, Score3 and Average as "local" to the procedure Average3Scores. They're local in the same sense that most of us are: we're known in our own neighborhoods but not in Paris or New Delhi. Similarly, local variables have meaning only within the procedure in which they are defined. An attempt to print or otherwise use them outside that procedure will result in an error. So why would we want to use them? The first reason is that local variables provide for greater efficiency in a program's use of the Mac's memory and in execution. (The technical explanation for this need not concern us here.) Second, defining local variables within procedures makes it legal to have two variables in one program with the same name — or three or four or one hundred — so long as their "scope" doesn't interfere, as shown here:

```
program LocalVar;
      var
            totcost:integer;
      procedure Sales;
            var
                  LocalCost:real;      ← This variable known only inside procedure called Sales
      begin
      end;
      procedure Markets;
            var
                  LocalCost:real;      ← Variable of same name known only inside procedure Markets
      begin
      end;
begin {main program}
{some statements}         ← The main program doesn't know either local variable
end.
```

Readability is one of the key goals in Macintosh Pascal programming, and local variables help to achieve this goal by making it possible to write programs with meaningful variable names at all levels.

Parameters in Procedures

Quite often, when we call a procedure into action, we give it information about the values on which we want it to perform its assigned tasks. Returning to our bowling league example, when we call the Average3Scores procedure, we might give the procedure three scores to average. This would, for example, permit us to gather the scores in the main program or in another procedure that checks to be sure the scores are valid, and then "give" the scores to the procedure, which would diligently calculate the average for us.

If a procedure uses parameters, the procedure must be defined and called in a way that allows for this use. The *definition* of the procedure must show the parameters in parentheses following the procedure name and tell Macintosh Pascal what types of variables they are. The line in the Mac Pascal program that *calls* the procedure must put those variables — *in the correct order* — inside parentheses as well. Part of the procedure Average3Scores, for example, would look like this:

```
procedure Average3Scores (score1,score2,score3: integer);
```

while the call to that procedure might look like this:

```
Average3Scores (a,b,c);
```

where a, b, and c are integer values (since that's what the procedure has been told to expect) obtained from the part of the program that now calls on Average3Scores to do its thing.

Passing Parameters Back to the Program

So far we've talked about passing parameters *from* a program *to* a procedure, which then operates on these values to produce some result. In this case, as we've explained, we show the name of the parameters in parentheses following the procedure name:

```
procedure DoSomething(alpha,beta:integer);
```

However, if a procedure is going to pass a value *back to* the calling program, then a *variable parameter* must be used. This is specified by preceding the variables in parentheses with the keyboard *Var*:

```
procedure DoSomething (Var alpha,beta:integer);
```

If you fail to do this, you'll find that your program doesn't know what changes were made to the variables by the procedure.

A Three-Sided Example

To gain a better grasp of parameters and variables, type in the following small example program, then Save it, Check it, and Run it. The program creates a triangle based on two points we specify and a third point (apex) calculated by a procedure.

```
program DrawTriangles;
 var
  xpoint1,ypoint1,xpoint2,ypoint2,count:integer;
 procedure Triangle (x1,y1,x2,y2:integer);
  var
   xapex,yapex:integer;
  begin
   Writeln('Point to where you want this apex.');
   repeat
    GetMouse(xapex,yapex)
   until button;
   MoveTo (xapex,yapex);
   LineTo(x1,y1);
   MoveTo(xapex,yapex);
   LineTo(x2,y2);
   Writeln('Here''s one triangle....');
 end; {Triangle procedure}
begin {main program}
 Writeln('Point to one corner and click.');
 repeat
  GetMouse(xpoint1,ypoint1)
 until button;
 Writeln('Now a second corner, same way.');
 repeat
  GetMouse(xpoint2,ypoint2)
 until button;
 MoveTo(xpoint1,ypoint1);
 LineTo(xpoint2,ypoint2);
for count: = 1 to 5 do
  triangle(xpoint1,ypoint1,xpoint2,ypoint2);
end.
```

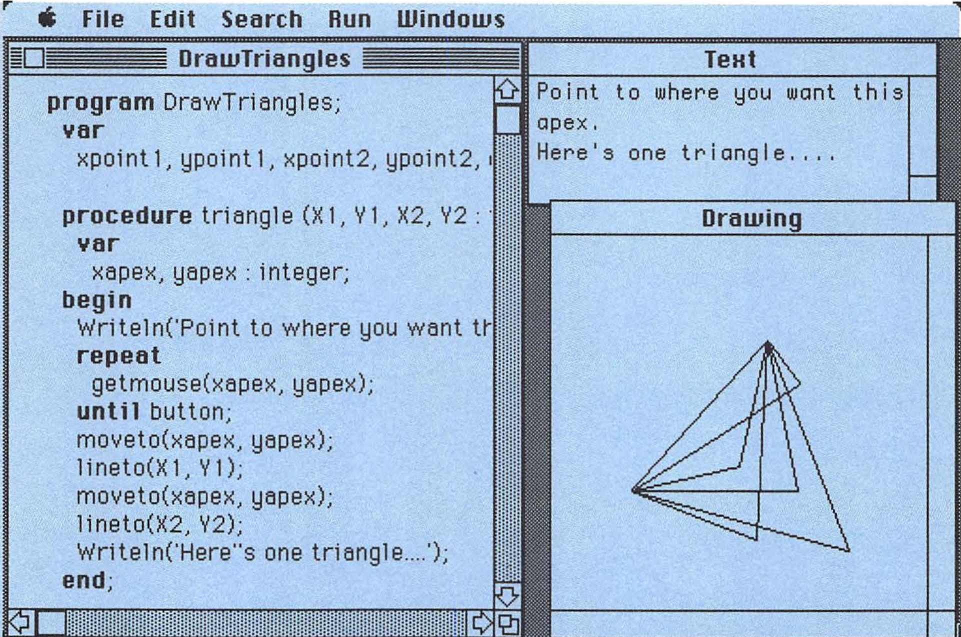
The program tells the user to select two points in a triangle with the Mac's mouse. It then uses these two points to create five triangles with apexes that are also selected with the mouse. Two notes of caution about using the program are in order. First, be sure the Text window is visible when you begin to Run the program. Second, click and release the mouse button very quickly to avoid ending up with straight lines focusing on a point instead of pretty triangles.

Figure 9-2 displays a sample run of the program, including both Text and Drawing windows. You might experiment with this program to see what interesting graphic shapes can be created using nests of triangles. You can get some ideas from the ancient Oriental paper-folding art of Origami, which is based on triangles.

Making Variables Local

Notice that the second-to-last line of the program, which calls the Triangle procedure, passes variables by the names by which they are known in the program: xpoint1, ypoint1, xpoint2, and ypoint2. The definition of the Triangle procedure shortens the variable names to x1, y1, x2, and y2. The

Figure 9-2. Sample Run of DrawTriangles Program



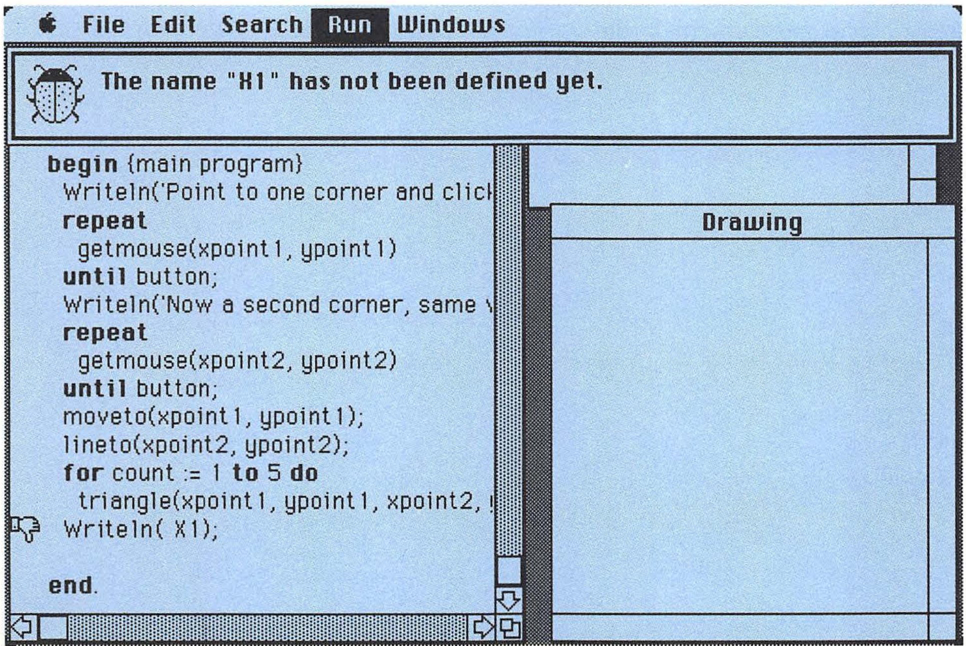
variables with these names aren't declared (and therefore can't be used) anywhere else, but because they are defined and given a type in the definition line for the procedure, they can be used locally in the procedure.

To prove this point about local variables, try adding a line that attempts to print out one of the variables x1, x2, y1, or y2 after the five triangles have been drawn. Be sure to include the line in the main program before the end statement. When you try to Run this modified program, a Bug box like the one shown in Figure 9-3 should appear.

Scoping a Conclusion

The idea of a local variable's *scope*, which we mentioned briefly earlier in the chapter, is also relevant here. In Pascal, variables, procedures, and a few other things you'll learn about as you move through the rest of the book have scope. The local variables x1, x2, y1, and y2 have as their scope the procedure called Triangle. In other words, these variables are recognized *only* within that procedure.

Figure 9-3. Bug Box Shows Local Variable Unknown in Program



Getting Information into a Procedure

The DrawTriangles program illustrated one way of getting a parameter passed to a procedure. The mouse and its button were used to gather information about where something should happen. The program then passed those parameters in the "parameter list" that followed the call to the procedure Triangle. Constants, calculated values, or keyboard inputs may also be passed to a procedure. So can information extracted from files of information stored on the Mac disk. The process of getting the data is explained in Chapters 13 and 15.

Circles Have a Point

The final sample program of this chapter demonstrates the use of two procedures, both called from a program. The example will show why and how procedures are used in Macintosh Pascal programs. Type in the program, Check it, Save it, and Run it.

```
program FlyingCircles;
 const
  xmax = 150;          ← Sets boundary of area where drawing will occur
  step = 10;
var x,y,xstart,ystart:integer;
procedure CircleMove(xpos,ypos,step: integer);
begin
 while ((xpos<xmax) and (ypos<ymax)) do
  begin
    FrameOval(x,y,xpos + step,ypos + step);
    xpos: = xpos + step;
    ypos: = ypos + step;
  end;
 Sysbeep(5);
end; {procedure CircleMove}
procedure DrawIt;
begin
  repeat
   CircleMove(x,y,step);
   x: = x + step;
   y: = y + step;
 until button;
end; {procedure DrawIt}
```

```
begin {main program}
  Writeln('Click at starting point.');
  repeat
    GetMouse(xstart,ystart)
  until button;
  x: = xstart;
  y: = ystart;
  DrawIt;
  Writeln('Click again to stop it.');
  PenPat(white);
  x: = xstart;
  y: = ystart;
  DrawIt;
PenNormal;
end. {main program}
```

Note the SysBeep procedure. As you might guess, executing this procedure causes the Mac's speaker to sound a short beep.

Figure 9-4 shows a sample run of the program, with the design appearing in the Drawing window.

Figure 9-4. Sample Run of FlyingCircles Program

```
 File  Edit  Search  Run  Windows

FlyingCircles
program FlyingCircles;
 const
  xmax = 150;
  ymax = 150;
  STEP = 10;
 var
  x, y, xstart, ystart : integer;
 procedure circlemove (xpos, ypos, s
 begin
  while ((xpos < xmax) and (ypos < yr
   begin
    frameoval(x, y, xpos + step, ypos
    xpos := xpos + step;
    ypos := ypos + step;
   end;
  sysbeep(5);
 end;

Text
Click at starting point.
Click again to stop it.

Drawing
```

This program methodically creates and erases an interesting graphic shape. In the lines that set the variable x to be equal to the value of xstart and y to be equal to the value of ystart, it demonstrates how to set up variables to pass to procedures. It also saves the original values for later use.

Notice that the program doesn't need to pass the variables x and y to the DrawIt procedure, since these are "global" variables — that is, variables known throughout the program. The procedure can use them quite nicely without our help.

If you like, try modifying the program. For example, try erasing the Drawing window after each funnel is completed; the effect is different, if perhaps not quite so artistic. Or try adjusting the constant values at the beginning of the program to see what other effects you can create. One interesting idea is to make the variable y increment by a different amount from x; this will result in an oval rather than in a circle. (Be careful, though, not to create an "inverted" oval, which will be invisible!)

Where Do Functions Come In?

We've spent all of our time so far on the subject of *procedures* and have yet to mention functions, which may cause you to become suspicious of our chapter title. Fear not; we will now turn our attention briefly to the subject of functions.

One reason functions get so little space in this chapter is because they have so much in common with procedures that we can almost say that everything we've told you about procedures applies to functions. However, there are some small but important differences.

Similarities to Procedures

Like procedures, functions are program elements which are good tools for shortening programs and performing repetitive tasks. Like procedures, functions can use local variables, too. Within a function, we can perform any valid Pascal task — which also makes functions like procedures.

Differences from Procedures

The key difference between a procedure and a function is that a function always returns a result, whereas a procedure will only do so if asked by the program which calls it. This results in a requirement for defining a function which does not exist in defining a procedure — the program must be told the *type* of the value to be returned.

As an example, we'll create a function called "RandomRange" which is given two numbers and is asked to return a random number between the two values.

```
function RandomRange (num1,num2:integer):integer;
begin
    RandomRange: = abs(random mod (num1-num2 + 1) + num1;
end;
```

This function uses the random function (see Chapter 8 for an explanation of its use) to produce the random number desired. Notice how the program can use the result of a function call directly. Thus, it is perfectly legal for a program containing the above function to include a line which says:

```
RandomNumber: = RandomRange(10,100);
```

This would have the effect of assigning the *result* returned by the function RandomRange to the variable RandomNumber directly. If, on the other hand, RandomRange were defined as a procedure, this would require two steps: one to call the procedure and one to assign the result to a variable. That might look something like this:

```
RandomRange(10,100);
RandomNumber: = ValueReturned; {or whatever name was included in
the procedure}
```

Notice that in defining the function RandomRange we included the type "integer" after a colon following the name of the function. This tells the program that the function is designed to produce a result of type integer. Misuse of this variable type would elicit an error, as assigning a value of one type to a variable of another type always does.

Summary

This chapter demonstrated the creation and use of Macintosh Pascal's best shorthand device — procedures. You've learned the difference between procedures and functions and when to use each. You've seen how to define a procedure and why procedures can make programs easier to read, to understand, and to change. You've learned how to pass parameters between the main program and a procedure. You've learned what local variables are and how to define and use them in procedures.

Mac-r-cises

1. Define a function called CountIt, which takes a string up to 255 characters long as input and returns the number of words in the string. For this purpose, a "word" is anything followed by a space. Use this function in a program called WordCounter.
2. Pick any program you wrote for an earlier chapter's Mac-r-cises section. Make it easier to follow and more efficient by substituting one or more procedures for parts of the program.
3. (Difficult) Using the Grapes program as a model, write a program that has the user click on the starting point for a brick wall and tell the program how many bricks wide the wall should be, and then draw the wall. Have the program "stagger" the blocks on alternate rows so the wall won't fall over.

10

Structured Data Types I: Arrays

When you finish this chapter, you'll know

- **What an array is**
- **How to get information into an array**
- **How to get information out of an array**
- **How to use two-dimensional arrays like tables**

This chapter will show you another important tool for creating complex and useful Macintosh Pascal programs: ways of carrying more than one piece of information at a time through a program. Once you've added the rules and methods for using complex data types to what you already know about using procedures, you'll be on the verge of becoming a serious Mac Pascal programmer.

Will I Need an Array Gun?

We have previously dealt exclusively with "simple" data types — those that can hold only one piece of information at a time. A variable of type Integer, for example, can have only one value at any one point in the program, and the value must be in the proper range for an integer. The value of the variable can be changed, but the variable cannot stand for or hold more than one value at a time.

For many data processing applications, this limitation presents no problem, but for the most complex uses to which we put computers, data types capable of holding more than one item are essential. Take a quick look at a (relatively) short example to get a better idea of what we mean.

Showing the Score

Suppose you needed to write a program that would ask the user to enter, say, ten golf scores and would then show the scores entered, average them, and display the result. How could you do that without using data types capable of holding more than one piece of information? Well, you could write a small For loop that would gather the ten scores, add each to the running total, and display the average. That would be easy enough. But how could you display the scores *after* all ten had been entered?

You'd need to define a separate variable for each score so that you could later display all the scores individually. Hopefully, you could do this in such a way that someone reading the program would see that each of the ten variables was intended to hold the same kind of data and that the pieces of information were related to each other — perhaps by using variable names such as Score1, Score 2, and so on; but the program would be quite cumbersome. Here's a skeleton of what such a program might look like. (Since it's not a complete program, do *not* try tc enter and run it.)

```
program GolfScores;
  var
     score1,score2,score3,score4,score5,score6,score7:integer;
     score8,score9,score10,total,average,count:integer;
  begin
   Writeln('Enter the first score, please:');
   Readln(score1);
   total: = total + score1;
   Writeln('Enter the second score, please:');
   Readln(score2);
   total: = total + score2;
etc.
etc.
{when done with all ten scores...}
   Writeln('Round 1 score = ', score1 :2);
   Writeln('Round 2 score = ', score2 :2);
etc.
etc.
```

Attempts to streamline this program will be frustrated by the fact that a For loop to cause the program to get ten different variables cannot be used if the ten have different names. Similarly, a For loop cannot be used to print ten different variable names. What if the program had to deal with 100 golf scores instead of ten? Or what if a paycheck program had to deal with, say, 1,875 employees? We'd wear out a lot of fingers

typing in all those lines of program code, to say nothing of creating a degree of probability approaching certainty that we'd end up making mistakes.

Much of the data we deal with in real life takes the form of lists and tables such as those shown in Figure 10-1. We will assume a *list* to be a sequential collection of, for example, items to buy at the store, tasks to do at the office, or steps to take in writing a program; a list, then, is arranged in a single column. We will assume a *table*, on the other hand, to contain at least two columns of information. For example, a list of things to do at the office, the dates on which they must be completed, and the people to whom we must report their accomplishment could be thought of as a table arranged in rows and columns.

Figure 10-1. A Table Has Two or More Columns; a List Has Only One

Entering the Array

Both lists and tables are represented by *arrays* in Macintosh Pascal. Arrays and other "structured data types" are just about essential for handling the kinds of applications we've been discussing.

An array can be defined so that any piece of information in it can be put in the proper place, taken out and worked with, modified, put back, and printed or stored. An array can contain any of the simple variable types we've encountered so far: integers, real numbers, characters, or strings. An array can contain only *one* type of data, however. Mixing data types requires a different kind of data structure, which we'll discuss in Chapter 11.

A Better Way to Score

Let's look at a sample program that will solve our golf score problem. Typing it in, running it, and examining it will provide a good introduction to arrays in Macintosh Pascal. Here's the program:

```
program GolfScores1;
 var
  x,total:integer;
  GolfScores: array [1..10] of integer;
procedure GetScores;
 var
  x:integer
   begin
     total: = 0;
     for x: = 1 to 10 do
       begin
       Writeln('Enter score #',x:1);
       Readln(GolfScores[x]);
       total: = total + GolfScores[x];
     end;
  end;{GetScores}
procedure ShowScores;
 var
  x:integer
begin
  Writeln('Here are your scores: ');
  Writeln;
  for x: = 1 to 10 do
   begin
     Write('Round ',x:2,'    ');
     Writeln(GolfScores[x]:2);
   end;{ShowScores}
```

```
end;
begin {main program}
   GetScores;
   ShowScores;
end.
```

The main program calls the procedure GetScores, which asks the user to enter ten golf scores, using a For loop and the value of the counter "x" to prompt for each score individually. The procedure ShowScores then displays the scores in the same order. In other words, GetScores puts information *into* an array called GolfScores, and ShowScores gets the information back out.

Using Arrays

Now we expect you understand what an array is: a list or table of data of a predefined maximum size, which contains information that is all of one type. Let's go on, then, to look at how arrays are used in real-life programming problems.

Setting Up an Array

Information is usually put into an array with a For loop. Look at the GolfScores1 program again. Notice that the procedure GetScores is nothing but a For loop that goes through ten iterations, getting a score each time and stuffing it into the array called GolfScores at the next location.

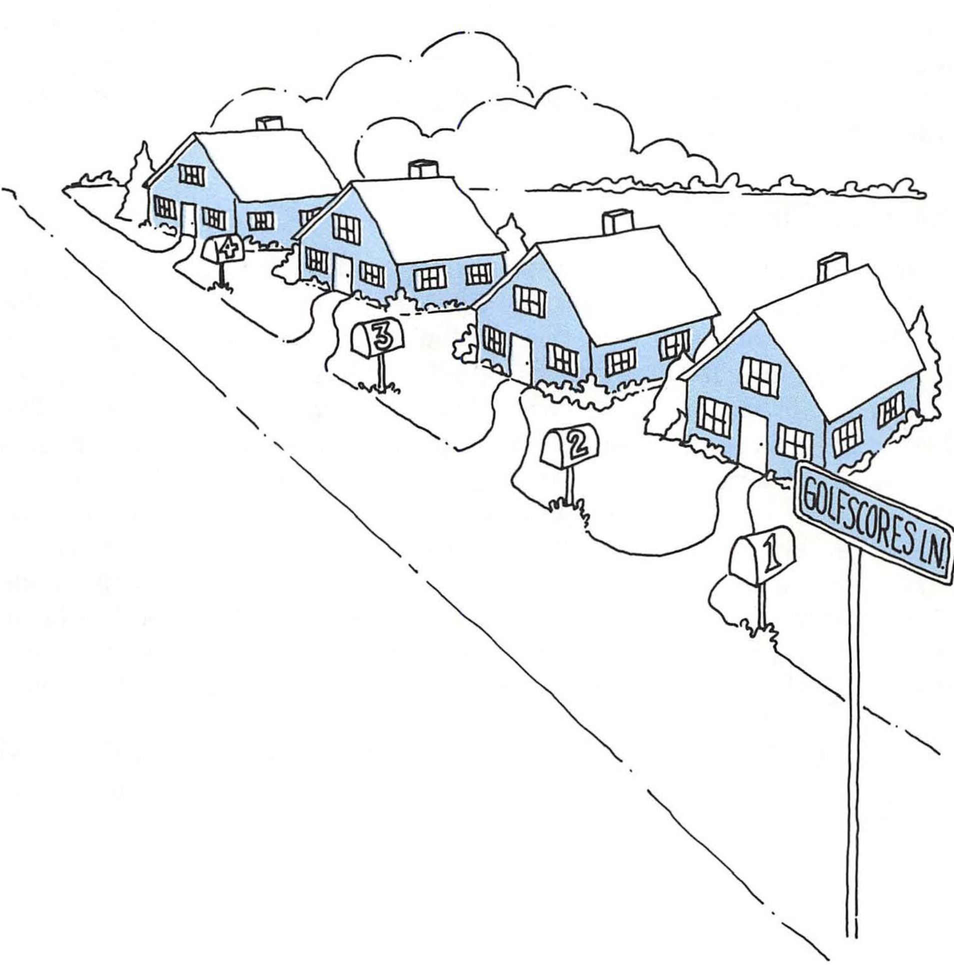
What do we mean by saying that an array has "locations"? The locations are something like mailing addresses. When the program defined the variable GolfScores to be an array containing up to ten integer values, the Macintosh set aside ten consecutive locations in its memory, labeled the whole block "GolfScores", and then labeled each of the ten empty "boxes" with a consecutive number between one and ten. If the block were a street, the address of the third box would be 3 GolfScores Lane. Instead, Macintosh Pascal uses the terminology "GolfScores[3]" to address the third box in the ten-item GolfScores array. Figure 10-2 should help you get the picture.

The number in brackets following the array name GolfScores is called an *index*. The index tells you which member of the array you're talking about: the third, seventh, or whatever.

Getting Information into an Array

Now that you know how items in an array are addressed, putting information into an array should be a piece of cake. Just use normal assignment statements with the array address on the left and the value to be placed into that array address on the right, and *voilà!* That's what the GetScores procedure in our GolfScores1 program does. Each time through the loop, the pointer "x" is incremented so that it points to the next location in the

Figure 10-2. Array Locations Are Like Addresses



array and assigns the user input to that variable. The line that performs this action looks like this:

```
Readln(GolfScores[x]);
```

As with any other assignment process, data to be put into an array can come from any of several different places: user input at the keyboard, mouse selection (as in the built-in procedure GetMouse), calculations, constants, or the Mac diskette, where the information may be stored in a file. It may even come from more than one place, particularly in the case of a table.

Getting Information out of an Array

Information may be extracted from an array just as easily as it was placed there — perhaps more so. Simply use another assignment statement, this time with the array address on the *right* and the variable to which the value you're extracting will be assigned on the *left*. For example, if you wanted one of our golf scores assigned to a variable, you could do it with a line of Mac Pascal program code like this:

```
ThisScore: = GolfScore[x];
```

Sometimes, as in the present GolfScores1 program, a variable doesn't need to be assigned a value because it will simply be printed or displayed on the Mac's screen. In that case, use a Write or Writeln statement that includes the array address in the parameter that defines what is to be written or printed.

Array Dismay

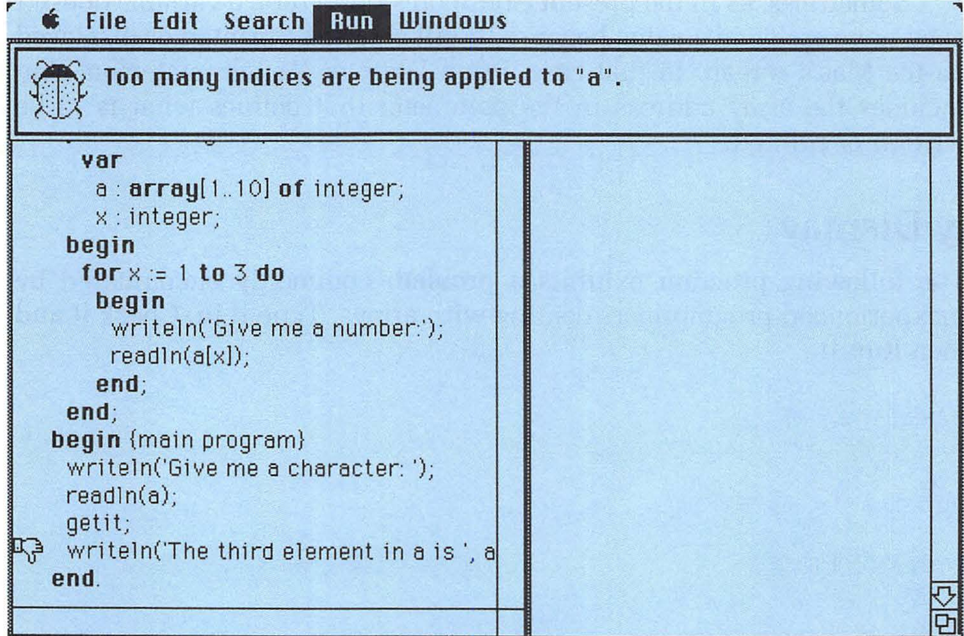
The following program exhibits a problem commonly encountered by inexperienced programmers dealing with arrays. Type it in, Check it and then Run it.

```
program ArrayDismay;
var
    a:char;
procedure GetIt;
var
    a:array[1..10] of integer;
    x:integer;
```

```
begin
for x: = 1 to 10 do
        begin
                Writeln('Give me a number:');
                Readln(a[x]);
        end;
end; {GetIt}
begin {main program}
Writeln('Give me a character:');
Readln(a);
GetIt;
Writeln('The third element in "a"is',a[3]);
end.
```

When we run this program, a Bug box tells us we've used too many indices for an array variable called "a" (see Figure 10-3). We *know*, however, that we've defined it as an array with one index (a list) and that we've tried to insert a single value in the correct range into it, so why do we get the error message?

Figure 10-3. What Do You Mean, "Too Many Indices"?



We've simply forgotten that the main program doesn't know about the array called "a", because it exists only inside the procedure "GetIt". If we really want the main program to be able to report the value contained in the array variable "a" — specifically in the third position of "a" as the program seems to want to do — we're going to have to do two things. First, we're going to have to move the line declaring the array "a" into the main program block (i.e., put it in the main Var section at the top of the program). Second, we're going to have to rename either the character variable or the array variable, since we can't have two variables of the same name defined with the same scope.

It is, of course, highly unlikely we'd make such a mistake as shown here in a short program like this. But in a long program, it's not difficult to lose track of which variables are used in which blocks and what their scope is. This is *particularly* true if we use such meaningless names for variables as "a" instead of giving variables names that tell us what their use is.

Tables

Tables are sometimes referred to as "two-dimensional arrays" because they have, well, two dimensions: they have, let us say, height and width. These dimensions are referred to in Macintosh Pascal as rows and columns. Table 10-1 shows that rows are the horizontal elements, columns the vertical. Since a two-dimensional array has at least two items stored

Table 10-1. Rows and Columns in Two-Dimensional Array

	column 1 *Last Name*	*column 2* *First Name*	*column 3* *Company*	*column 4* *Next Sales Call*
row 1	Allen	Bradley	Exeter Toys	7/12
row 2	Huntington	Donald	CBI	6/23
row 3	Johnson	Samuel	Dictionary House	5/4
row 4	Blue	Blackand	Sticks & Stones	
	*			
	*			
	*			
row 99	Smythe	Victoria	XYZ Computers	11/11

in row 5, for example, we need a way of addressing data in a two-dimensional array that is different from what we use with a one-dimensional array, or list.

The address for an item in a two-dimensional array simply states first the row number and then the column number, with the two separated by a comma. Thus the piece of information in the fifth row, second column of a table would be addressed something like this:

```
Writeln(table[5,2]);
```

Manipulating Data in a Table

Putting information into a two-dimensional array is done the same way as setting up a list, except it requires two nested For loops — one loop counting rows and the other counting columns. The Table1 program below demonstrates this concept. If you wish, type the program into the Mac and Run it.

```
program Table1;
var
 row,column:integer;
 table:array[1..5,1..3] of integer;
begin
 for row:=1 to 5 do
  begin
   for column:=1 to 3 do
    begin
      Writeln('Enter row ',row:1,', column ',column:1);
      Readln(table[row,column]);
   end;
  end;
end.
```

It may not be entirely obvious what this program is doing, so let's take a closer look at it.

The Table1 Program

This program sets up counter variables called Row and Column to correspond to the rows (horizontal groups) and columns (vertical groups) of information in a table. The program also creates an array called Table, which is 5 rows deep and 3 columns wide (as shown by the numbers in the array definition line).

The program simply goes through two nested For loops to get the information for each row and column. The main For loop begins with "for row: = 1 to 5 do". The first time through the loop the variable Row stays at the value 1, while Column increments from 1 to 3. As a result, when the first run through this loop is complete, data has been put into positions [1,1], [1,2], and [1,3] of the Table array. (Remember, the locations correspond to row 1, column 1; row 1, column 2; and row 1, column 3.)

The program now goes to the next row by incrementing the variable Row and does the same thing for row 2, then row 3, row 4, and finally row 5.

Extracting Information from a Table

To retrieve or extract data from a table, just reverse this process. Assign the array address with the row and column numbers to a variable (or use it in a Write or Writeln statement), then manipulate it in the same way as you would a simple piece of data of the same type.

Putting It All Together

Let's have some fun with a program example that uses a two-dimensional array for a possibly useful (or at least amusing) purpose. Type in the program (note that the group of lines making up the bulk of the Get-Favorites procedure would be an excellent place to use the Mac's built-in Copy and Paste editing features), Check it, and Save it. Then read the material following the listing to learn a little about what the program does and how to use it.

```
program FickleFavorites;
 var
  favorites:array[1..5,1..3] of string;
  row,column:integer;
  ans:string;
 procedure GetFavorites(row:integer);
 begin
  Writeln('Favorite # ',row:1:1,' name:');
  Readln(favorites[row,1]);
  Writeln('Favorite # ',row:1:1,' age:');
  Readln(favorites[row,2]);
  Writeln('Favorite # ',row:1:1,' field:');
  Readln(favorites[row,3]);
 end;
```

```
procedure Fickle;
 var
   name,age,field:string;
 begin
   repeat
     Writeln('Which favorite person number gets it?');
     Readln(row);
   until ((row< = 5) and (row>0));
   name: = favorites[row,1];
   age: = favorites[row,2];
   field: = favorites[row,3];
   Writeln(name,' is ',age,' years old');
   Writeln('and famous in ',field,'.');
   Writeln;
   Writeln('Who replaces this superstar turned has-been?');
   GetFavorites(row);
 end;{Fickle}
begin {main program}
 for row: = 1 to 5 do
   GetFavorites(row);
 repeat
   Writeln('Give someone the axe?');
   Readln(ans);
   ans: = copy(ans,1,1);
   if ((ans = 'Y') or (ans = 'y')) then
     Fickle
   else
     Writeln('Done!');
   until ((ans = 'N') or (ans = 'n'));
end.
```

This program allows us to enter up to five of our favorite people in whatever field or fields of endeavor we like. In each case, we supply the person's name, age, and field. Then the program asks if we'd like to axe someone. Do it a few times just to see how things work. Bump off your fifth favorite, put a different person in that place, and then bump that new one out. Discover how getting information into and out of an array works.

The program lacks one element it needs in order to be really useful: it does not permit us to reexamine entries after we've made them. Well, hold onto your mouse, because we're going to give you a chance to fix that problem in the Mac-r-cises at the end of the chapter!

Play with the program as long as you like, but remember, it will "forget" everybody when you exit the program; the names will have to be entered all over again. (The program itself, of course, is safe on the disk.)

Designing Arrays

Arrays are obviously a useful type of data structure; they represent most things in the real world more precisely and efficiently than simple data types ever could. In designing arrays, however, you need to keep some basic rules in mind.

> 1. Think about the array's shape, size, and contents in advance. Strive to have a user enter information into the system and the array in the order in which the user is used to working with the data. For example, don't ask for a person's age before asking for the person's name.
> 2. Define the array to be big enough to accommodate the largest amount of information you can anticipate storing in it. You can always enter less than the defined maximum, but you can't enter more!
> 3. Be careful about the scope of an array so that you don't end up printing or using information different from what is expected.

A Word about Subrange Data Types

The program GolfScores1 earlier in this chapter exposed us to a new variable type in the third line of the *Var* section of the program. Look back at that program now and notice that this line defines a variable called GolfScores as an "array of integer". The line also contains a strange construction inside square brackets with the number 1, two dots, and the number 10. This strange construction is called a "subrange" data type.

We must tell Macintosh Pascal the size of any array we define. In the process, we sometimes (though not always) also tell Pascal what valid contents of the array will consist of. We do this by defining the array to be made up of a subrange of an otherwise large or limitless range of values.

For example, defining GolfScores to be an array [1..10] of type Integer tells Pascal that the array will have ten values in it at most. This allows the Mac to know how much memory it will need for this particular part of the program and permits it to efficiently keep track of where the information is stored in the memory.

Subranges with Simple Data Types

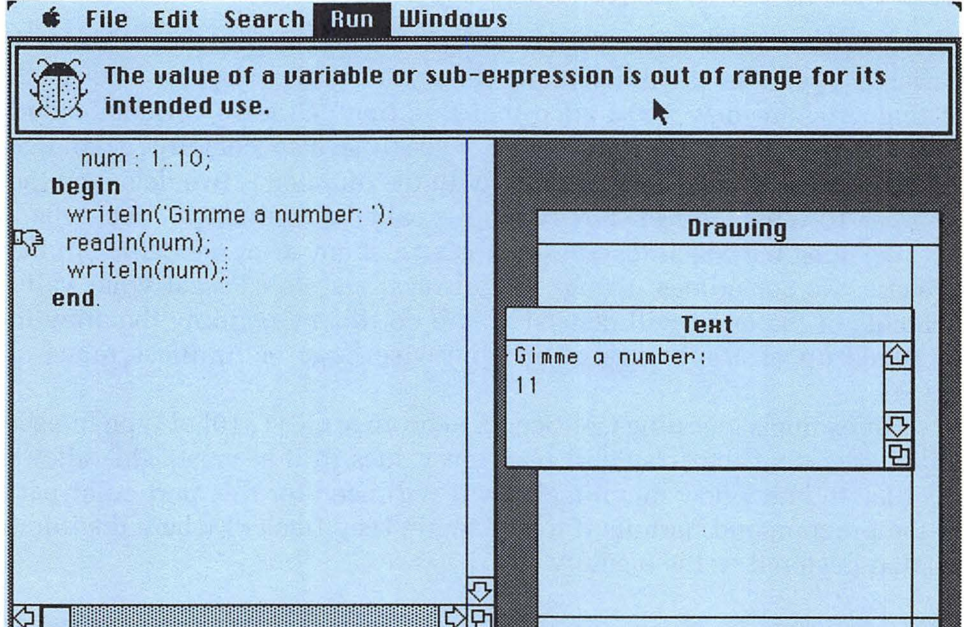
Let's briefly take a look at how to use a subrange data type outside an array. Strangely enough, this will lead us neatly back to its use in structured data types.

We can define a variable named, say, Value to contain only a certain range of values, for example, one to ten, by defining it as shown in the following simple program:

```
program Subrange1;
var
  value: 1..10;
begin
  Writeln('Gimme a number:');
  Readln(value);
  Writeln(value);
end.
```

Type this program in and Run it. (Don't bother to Save it.) Try entering a number larger than ten. The result should resemble Figure 10-4. Mac

Figure 10-4. Subrange Data Type Value Out of Range



Pascal produces a Bug box if the user attempts to enter any value outside the subrange defined for the variable.

In this chapter, subrange data types indicate only the *size* of an array. Chapter 11, however, will show you how to get great mileage out of subrange data types for error-handling and program definition.

Subranges Are Limits, Not Requirements

It is important to note again that the subrange type sets the *maximum* size for the array being defined. The array may have fewer elements; it just may not have more. Thus, the GolfScores program could easily ask the user how many scores to enter (up to ten) and then set a variable to that value for the loop and average calculation processes. As long as the number entered is ten or smaller, the program will run as expected.

Summary

This chapter has acquainted you with a very useful Macintosh Pascal concept: structured data types. Specifically, you've learned about one-dimensional arrays (which resemble *lists* in the noncomputer world) and two-dimensional arrays (which are *tables* in real life). You've learned how to get information into such arrays, how to get data out of them, and how to set them up and use them in various situations.

You've seen that arrays have the disadvantage of requiring their data to be all of a single type. Arrays will not, for example, permit manipulation of an employee's name (string), pay rate (real), or stock option status (boolean). Such mixing of data types requires the use of more advanced data structures — subject of the next chapter.

Mac-r-cises

1. Okay, as we warned you, your first assignment is to modify the program FickleFavorites to permit the user to look at the Favorites list in addition to axing and replacing people in it. (We'll give one tiny hint: You already have a way for the user to give an answer to the question about axing someone; just add another single-letter response that means "Let me see my list again" and write a procedure to display the list.)
2. Write a program that permits information to be added to a table and then converted to a list, row by row.

3. Write a program that creates a table having the same number of rows as columns (a "square" table). Then have the program transpose the table, putting the first row of the input table into the first column of a new table, the second row into the second column, and so on.
4. (Difficult) Write a program that accepts six numbers from the user and then draws a horizontal bar graph in the Drawing window. Each line of the graph should be proportional to the others in terms of the value of the number. In other words, if the user types in a 10 and then a 20, the second bar should be twice as long as the first. (Hint: Try using the value entered — or some multiple of it — as one of the definition points in a PointRect statement.)

11

Structured Data Types II: Sets and Records

When you finish this chapter, you'll know

- **How to create your own data types — and why**
- **What a "set" is and how to use it effectively**
- **How to use a record to mix types of data**
- **How to get information into and out of records**

This chapter will look closely at the two most powerful structured data types in Pascal. In some ways, the two types are extensions of the arrays surveyed in the last chapter. In other ways they are very different.

The Type Section of a Macintosh Pascal Program

We have not yet discussed the part of a Macintosh Pascal program where sets, records, and other data types are defined. Before we delve into what a set is or what a record does, let's look at this section of a Pascal program so you'll know how to define sets and records when the time comes.

The Const section declares the values of constants in programs; the Var section defines the types of variables to be used. A third section, Type, falls between the other two sections in Pascal program structure. Macintosh Pascal treats sets and records as *types*.

Look at the following line of Mac Pascal code (don't type it in). We'll use it in the discussion about types that follows.

```
program Test;
 type
  color = (blue,green,red,purple,orange,yellow);
```

The Power of TYPEing

This simple line of Macintosh Pascal code defines a type of data that Mac Pascal does not, on its own, know anything about. The type in the example is called "color" and consists of the values shown in parentheses following the equal sign. A program containing this line can then define a variable to be of type "color", as in

```
var
  favorite:color;
```

Any valid value can now be assigned to the variable called Favorite. The valid values are defined in the list following the declaration of color as a type of variable. Thus the following statement is valid:

```
favorite: = purple;
```

The statement would be nonsense to Mac Pascal in the absence of the earlier declaration of color as a new data type, but it's acceptable now. However, the line

```
favorite: = brown;
```

will not be recognized, because brown is not on the list and therefore is not a valid color for the purposes of the program, even though we humans would say it *is* a valid color (and may even be our "favorite").

We can use the Type section to create whole worlds of data types for use in Macintosh programs. This is one of the things that makes Pascal a powerful and useful language. The ability to create new data types gives you the power to make programs in Pascal much easier to read, work with, and learn from than programs written in most other programming languages.

Where'd Types Come From?

Types have been used throughout this book. Until now, however, types have been predefined by Mac Pascal, so we didn't have to declare them.

Thus, when we defined a variable to be of type Integer, we didn't have to stop and write a type definition like this:

```
type
  integer = (-32767..32767);
```

which would define Integer as a data type consisting of all the values between –32767 and +32767. (Do you recognize the use of two values separated by two dots? The last chapter used this format to define the size of an array. Here, it defines the *contents* of the type Integer.)

You can probably think of many ways to use specially defined data types to make programs more readable and useful. Here are a few sample data types. They are unrelated to one another.

```
type
  hit = (single,double,triple,homerun);
  computer = (Apple,Mac,Brandx,Bigblue);
  flower = (daisy,orchid,rose,tulip,fuschia,pansy,bird_of_paradise);
  decimal_digits = (0..9);
```

Using Typed Data in Programs

Three steps are required to use a new data type in a Macintosh Pascal program:

1. Define the new data type in the Type section of the program.
2. Define a variable in the Var section to be of the type we've defined.
3. Assign a value to such a variable and then use that value in a program.

Table 11-1 helps to explain a little more clearly what goes on inside the computer when we define and use a new data type.

Beginners often make the mistake of not taking the second step: defining variables to be of a certain type. They then later attempt to assign a value to the data type itself. For example, having defined a data type called Color, as we did earlier, someone might write the following line — and get a Bug box when the program came to it:

```
color: = blue;
```

Color cannot be assigned a value because it is a data type, not a variable. Attemping to assign a value to a data type is equivalent to trying to do this:

```
integer: = 3;
```

which obviously won't work. Mac Pascal "knows" that an integer is a type of data, not a variable that can be assigned a value. Trying to define a variable named Integer (or, for that matter, Real, String, or any other predefined Macintosh Pascal data type) will produce a Bug box that looks like this:

A type or procedure name has been found where a variable, field name, or value is required.

Types of Types

You've already been introduced to two kinds of data types that we can define in Macintosh Pascal. The first is similar to the numbers separated by two dots inside square brackets used to define the size of an array (see Chapter 10). It is called a *subrange* data type because we define the type to encompass part of a total possible range of the variable. The second is an *enumerated* data type (also referred to in some Pascal books as a

Table 11-1. The Computer and New Data Types

Program says:	*Computer "thinks":*
type day = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);	Here's a new type of data called a "day". It can represent only these seven values.
var payday: day	Here's a new variable. It has a type called Day, which I've seen before. It can, therefore, have any of the seven values allowed by the type called Day.
procedure * * * payday: = Monday; * * *	Here's a variable I already know about called Payday. The program says I should associate the value Monday with the variable Payday. Since Monday is an allowable value for data of type Day, I'll do as I'm told.

"scalar"). In defining this type, we enumerate every possible value for a variable of the type. Color, as defined earlier, is an example of an enumerated data type.

Let's look at these two types in more detail.

Subrange Data Types

A subrange data type restricts a variable's range of possible values to a subset of the range it might otherwise encompass. For example, an Integer variable normally can have any value from –32767 to +32767, but if we wanted to define a variable that would recognize only the values 5 to 15 (which we might want to do, for example, if we were asking the user to select an item between 5 and 15 from a numbered list of options to be pursued by the program), we could set this up with the following line of Mac Pascal code:

```
type
     response = 5..15;
```

This line tells Pascal that only integers from 5 through 15 can be assigned to a variable declared to be of type Response.

Similarly, a variable of type Char normally can be assigned any character value. To restrict a particular type of variable to uppercase characters, we can define it as follows:

```
type
     uppercase = 'A'..'Z';
```

It is not necessary to type the letters intervening between A and Z, since Macintosh Pascal knows all the letters of the alphabet.

Enumerated Data Types

Enumerated data types are often defined to encompass collections of things or attributes to be used in describing variable information, as seen from our earlier example using "color". But that's by no means the end of this type's power. We can do three things with an enumerated data type other than assign one of its values to a variable. We can: (1) find the position of any possible value relative to other values; (2) pick out the next item in the enumerated list; or (3) pick out the preceding item in the enumerated list.

Finding the Order of Data in Types

Let's use a little program example to demonstrate these three uses. Type in the program, Check it, Save it, and Run it, as usual.

```
program DayFinder;
  type
   day = (Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday);
var
   today:day;
begin
Writeln('What day is today?');
Readln(today);
Writeln('Today is ',today,', the #',(ord(today) + 1):1:1,' day of
   the week.');
Writeln('Tomorrow will be ',succ(today));
Writeln('Yesterday was ',pred(today));
end.
```

The built-in Macintosh Pascal function Ord (for "order") returns a number corresponding to the position in the list of enumerated values that belongs to the item we've asked it to find. Thus, if we type TUESDAY, Mac Pascal looks at the data type called Day and finds Tuesday as the third element in the type's enumerated list — so it returns a 2. (Remember, Mac Pascal will ignore upper- and lowercase in this situation.)

You might well be asking why the computer would call Tuesday the second day of the week rather than the third. Computer languages treat the first item in almost all lists as the "zero-th" element, the second as #1, the third as #2, and so on. We simply have to adjust to this and add 1 to the Ord value returned in such cases — which we did.

The Mac Pascal function called Succ (which stands for "successor") returns the item on the list that follows the one we supply in parentheses. Thus Succ(Tuesday) sends Pascal to the data type called Day in order to find Tuesday. It then goes to the next position and declares the next (successor) day to be Wednesday. Similarly, Pred (for *pred*ecessor) finds the item immediately prior to the one we provide, so Pred(Tuesday) comes up Monday.

> **Note**
>
> If you try to use Pred with the first item on an enumerated data list or Succ with the last item on such a list, you'll get a Bug box that tells you that you're trying to supply an out-of-range value, since the first element in an enumerated data list has no predecessor and the last no successor. Our programs, then, must check for this condition before they can do anything fancy with Pred or Succ.

Set 'Em Up!

If you grew up with the "New Math", you have probably already made the jump to the next idea by noticing that enumerated and subrange data types look like mathematical "sets". If you didn't learn New Math, don't be upset; sets really aren't that hard to understand.

Pascal was the first popular computer language to incorporate the idea of a set into its syntax as a particular data type. It provides operators to perform tasks using sets. Since we'll occasionally want to use sets, let's take a quick look at what a set is, and then study a program example that uses sets.

A Set of What?

We nonmathematicians may think of sets as sets of dishes, sets of collectors' postage stamps, sets of encyclopedias, chess sets and the like. These things are sets in a mathematical sense as well — but so are some other things we might not normally think of as "sets". Stripped of its jargon, a set is simply a collection of items dealt with as a group. The items need not have anything in common, though in most practical applications there is some reason for their being grouped together.

Look at the following program segment and try to determine what a set is in Macintosh Pascal:

```
type
  BigLetters = 'A'..'Z';
  BigLetterGroup = set of BigLetters;
var
  BigVowels:BigLetterGroup;
*
*
*
begin
  BigVowels: = ['A','E','I','O','U','Y'];
```

A *set* in Macintosh Pascal defines a data type. The preceding program first defines a data type called BigLetters as consisting of all the uppercase letters of the alphabet. It then defines another data type called BigLetterGroup as consisting of a *set* of the data type called BigLetters. Third, it defines a variable called BigVowels as being of type BigLetterGroup and in turn assigns BigVowels the values it will contain as a set.

This leads to an important rule about using sets as data types in Macintosh Pascal:

> ***We must assign values to a set before we can manipulate its contents.***

Forgetting to assign the letters A, E, I, O, U, and Y to the set variable BigVowels is a common Pascal programming mistake.

Manipulating Sets

The example using the set BigVowels could make a person wonder why anyone would use such a seemingly complex idea as sets. Arrays and other structured data types seem far easier to grasp. The answer is that Macintosh Pascal (along with most other implementations of Pascal) provides a powerful box of tools for comparing, evaluating, and locating information within sets. Since the tools cannot be used with other data types, we need sets to gain access to their power.

Four common tasks that can be performed with sets using simple Macintosh Pascal operators are union, intersection, difference, and membership. Let's look briefly at each of these operations in turn.

Assume that a program defines several sets of people with names and contents as follows:

```
men: = [Ron,Dan,Les,Robert,Mitchell,Sam];
women: = [Carolyn,Meredith,Norma,Sheila,Mary,Christine,Heather];
friends: = [Dan,Robert,Les,Carolyn,Mitchell];
strangers: = [Ron,Sam,Meredith,Norma];
family: = [Dan,Carolyn,Sheila,Mary,Christine,Heather];
```

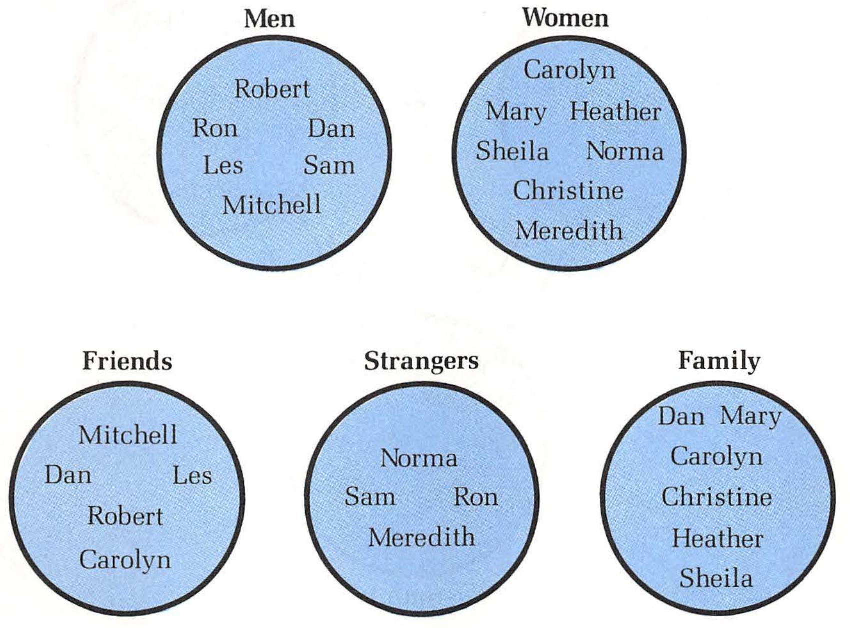
Figure 11-1 depicts this situation graphically so you can see what happens when we perform various operations and comparisons on groups of these sets.

Union of Two Sets

The *union* of two sets is defined as a third set made up of all the elements in *either* (or both) of two other sets. Macintosh Pascal carries out this operation by using the sign for addition between the two sets. If we wrote

```
group1: = friends + family;
```

Figure 11-1. Sets of People as Defined for Example



we'd end up with a set called Group1 containing the following people: Dan, Robert, Les, Mitchell, Carolyn, Sheila, Mary, Christine, and Heather (see Figure 11-2). Dan and Carolyn are found in both sets, but their names appear only once in the new set.

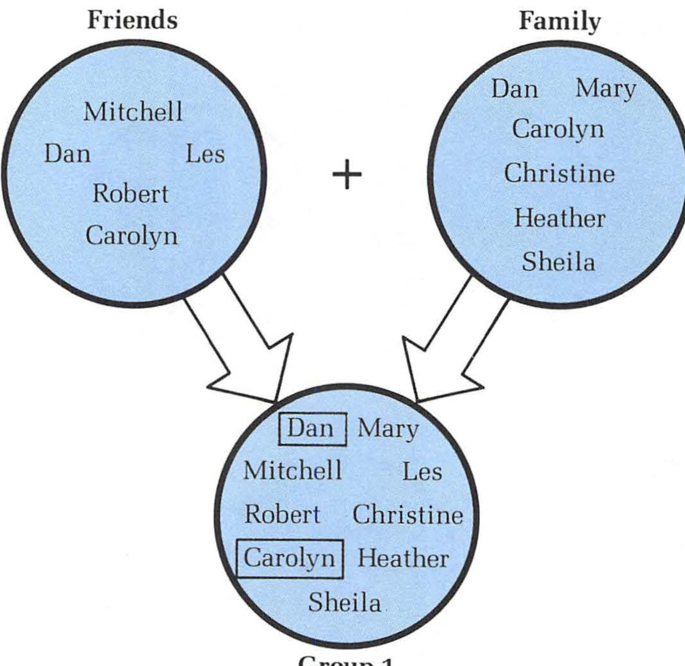
Intersection of Two Sets

The *intersection* of two sets produces a new set containing only items that are members of *both* of the sets being worked with. To calculate the intersection of two sets in Macintosh Pascal, we use the asterisk (*), which happens to be the same operator used for multiplication of numbers. Thus, to define a new set called FamilyFriends containing only people who are both family and friends, we'd write:

```
FamilyFriends:=family*friends;
```

The new set would contain Dan and Carolyn, the only people who are members of both of the other sets (see Figure 11-3).

Figure 11-2. Union of Two Sets of People



Difference of Two Sets

To find everyone who is in one set and not in another, we use the *difference* operator, which logically enough is the minus sign (-). Thus, to list all the men who are not friends, we would define a set as follows:

```
group2: = men-friends;
```

The resulting list would contain the names of Ron and Sam (see Figure 11-4).

Membership in a Set

Perhaps the most powerful thing we can do with a set in Macintosh Pascal is to find out if a particular item is located anywhere in the set. To do this, we use the "in" operator. For example, to find out if Mitchell is a friend, we would write the following line of code:

```
if Mitchell in friends then Writeln ('Yes, he is.') else Writeln
('Nope.');
```

Figure 11-3. Intersection of Two Sets of People

The technique provides an excellent way to check the validity of entries made by the users of our programs and to categorize entries. It is also helpful for other common data processing functions.

A Program for All Seasons

We have introduced the basic ideas of custom-made data types and sets. Now let's see how these two types can be used together in action. Type in the following program, Check it, Save it, and Run it. Observe what happens and analyze the program to find out why those results appear. Then read the discussion that follows.

```
program Seasons;
  type
    month = (JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC);
    monthset = set of month;
  var
summer,winter,fall,spring:monthset;
    thismonth:month;
```

Figure 11-4. Difference of Two Sets of People

```
begin
  summer: = [JUN..AUG];        ← Remember to initialize each set!
  winter: = [DEC,JAN,FEB];      ← Can't use dots; items not adjacent
  fall: = [SEP..NOV];      ← Don't set up spring; it's all that's left
  Write ('Which month?');
  Readln(thismonth);
  if thismonth in summer then
    Writeln(thismonth,' is a summer month.')
  else if thismonth in winter then
    Writeln(thismonth,' is a winter month.')
  else if thismonth in fall then
    Writeln(thismonth,' is a fall month.')
  else     ← Nothing else left for it to be!
    Writeln(thismonth,' is a spring month.');
end.
```

Figure 11-5 shows what a sample run of this program, with DEC as the input month, would look like. (Note, too, that we must type in the month exactly as we have defined it in the program; "December" or "Dec" won't do!)

Figure 11-5. Sample Run of Seasons Program

```
 File  Edit  Search  Run  Windows

Seasons
program Seasons;
  type
    month = (JAN, FEB, MAR, APR, MAY, J
    MONTHSET = set of month;
  var
    summer, winter, fall, spring : montl
    thismonth : MONTH;
begin
  summer := [JUN..AUG];
  winter := [DEC, JAN, FEB];
  fall := [SEP..NOV];
  Write('Which month?');
  Readln(thismonth);
  if thismonth in summer then
    Writeln(thismonth, ' is a summer m
  else if thismonth in winter then
    Writeln(thismonth, ' is a winter mo
  else if thismonth in fall then

Text
Which month?DEC
DEC is a winter month.

Drawing
```

We could make many variations of this program. For example, we could define a new set called FirstHalf to contain the months January through June and then tell the user which half of the year a particular month falls in. Each new example is another demonstration of the great power behind using user-defined data types and sets in Macintosh Pascal programs.

On the Record

There is one problem concerning managing and manipulating data in a Macintosh Pascal Program that we haven't dealt with yet. The problem arises from the necessity of grouping different types of data together into one variable for ease of use and clarity of the finished program. So far, all the data types we've encountered can accommodate only one type of information. Now let's meet the most flexible and powerful of all the structured data types: the record.

In most common data processing applications, data of several different types are mixed into a collection of information that forms a unit. For example, payroll information for an employee might include name (string), address (string), age (integer), number of dependents (integer), wage rate (real), department number (integer), date of birth (string), one-digit retirement code (char or string), and a list of Yes/No answers to health questions (boolean). Each of these types of items *could* be put into, and manipulated in, a separate array, but such an approach would quickly become cumbersome and hard to follow. That's where the "record" data type comes in!

Some Basic Ideas about Records

As far as Pascal is concerned, a *record* is a collection of information of various types, the elements of which usually have something in common. A record might be about a person, a month of sales activity, or an inventory item in a store or warehouse.

A record in turn is made up of one or more *fields* with each field containing a particular type of information about the subject of the record. For example, the record for each item in an inventory record might contain a name field, a quantity-on-hand field, a supplier's name field, a unit price field, and any number of others.

What Field Are You In?

Note here that a field in each record, in a collection of records such as we are now describing, must contain data of the same type as the corresponding field in another record in the same collection. For example, if the first field in an inventory record is the item name, then the first field in any other inventory record in that collection would also have to be the item name field. Thus, once we know the *structure* of any record in a group of records, we can extract desired information from any other record in the group with relative ease — as we'll see in a few moments.

Defining a Record

Macintosh Pascal defines a Record as a data type in the Type declaration section of a program. It "flags" it with the keyword *record*. Since a record will almost always contain more than one field, its definition requires an End; statement to tell Pascal that we are done defining this record type and are ready to move to another data definition step. For example, the payroll data record format discussed earlier might look something like this if it were defined in Mac Pascal:

```
type
  employee = record
    name:string[40];
    address:string[40];
    age:integer;
    NumberOfDependents:integer;
    WageRate:real;
    department:string[5];
    retirement:char;
    smallpox:boolean;
    arthritis:boolean;
end;
```

Figure 11-6 shows how this record might appear in the Mac's memory and on the Mac diskette.

After defining the record called Employee, we could, of course, define further types of data, move into the Var section of the declaration part of the program, or do anything else appropriate to the program creation.

Note that the definitions for Age and NumberOfDependents are on different lines, even though both are integer values. This is our usual

custom in the Var section of a program. It is *permissible* to put adjacent multiple definitions on the same line in a record definition, but it makes for more readable programs to put each field of the record on a different line.

Collections of Records

We usually want to work with many records in a program. Two ways to work with groups of records in Macintosh Pascal include *arrays* and *files*.

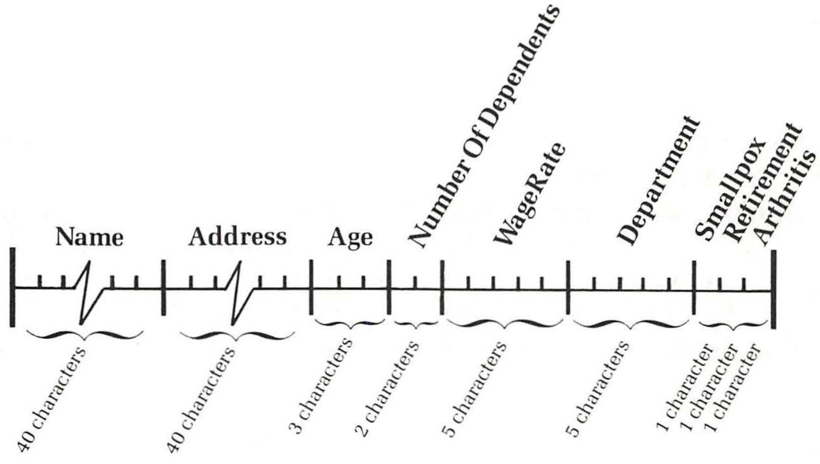
Arrays of Records

A new data type can be defined as a collection of records in an array in exactly the same way that arrays of other predefined data types are defined. You probably recall from Chapter 10 that an array of integers, for example, is defined by a line of Pascal code like this:

```
type
  GroupOfNumbers:array[1..10] of integer;
```

Integer is one of Macintosh Pascal's predefined data types, so it doesn't need to be defined. Before defining an array of a data type *you*

Figure 11-6. Structure of Employee Record



have created, however, remember that you also need to define the data type, something like this:

```
type
  inventory = record
    ItemName:string[45];
    QuantityOnHand:integer;
    UnitPrice:real;
    LowPoint:integer;
  end;
  MasterInventory:array [1..1000] of inventory;
```

These lines create an array called MasterInventory with room for 1,000 records, each of which looks like the layout defined for the record called Inventory appearing just before the array definition. (The computer's memory may not hold an array this large — along with the program, the data, and the rest of the things in there — but that's a practical issue, not a limitation of Macintosh Pascal.)

Files of Records

The data type MasterInventory in our example could also be defined as a "file" of records with the pattern of "inventory". If we do that, we don't need to tell Mac Pascal how many records to provide for. Since the program would work with only one or two records at a time, it would not need to allocate space for 1,000 records (or any other number) at the time the program runs.

As you can probably guess, files of records are kept on the Mac's diskettes for permanent storage and recall. Using such files involves complex tasks. The computer must keep track of a host of details — for example, where records are on the file in relation to one another, and when the end or beginning of the file is reached. Detailed instruction in the use of Macintosh disk files lies beyond the scope of this introductory book, although we provide some basic information in Chapter 13.

Working with Data in Arrays of Records

As we noted with arrays, we need some way to individually address elements of a structured data type individually. For an array we simply provide row and column numbers in square brackets, as in:

```
TestArray[1,3]: = 39;
```

This statement assigns the value 39 to the third item in the first row of a two-dimensional array called TestArray.

Using Field Name for Access

Addressing information in an array of records is similar, except that each field in the record has a *name* associated with it rather than a column number. Thus, when addressing an item in an array of records we use a combination of the item's record number within the array (the approximate equivalent of the "row" in an ordinary array) and the name of the field we wish to examine or change (which we might think of as the "column").

As an example, let's use the following program, which enables us to build a ten-item shopping list, calculates the total of the bill, and lets us look at any one item to see how many we've ordered at what price. Type the program into the Mac, Check it, Save it, and Run it.

```
program Shopper;
 const
   maxitems = 10;          ← Adjust for more or fewer items
  type
    ThingToBuy = record
    item:string[30];
    quantity:integer;
    price:real;
end;
  var
    goodie:integer;
    stuff:string[30];
    ShoppingList:array[1..10] of ThingToBuy;          ← Defines array
 procedure BuildList;
  var
   goodie:integer
 begin
   for goodie: = 1 to maxitems do
     begin
      Write('Name of item? ');
      Readln(ShoppingList[goodie].item);
      Write('Quantity? ');
      Readln(ShoppingList[goodie].quantity);
      Write('Price per unit: $');
      Readln(ShoppingList[goodie].price);
   end;
 end;
```

```
procedure CheckItem {thing:string};
  var
    goodie:integer
begin
  for goodie: = 1 to maxitems do
    begin
      with ShoppingList[goodie] do
        begin
          if item = thing then
          Writeln(item,' ',quantity:1,' $',price:1:2);
        end;
      end;
  end;
procedure TotList;
    var
      count:integer;
      tot:real;
    begin
      tot: = 0;
      for count: = 1 to maxitems do
        tot: = tot + ((ShoppingList[count].quantity)*(ShoppingList
[count].price));
      Writeln('Total shopping funds needed....');
      Writeln('$',tot:1:2);
    end;
begin {main program}
  BuildList;
  TotList;
  repeat
    Writeln('Enter item to review');
    Writeln('or DONE to stop.');
    Readln(stuff);
    CheckItem(stuff);
  until stuff = 'DONE';
end.
```

When the program asks for "name of item", type in an item you'd like to buy (such as a second copy of this book?) up to 30 characters. Press [Return]. The program will then ask for the quantity and the price per unit of the product. After you've provided the information for all ten items, the program will tell you how much the total shopping bill will be. Next, look over part of the list (though you can't change it) by typing in the name of an item you'd like to see again. You can go on doing this until you want to type "DONE".

Figure 11-7 shows a sample run of the Shopper program, which has been changed to permit only *three* items to be entered so that it is easier to see all the items and their price information.

Finding Things in Records

Notice that the procedure called BuildList simply uses a counter called Goodie to cycle through the array of records, asking in turn for each of the three pieces of information we need for each item. The Readln statement is an assignment statement, so the line that says

```
Readln(ShoppingList[goodie].item);
```

puts the user's input into the field called "item" in the "goodie-eth" portion of the ShoppingList array. The period in the statement just before the word *item* separates the array name and record number from the field name. This method of accessing fields within a record array permits us to use field names to assign a variable to a specific part of a record. ShoppingList is defined in the program as an array of the same format as the record called ThingToBuy. ThingToBuy has three fields. Each field can be addressed as we did with the Readln statement above, that is, with the array name and record number followed by a dot followed by the field name.

Figure 11-7. Sample Run of Shopper Program

```
 File  Edit  Search  Run  Windows

Shoppper
  procedure buildlist;
   var
    goodie : integer;
  begin
   for goodie := 1 to maxitems do
    begin
     Write('What item? ');
     Readln(shoppinglist[goodie].item
     Write('Quantity? ');
     Readln(shoppinglist[goodie].quant
     Write('Price per unit: $');
     Readln(shoppinglist[goodie].price
    end;
  end;

  procedure checkitem (thing : strin
   var
```

```
Text
Computer
Quantity? 1
Price per unit: $2100
What item? Mac Pascal Primer
book
Quantity? 3
Price per unit: $19.95
What item? Disks
Quantity? 10
Price per unit: $4.25
Total shopping funds
needed....
$2202.35
Enter item to review
or DONE to stop.
DONE
```

We could use the same approach to recover this item with a line like this:

```
Writeln(ShoppingList[goodie].item);
```

assuming that the value of "goodie" was known and could be supplied to the program.

This is cumbersome, however, if we want to display several fields from one record at one time — as programmers often do. Therefore the designers of the Pascal language provided the With operator.

Using the With Operator for Access

The procedure called CheckItem in the Shopper program also uses indexing that cycles through the ShoppingList array one record at a time. But the fifth line of the procedure begins with a new word, *with*. The small program segment following the With statement checks to see whether "item" in the record currently being examined matches up with the thing for which we are searching. If it does, it prints the information about that item. If not, it goes on to the next record.

The With operator avoids the necessity of coding this segment the long way, which would look like this:

```
begin
  if (ShoppingList[goodie].item = thing) then
Writeln(ShoppingList[goodie].item,ShoppingList[goodie].quantity);
end;
```

The Writeln statement in that form of the program segment is so repetitive and lengthy that it can't even fit on a line. Who needs it? The shorter our program lines, the less likely we are to run out of computer memory or to make silly typing mistakes.

Summary

You can now consider yourself an expert (well, nearly) in dealing with three powerful new kinds of data types in Macintosh Pascal: types of data we create ourselves in the Type section of a program; sets, which allow us to use powerful manipulation tools for comparing, merging, and locating information; and records, which permit us to mix several types of data in one data structure.

We'll see in the next chapter that Macintosh Pascal makes extensive use of records and other data structures in defining its way of handling the mouse and other "events" that have a lot to do with how the Mac works.

Mac-r-cises

1. Using the sets of people in Figure 11-2, determine who, if anyone, will be in the groups of people defined as follows:
   a. men*women
   b. (men + women)-friends
   c. ((friends*women)*men) + strangers
2. In chess, the queen is worth 8 points, the rook 5, knights and bishops 3, the king 2 (some would say 2.5, but we'll keep it simple here), and a pawn 1 point. The abbreviations for these pieces are, respectively: Q, R, N, B, K, and P. Write a program that permits you to find out how many of each piece you and your opponent each have left on the board and then prints the relative point values of your pieces at that point in the game. (Assume you both always have a King; if you didn't, you'd have lost already!)
3. (Difficult) Write a program into which you and your friends could type the name, sex, age, height, weight, eye color, and hair color of up to ten people. Then the program should ask a user what sex, age, height, weight, eye color, and hair color he or she prefers, find the person on file who most closely matches that description, and then print that person's name. "Most closely matching" is defined as having the most traits that match with the input.

12

Of Mice and Events

When you finish this chapter, you'll know

- **How to find out where the mouse is on your display**
- **How to tell when the mouse button has been pressed and released**
- **How to use the Event queue to provide a user interface for your programs**

This chapter will show you how to use the Macintosh mouse in a real-life program that tells where the mouse is and what it's doing. It will also look at the "Event Manager", a feature of the Macintosh that can keep track of what the user of a program has been doing with the keyboard, the mouse, and other parts of the Mac system, so that the program can respond accordingly. These tools are essential to developing meaningful programs that take advantage of the Mac's unique features.

We should note at the outset, however, that processes involving the Event Manager can be handled in other ways than with calls to built-in procedures and functions in Macintosh Pascal. So why bring up the Event Manager at all? There are two reasons. First, knowing how the Event Manager works will help you to understand some of the revolutionary ideas behind the Mac. Second, the Event Manager handles something called an Event *queue*, which is a pipeline of events. The Event Manager therefore lets us look back in time at the last several events that have taken place in the interaction between the Mac and the user, whereas with calls to functions and procedures we can only deal with what is happening now or with what happened last.

Where's the Mouse?

You are already familiar with the built-in Macintosh Pascal procedure called GetMouse. You may recall that we defined two integers, x and y (arbitrarily named) and then used a procedure call like this

```
GetMouse(x,y);
```

to find the vertical and horizontal locations, respectively, of the mouse. In most applications, however, you'll want to use a new data type called *Point* to store information about the mouse location rather than using two separate integers.

What's the Point?

The Point data type can be thought of as a two-part record. It contains two items, the first of which defines the vertical position of the mouse and the second the horizontal. Macintosh Pascal logically labels them v (vertical) and h (horizontal). Their values are accessed in the same way that parts of a record are accessed. The following program demonstrates Point in a visual way. Type it in, Check it, Save it, and Run it, as usual.

```
program LineItUp;
var
  target,position:point;
  found:boolean;
  count:integer;
procedure CheckLocation;
 begin
  if position.v = target.v then
    if position.h = target.h then
        begin
          Sysbeep(5);
          Writeln('You got it!');
          found: = true;
        end        ← No semicolon inside If...Then...Else group remember?
        else
          begin
            Writeln('You're on the right vertical line,');
            Writeln('but your horizontal is off.');
          end;
```

```
  end;
begin {main program}
  found: = false;
  target.v: = 50;
  target.h: = 100;
  PaintRect(100,50,110,60);
  repeat
    while not found do
      begin
        GetMouse(position.v,position.h);
        CheckLocation
      end;
  until button;
end.
```

This program will place a small black rectangle in the Drawing window. Now try to line up the cross hair cursor with the upper left corner of that rectangle. The program will first tell you when the cross hair is properly aligned vertically and then tell you when you've found the corner precisely. Press the mouse button to stop the program.

Figure 12-1 shows a sample run of this program after the mouse has been properly aligned with the upper left corner of the box.

Notice that the program defines two variables — position and target — to be of type Point. During program execution these variables will be accessed often, either to assign them a value or to check and compare them. The values may be accessed in a program by using a construction like this:

```
if position.v = target.v then
```

The ".v" at the end of the variable name means that we want to work with the vertical component of the point here. Similarly, the ".h" added at other places in the program means that we're interested in the horizontal portion of the point in question.

Why use Point variables when programs would be easier to type in if we used x and y? There are at least two reasons. First, it helps to keep programming clear because with variables of type Point, we can't get mixed up as to which variable is the vertical position and which is the horizontal; the coordinates are prelabeled to minimize confusion. Second, since a variable of type Point has two components, we need only two such variables to define the limits of a rectangle or other shape for the Drawing window. Therefore, using points lets us reduce the number of variables we must keep track of in programs.

What's the Mouse Doing in There?

Now that we can *locate* the mouse, let's learn how to figure out what the mouse is doing when we locate it. To do that, we need to know two things about the mouse: First, has the button been pressed? Second, has it been released? Macintosh Pascal supplies two built-in functions, Button and StillDown, to tell us this. These functions return a result of type boolean; that is, they are either true or false.

As you no doubt remember, we can test a boolean value using If, While, Repeat...until, and similar statements. We have made use of such statements as

```
if button then do
```

several times in this book. We can do the same thing with the new function being discussed in this section. First, though, let's be sure we understand the more familiar function, Button.

Figure 12-1. Sample Run of LineItUp

```
 File  Edit  Search  Run  Windows

LineItUp
        writeln('You''re on the right ver
        writeln('but your horizontal is c
      end;
  end;

begin{main program}
  found := false;
  target.v := 50;
  target.h := 100;
  paintrect(100, 50, 110, 60);
  repeat
   while not found do
    begin
     getmouse(position.v, position.h);
     check_location;
    end;
  until button;
end.

Text
You're on the right vertical
line,
but your horizontal is off.
You got it!

Drawing
```

Button, Button, Who's Got . . .?

The built-in Button function won't tell who *has* the button, but it will tell whether the button is being pressed when the function is called. You may have noticed, though, that the button sometimes has to be clicked more than once or else held down a second or two to get the Mac's attention. Here's why this happens.

A function or procedure in Macintosh Pascal works with the state of the system and the program at the moment it is called into action. It tests for the presence or absence of certain conditions or events (for example, the button on the mouse being pressed) virtually instantaneously. If the mouse button is down *at the instant* the function that checks for the button is called, the function will set the value of the boolean variable Button to "true". If the mouse button is clicked a split second after the function has finished checking the button and found it not to be down, however, there is a delay until the function is called again before the mouse button will be interpreted as "down", or pressed. Depending on how much processing goes on between calls to the Button function, as much as several seconds could pass between the time the mouse button is pressed and the time the system reacts.

Is It Still Being Pressed?

At times we will want to determine whether the mouse button is down *again* or *still* being held down by the user. Macintosh Pascal does this with the built-in function, StillDown.

StillDown can be used after a call to Button returns a value of "true". StillDown determines whether the button has been released since the last time it was checked and found down. If the button has *not* been released between checks, the function StillDown returns a value of "true". Otherwise, it returns "false". Pascal programs that require users to drag items around the screen, for example, often use StillDown to ensure that the mouse button is still being held down and that the user, therefore, is still moving the object around the screen. When a call to StillDown turns up a "false", the program will leave the object where the user "dropped" it and move on to other processing tasks.

An Example of Mousing Around

The following program demonstrates StillDown and also provides insight into the way many popular Macintosh programs work. It creates a shape and then lets us drag it around the Drawing window with the mouse

button pressed. Type in the program, Check it, and Save it as usual. Then Run it a few times and read on for a discussion of what the code does.

```
program Dragging;
  var
   origin,newpos:point;
procedure DragIt;
  begin
   if button then
      begin
        While StillDown do
          begin
            GetMouse(newpos.v,newpos.h);
            EraseRect(origin.h,origin.v,origin.h + 20,origin.v + 30);
            FrameRect(newpos.h,newpos.v,newpos.h + 20,newpos.v + 30);
            origin.v: = newpos.v;
            origin.h: = newpos.h;
          end;
      end;
 end;
begin{main program}
  repeat
   GetMouse(origin.v,origin.h);
  until button;
  FrameRect(origin.h,origin.v,origin.h + 20,origin.v + 30);
  if StillDown then
  DragIt;
end.
```

Notice that the procedure called Dragit basically draws and erases rectangles, erasing the one where the mouse was positioned and drawing a new one at the current mouse position. This continues until the user releases the mouse button (that is, while StillDown reports "true"). Note, too, the change in the origin of the rectangle at each new reading of the mouse position.

A click of the mouse button activates the dragging procedure, and a release of the mouse button ends it. This combination of mouse events is used over and over in Macintosh Pascal programs.

The rectangle drawn by the program moves awfully slowly at times. At other times, when the mouse is moved quickly, the rectangle seems to "jump" from one place to another. This behavior is a function of how quickly (or how slowly) Mac Pascal's QuickDraw Graphics routines (described in Chapter 6) execute while keeping track of the mouse and doing other housekeeping things not seen from the outside. It took a lot of skill and understanding to achieve the smooth movement seen in programs like MacPaint and MacDraw.

The Mouse as an Event

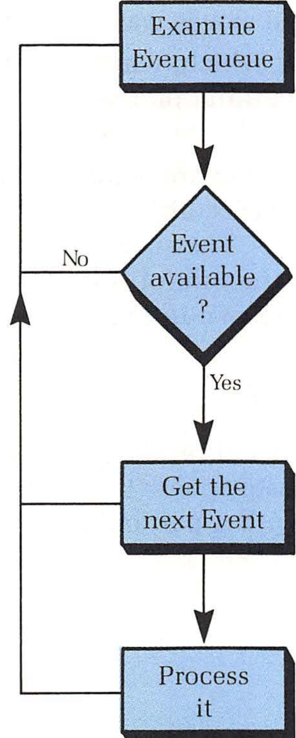
The proper term for things like the mouse button being activated is "event". Other examples of events are a key being pressed on the keyboard or a data item arriving from the disk drive. Let's look now at the way the Macintosh manages events.

The Event Queue

In many respects, the Macintosh is an "event-driven" system. That is, the general flow of interaction between user and computer can be considered as a sort of endless loop that looks something like Figure 12-2.

Various events occurring throughout the system affect the way the Mac responds. The system stores the events in a list called the Event queue. The system then goes through the queue, one event at a time, and takes

Figure 12-2. The Event-Driven Mac



whatever action is appropriate at that point. Events are generally (but not always) processed in the order in which they occur.

What's a Queue?

Simply stated, a queue is a line or a stack. When our British friends stand in line at the grocery store, for example, they refer to the line as a "queue". The term has long been used in computer jargon to mean a line, list, or stack of things stored in some logical order. Mac's Event queue is simply a list of events that have occurred, along with specific and helpful information about each event.

What Is Stored in the Queue?

The system stores the following information in the Event queue for each event that occurs in the course of Mac operations:

1. An *event code*, which can be used to figure out *what* the event was.
2. A *message code*, which provides further clarification of the event.
3. A record of *when* (relatively speaking) the event took place.
4. Data indicating *where* the mouse was when the event occurred.
5. A field of event *modifiers* which, when combined with the event code and the message code, permits programmers to be specific about the precise nature of the event being analyzed or reacted to.

These pieces of information are actually fields in a record of a predefined type called the "EventRecord". Each has a name that can be used in Macintosh Pascal programs. They need only to be declared as variables of type EventRecord. The structure of this record and the names of the various data fields in it are shown in Figure 12-3.

Using the Event Record

From looking at Figure 12-3 and recalling how records are used in Macintosh Pascal, you can probably guess that, before you can do something with the code associated with a particular event, it must be referenced something like this:

```
WhatHappened.where
```

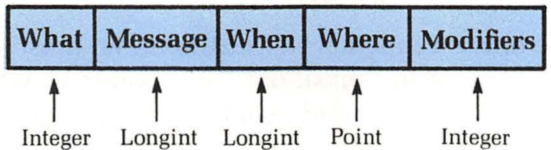
assuming that the variable WhatHappened has been declared to be a variable of type EventRecord. However, since the field Where is declared to be of type Point, it must actually be referenced as *two* integers: WhatHappened.where.v and WhatHappened.where.h.

Getting Information from the Event Queue

The next program shows how to get an event off the Event queue and examine its contents. Type the program into the Mac, Check it, Save it, and Run it.

```
program EventQueue;
 var
  WhatHappened:EventRecord;
procedure GetEvent;
begin
  if GetNextEvent(14,WhatHappened) then          ← See discussion below about "14"
   begin
    Writeln('What: ', WhatHappened.what);
    Writeln('Message: ', WhatHappened.message);
    Writeln('When: ', WhatHappened.when);
    Writeln('Where: ',WhatHappened.where.v,' ',
          WhatHappened.where.h);
    Writeln('Modifiers: ', WhatHappened.modifiers);
    Writeln;
   end;
end;
begin {main program}
 repeat
  GetEvent
 until button;
end.
```

Figure 12-3. Structure of Event Queue Record



How GetNextEvent Works

The procedure GetEvent in the preceding program calls for the built-in Macintosh Pascal function GetNextEvent. It takes two arguments. The first, a "mask", tells the function what kinds of events we are and are not interested in seeing. More about this in a moment. The second is the name of the variable, previously declared as being of type EventRecord, to which information about the event will be assigned.

Experiment with the program. Try pushing various keys and key combinations on the Mac's keyboard. Write down information displayed after the "What:", "Message:", and "Modifiers" lines on the Mac screen. Try to discern a pattern or patterns that can help you figure out what's going on when the Event queue is analyzed. Then go on to read the next section, which describes masks and codes used with the Event Manager.

A Note about the Where

There may seem to be no difference between the values returned by the Where fields where.v and where.h in the event record from the Event queue and the values that would be returned by the function GetMouse. However, as indicated earlier, GetMouse uses the upper left corner of the Drawing window as its base of operations. It labels that point 0,0 and relates all other locations to it. This point is called a "local coordinate".

The Event Manager's Where fields, on the other hand, use "global" coordinates to locate the mouse. The global setting is far larger than the Mac screen. It contains more than *four billion* uniquely identified points, with a horizontal position ranging from –32768 to +32767 (left to right) and a vertical position also ranging from –32768 to +32767 (top to bottom). The center of the global coordinate plane is the 0,0 point.

Coordinates and planes become extremely important when dealing with advanced graphics (beyond the scope of what we've discussed so far). If you keep in mind the fact that GetMouse deals with local coordinates and the Where fields in the Event queue deal with global coordinates, programs that handle both and compare the results won't surprise you by their apparent differences.

Who Is That Masked Event?

When we call the built-in function GetNextEvent, we must supply a "mask" that Mac Pascal can use to "mask out" any events in which we are not interested. The mask is calculated by adding up values assigned to each

type of event with which we have the option of dealing in the Macintosh Event Manager. Table 12-1 shows mask values for some of the most common events; other mask codes can be found in the *Macintosh Pascal Reference Manual* if they are needed.

We tell the Mac which events we are interested in by adding up the values associated with all the events we want information about. For example, if we want to know only about button events, we would use a mask of 6 (4 for mouse button up, 2 for mouse button down; 4+2=6). Similarly, if we are interested only in whether a key has been pressed, we would use a mask of 8. If all three events are of interest, we'd use a mask of 14, and so on. The codes 6, 8, and 14 will be of most interest in the programming we'll do.

What's Happening?

When we run our little program, the event code in the What field displayed on the screen will always have a value between 1 and 15. This code number tells what kind of event is being examined. Table 12-2 gives the codes for the most common of these events; the others are described in the *Macintosh Pascal Reference Manual*.

If we are interested for the moment only in a key being pressed on the Mac's keyboard, we can run a temporary test such as this:

```
if WhatHappened.what = 3 then...
```

This statement examines the contents of the What field in the event record under evaluation and determines whether it's a three. If it is, the

Table 12-1. Event Manager Mask Codes

Code	*Associated Event*
1	Null event (no Event at all)
2	Mouse button down
4	Mouse button up
8	Key pressed on keyboard
16	Key released on keyboard
32	Auto repeating key being pressed on keyboard

program will do what we've indicated we want done when any key is being pressed on the keyboard. Be sure not to confuse these event *codes* with the event *masks* we've discussed earlier.

The Modifiers

The modifier field in the Event queue tells what key combinations are being pressed on the keyboard. If the event being examined is a key press, the Modifiers field will tell us if the key was pressed in combination with [Option], [⌘], [Shift], or [Caps Lock] — or some combination of these keys. This knowledge is important if we want a program to do one thing when the user types in the letter *A*, for example, and something else if A is typed while holding down the [Option] key.

If the What field of the event record being considered by our program is a 3 or a 5, indicating that a key is being pressed on the Mac keyboard (and, in the case of the 5, being held down), then the Message field will contain a code that tells us (if we know how to decode it) specifically *which* key is being pressed. But we still need more information. To prove that, Run the EventQueue program and type in, in succession, the lowercase letter *a*, an uppercase letter A, the [⌘] key along with the lowercase letter *a*, and the [Shift][⌘] A key combination. Write down the What, Message, and Modifiers values for each key combination pressed.

You should see that the What value remains 65 whether you press A or [Shift][⌘] A. Similarly, the lowercase *a*, with or without other keys, always produces the value 97 in the What field. So how can a program tell these combinations apart if it needs to do so?

The answer lies in the Modifiers field. This field is similar to the mask field used with the GetNextEvent function. Its value depends on the

Table 12-2. The Event Code: What's the Event?

Code in What	*Event Described*
0	No Event
1	Mouse button down
2	Mouse button up
3	Key pressed
4	Key released
5	Auto repeating key being pressed

sum of numbers assigned to specific keys in the system. Table 12-3 provides the values assigned to each key condition.

Look at the Modifiers recorded for each key-press combination you tested with the lowercase letter *a*. The *a* by itself produced a 128 in that field, indicating that no other key was being pressed. When you added the [Shift] key, you produced 640, which is the sum of 512 ([Shift]) and 128 (between states, a value always present). The [⌘] A key combination produced a Modifiers field value of 384, which is 128 + 256. And [Shift][⌘] A produced 896, or 128 + 256 + 512.

By studying the code in the Modifier field, then, along with any key press code in the What field, we can determine precisely which keys were pressed to cause the event under consideration. Using the Chr function described in Chapter 7, we can determine which key was pressed in association with [⌘], [Option], or other keys. The information in these two fields permits us to design a program that responds appropriately to whatever input the user furnishes.

A Final Function: EventAvail

We'll conclude our quick survey of the Event Manager's activities with the mention of one more built-in Macintosh Pascal function: EventAvail. This function has the same visible effect as the GetNextEvent function; that is, it places in a variable of type EventRecord the same information that would be placed there by the GetNextEvent function. The difference between the two lies in their effect on the queue itself.

When GetNextEvent finishes executing, the event it retrieved from the queue is removed, leaving the queue with one less event than when the function was called. EventAvail, on the other hand, simply retrieves the

Table 12-3. Modifier Field Values and Their Meanings

Value	*Associated Key*
2048	[Option]
1024	[Caps Lock]
512	[Shift]
256	[⌘]
128	Between states (no other keys pressed)

information, leaving the event itself on the queue. Thus EventAvail can be used to analyze an event without disturbing the queue.

Summary

This chapter has looked at two major components of the user-friendly interface of the Macintosh computer: the mouse and the Event Manager. You've learned how to determine where the mouse is located, whether its button has been pressed, whether the button has been pressed since last checked, and whether it is still being held down. You've learned that the Event Manager is composed of records containing information about the nature and timing of various kinds of events that can cause a Macintosh Pascal program to react. Finally, you've learned how to get information about an event from the Event queue and decode this information so you can use it in making decisions about processing in your Pascal programs.

Mac-r-cises

1. Write a program that permits the user to position the mouse and click the button to cause the program to begin drawing a series of circles starting at that point and radiating outward. Have the program stop drawing circles when the mouse's button is released.
2. To simulate the way the Mac uses the mouse to select items from the Desktop, write a program that: a) places a round-cornered rectangle in the Drawing window; (b) puts some text inside that rectangle; (c) waits for the mouse button to be clicked; (d) if the mouse is inside the rectangle, inverts that rectangle, otherwise simply beeps. (Hint: You'll need to check the point at which the mouse is positioned against the upper and lower limits of the *y* coordinate of the rectangle and against the left and right limits of the x coordinate.)

13

Using the Printer and Disk Files

When you finish this chapter, you'll know

- **How to send information to the printer**
- **How to write text files on the Mac diskette**
- **How to read information from Mac diskettes**

So far this book has treated input to and output from Macintosh Pascal programs as temporary phenomena. Unless we printed the screen with one of Mac Pascal's built-in screen printing functions or took a picture of it with a camera, we lost its contents as soon as we moved on to other things. Of course we typically want more permanence in our work. To get such permanence from Macintosh Pascal, we can do one of two things: print program output on the Imagewriter printer that comes with the Mac and/or save data on a Mac diskette. In this chapter, we're going to learn more about both these processes.

First, a Word about Files

Most of us have had experience working with files. Generally, we think of files as bunches of paper stuffed into manila folders. We may have files of bills, receipts, insurance policies, love letters, and other such things. Businesses traditionally dump things like correspondence with customers, orders, payroll records, inventories, and legal documents into files.

Files in the world of computers are somewhat the same — and somewhat different. Like a paper file, a computer file is often a place in which we store information to which we later will want access. That, for example,

is what a disk file usually is. Mac Pascal, however, also provides a different kind of file, called a "file of text", which performs a special purpose in printing program results on the Imagewriter. We'll get back to that later in the chapter.

What Do Files Contain?

Basically, files contain *records* of information. (And of course you remember all about records from Chapter 11, right?) With the exception of text files, all Macintosh Pascal files contain records, which contain fields, which in turn contain individual items of data to be stored in the file. Figure 13-1 shows this graphically.

In fact, it is mainly in dealing with files that the idea of records becomes useful to a Mac Pascal programmer. Aside from specialized records like those in the Event Manager, most Pascal records are stored on diskettes.

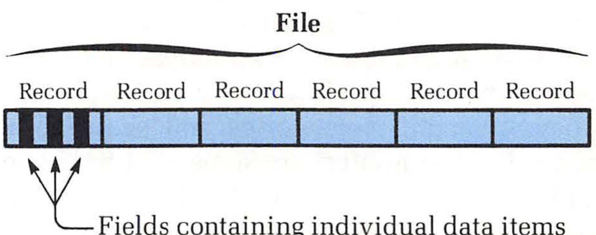
Sending Output to the Printer

So far in this book, everything we've ordered our programs to print has been displayed in the Text window on our Mac screens. But if the boss asks for a report on, say, trends in blue jean sales, we will have some difficulty. Either we'll have to bring the Mac to his office (in which case we may never get to use it again), ask him to come to our office (in which case we may never get to use *it* again), or take a photograph of the screen. There is, needless to say, a better way.

Using the Printer

We have made extensive use of the Write and Writeln statements throughout this book. Mac Pascal has an alternate form of these commands that

Figure 13-1. Data Items, Records, and Files



differs only slightly in format but greatly in results. It turns out that an item at the beginning of the parameter list associated with a Write or Writeln statement can tell Pascal to send the information in the rest of the parameter list to some destination other than the display.

To do this, we define a variable to be of type Text and then use a new command, Rewrite, to set up a file called Printer: This is shown in the following small program:

```
program PrintIt;
var
   file1:text;
begin
  Rewrite(file1,'Printer:');
  Writeln(file1,'This is a test.');
  Writeln(file1,'This is another line of text.');
  close(file1);
end.
```

After typing the program into the Program window, Check, Save, and Run it. Be sure to turn the printer on and select it before running the program, or you'll have problems. Now experiment a little. Change the contents of the messages being printed. Add more lines of message before the "close" statement.

The program really presents two new ideas. The first is the variable type Text and the second is the idea of setting up and using a file. Let's briefly examine each of these ideas in turn.

Texting, Texting

Defining a variable to be of type Text really declares the variable to be a *file* of type Text. In other words, the variable File1 in our test program has to be a file because we said it was a Text variable (not because we named it "File1"). A Text file in Pascal is distinguished from a nontext file, which contains records of structured data types and is more complex to use. This discussion will focus exclusively on text files.

Using a Text File

To use a file of any type, including a file of text, we must first *open* it so that the program we are working with can get at it. That's what the Rewrite statement accomplishes in our sample program. It creates a new file and makes it available for information to be written onto it.

We put information into a file with a Write or Writeln procedure that has, as the first item in parentheses following its call, the name of the file to which we want the information sent. Thus we programmed the line

```
"Writeln(file1,'This is a test.');
```

to tell the program to send the output sentence "This is a test." to the printer, which we previously assigned to "file1".

Once finished with a file, we should always *close* it, so that it will be available for use later in this or other programs. The Close statement accomplishes this task, closing the file named in the argument supplied with the call.

Using Output Formatting

As you know, we can partly control the appearance of information sent to the display screen's Text window by using values for the minimum size of the field and the number of digits to display following the decimal point. We have frequently used lines like this to display a number the way we wanted it:

```
Writeln('Value = $',:2:2);
```

The number following each colon tells the program something about how to display those values. Output to the printer — including strings — can be formatted by the same technique. This makes printing such things as tabular reports quite straightforward. We want our numbers to line up, so we use the colons and numbers for minimum places and decimal places to force them to do so. We can also ensure that text doesn't overrun its assigned place in a report by using a statement like this

```
Writeln(file1,username:40);
```

to tell the printer to confine the variable Username to 40 characters. If it is less than 40 characters, Username will print in its entirety beginning at the leftmost position of the print line. If it is longer than 40 characters, only the first 40 characters will print.

Using Your Imagination

Now that you understand how Macintosh Pascal routes program output to the printer, you should be able to go back through this book, look at the sample programs and Mac-r-cises, and make some of them more useful by having them send their output to the printer. Take a few minutes to do that now, so that when you need a printed list or report for a real-life program, you'll feel comfortable in using the Mac's printer.

Permanent, Recoverable Storage: Disk Files

A distinct advantage of a printout over a screen display is that it is permanent. It has a major drawback, however: once information has been printed out, it cannot be read back into the computer (if, say, we wanted to modify a report we had just written). If the information in the report changes later, we have to go through the whole process of entering the information again.

Storing information on the Mac's diskettes becomes important at this point. Any information a Macintosh Pascal program can gather and place in variables of type Record, File or Text can be sent to a disk for permanent storage. Later, we can recover some or all of this information, vary the sequence in which it is reported, or make other changes in it. Thus we can exercise total control over the data we put into our system. This section will look at the most basic of disk operations: putting information onto the disk and retrieving it to be routed to the printer. We'll work strictly with text files; manipulating data in structured data types is the subject of Chapter 15.

A Sample Disk Writer

In this section, we'll look at two programs. One will *read* information *from* a text file, and the other will *write* information *into* a text file.

The following simple program writes information into a text file. It teaches all the basic principles you'll need for even complex disk file manipulation. Type it in, Check, Save, and Run it. When the program asks for three lines of text, you can type in anything you like, no matter how long or short, or whether it contains punctuation, numbers or anything else. At the end of each line, press the [Return] key.

```
program Diskwriter;
 var
  file1:text;
  test1,test2,test3:string;
begin
 Rewrite(file1,'textfile1');
 Writeln('Enter line 1:');
 Readln(test1);
 Writeln('Enter line 2:');
 Readln(test2);
 Writeln('Enter line 3:');
 Readln(test3);
 Writeln(file1,test1);
 Writeln(file1,test2);
 Writeln(file1,test3);
 close(file1);
end.
```

Note the strong similarities between writing to a disk file and writing to the printer. In fact, from Macintosh Pascal's perspective, there is no difference except that the printer has a predefined file name, "printer:", while a disk file requires you to give it a name. The program defines a file (Text) variable called File1. The first line in the main program block associates it with a disk file named TextFile1. From that point on, output routed to File1 goes to the disk file TextFile1. If you're skeptical about this, Quit from Mac Pascal and notice in the File window on the main Mac desktop a new file called Textfile1 which wasn't there before; you've just created it!

How to Write Data to the Disk

Note that we used three separate Writeln statements to write three lines of information to the disk. This permits us to retrieve the lines individually later. The Writeln statement causes Mac Pascal to put a carriage return and a line feed — a combination of characters that it will later treat as marking the end of a line in the file — at the end of each line as the line is written to the disk. We could have written, instead,

```
Writeln(file1,test1,test2,test3);
```

This is a perfectly valid Macintosh Pascal statement, but it would have made all the lines run together into one line on the file. That would make

retrieval in a usable format more difficult. It is also legal to use the Write statements instead of Writeln, but retrieval of data will then require more thought and planning.

Closing Files

Note, finally, that the Diskwriter program, like the PrintIt program you saw earlier, ends with a statement closing the file(s) with which the program has been working. *This is very important.* If we fail to close a file and then try to use it again during the same session, we'll get a lot of messages about files that don't exist or are busy. Worse yet, Macintosh Pascal won't be able to identify the end of a file and will therefore start printing gibberish on the printer or screen. *Make it a habit always to close all files at the end of processing.*

Getting Information Off the Disk

Now that we have all that information stored on the Mac disk, how do we get it back to make some use of it? That process requires three steps: opening the file for output, reading all the information from the file but stopping when the end of the file is reached, and closing the file. Here's a small program that does this for our newly created file, TextFile1. Type it in, Check, Save, and Run it. Be sure the printer is turned on before you begin.

```
program DiskRead;
 var
  file1,list:text;
  test:string;
begin
  reset(file1,'textfile1');
  Rewrite(list,'printer:');
  while not eof(file1) do
    begin
     Readln(file1,test);
     Writeln(list,test);
  end;
 page(list);
 close(file1);
 close(list);
end.
```

There are some new statements here. Let's look at them to gain a better understanding of this program.

What Does Reset Do?

The Macintosh Pascal statement called Reset appears as the first line in the main block of our sample program. This statement opens a file to be used as a source of input to our program. It is similar to Rewrite, which we've discussed previously. Rewrite opens a file to be used for *output*, while Reset opens an existing file to be used as an *input* file.

Using Reset with a file that does not exist will produce a Bug box. If we use Rewrite with a file that does not exist, however, Mac Pascal will simply create one by that name. In fact, the Rewrite statement creates a new file even if one with the name we supply already exists, so we have to be careful how we use that statement or we may erase existing files.

Checking for End of File

The line that begins with While is a bit cryptic, so let's take a closer look at it. The new part of the statement is the term "eof(file1)". "EOF" is a standard computer abbreviation for End Of File. We always have to tell "EOF" what file we are concerned with; in this case, we don't want to read information past the end of the file called TextFile1, which is assigned by the program to file1. We tell Mac Pascal to read and report information *only* until it finds the end of this file.

Checking for the EOF is very important because of the way files are sometimes organized on the disk. Our name and address file, for example, might be right next to this week's shopping list. If we don't watch out, we'll end up with spaghetti listed among our friends! (Actually, this couldn't happen; we'd get an error notifying us of an attempt to read past the end of file.)

Our sample program reads a line from the file called TextFile1 (File1), prints it by means of a Writeln statement that uses the printer file variable called List, and then gets the next line from the file. It goes on doing this until it reaches the end-of-file marker, at which point it stops processing.

Cleaning Up

After completing this reading and printing, our program still has two small housekeeping tasks to do. It has to close the files, as indicated earlier. It should also cause the page it's been printing to be ejected so that the printer will be ready with a clean piece of paper for its next job.

Files are closed with two separate and explicit statements, as you can see. A third statement,

```
page(list);
```

causes a page to be ejected by the printer (sometimes called form feed).

Routing Output to Screen Instead

Sometimes we may send information to the printer and then change our minds and decide that we want it on the display screen instead. During program debugging, for example, we don't want to waste reams of paper on printing the wrong information. How, then, can we change a program, such as our sample program, to write information to the screen instead of to the printer? It's very easy. We simply remove all references to the file called List so that, for example, the statement Writeln (list, test); would now read:

```
Writeln(test);
```

We may remove the Rewrite statement as well, but it's not necessary. It doesn't do any harm to rewrite a file and then not use it.

More Complex Disk Operations

In this chapter, we've purposely confined ourselves to text files, not files using numbers to perform calculations or any structured data types such as those which make up most useful data processing records. You can get a lot of mileage out of knowing just this much about disk files, but you'll almost certainly want to learn to manipulate the more complex data structures discussed in Chapter 15 before you attempt any serious programming assignments.

Summary

In order for computers to be truly useful, we need to make permanent or semi-permanent records of information they produce. In this chapter, you've learned how to do three things to facilitate this aspect of your Macintosh Pascal programming: to send program results to the printer, to put information on a diskette for later retrieval, and to get that information off the diskette and into the computer again.

For another example of the use of disk files, you can study the program DumpFile in Chapter 17.

Mac-r-cises

We won't provide formal Mac-r-cises for this or the following chapters. By now, ideas for ways to use the Mac and its power should be pouring out of your head without any help from us. We've provided the skeletons of some useful disk programs in this chapter; use them to experiment with different kinds and amounts of data. Take control of the Mac and of the Mac-r-cises. After all, personal control is one of the things the Mac is all about!

14

Some Advanced Graphics Ideas

When you finish this chapter, you'll know

- **How to hide, show, and obscure the cursor**
- **How to resize, relocate, and manage your Text and Drawing windows**
- **How to animate, resize, and manipulate rectangles inside the Drawing window**

Chapter 6 covered the basics of using QuickDraw routines from Macintosh Pascal to create and manipulate graphics in the Drawing window. There is, of course, much more than this to be done with graphics on a machine as graphics-oriented as the Macintosh. This chapter will explore three additional graphics concepts that can further enrich your programs and give you a deeper appreciation for the power of graphics in Mac Pascal. Three new tools will be supplied to your kit bag of techniques to help your Mac programs take on the peculiar quality programmers refer to as "Mac-like".

These three new tools are cursor manipulation, window management, and sizing of rectangles within the Drawing window.

Cursor Manipulation

The cursor moves across the screen when we move the Mac's mouse. Normally, we refer to it as a pointer, but the Macintosh commands that manipulate it are called "cursor commands", so we'll use the two terms interchangeably for the next few pages of our discussion. (No one seems

to know where the term *cursor* originated, but it has been widely denied that it was named after the colorful language that poured forth from the mouth of the first programmer who tried to write a Pascal program on a computer screen.)

The cursor takes on different shapes in different Macintosh applications. It is most often a bold arrow pointing approximately northwest. When a Macintosh Pascal program is running, the cursor turns into a crosshair. In MacWrite, it is often a very thin vertical bar, while in MacPaint it may take on myriad shapes, sizes, and patterns. The cursor's shape is controlled by whatever program is running on the machine.

The cursor also seems to go away from time to time, only to return to the same spot the next time the mouse is moved. You can easily learn how to hide, obscure, and show the cursor on the Mac's screen.

Where'd It Go?

Let's have a little fun manipulating the cursor. Pull down the Instant window and type in the following line of Mac Pascal code. Then click on the Do It button and watch the arrow go away. *Be sure to leave the cursor on the "Do It" button after clicking there*, otherwise getting the cursor back is going to take some real gymnastics. Here's the code:

```
HideCursor;
```

Logically enough, this command (it's actually a procedure call) causes the cursor to disappear. We can make it reappear with the ShowCursor command. To demonstrate this command, *leave the mouse undisturbed* and type the following line into the Instant window beneath the previous line. Then click on the Do It button again (even though you can't see the arrow click in anything at this point!).

```
ShowCursor;
```

What happened? Nothing, probably. The ShowCursor command seems not to have worked — and, in fact, it didn't, for a reason that will become clear in a moment. For now, let's get out of this mess. Still *leaving the mouse in its proper place to click the Do It button*, type in the ShowCursor command two more times so that the Instant window contains the following sequence. (The top line or two may be invisible without scrolling, which we urge you *not* to do until we have the cursor safely back with us.)

```
HideCursor;
ShowCursor;
ShowCursor;
ShowCursor;
```

Now click on the Do It button, and *voilà!* the cursor magically reappears.

Balancing the Hide and Show

You've just witnessed a very important Macintosh Pascal rule in action. That rule should now be emblazoned in your mind forever.

> ***Every HideCursor procedure call* must *be balanced by a corresponding ShowCursor procedure call.***

This is similar to the kind of balance you've learned to supply in the case of Begin and End statements.

But why did we seemingly need three Shows to balance one Hide? Well, it's like this. . . .Our little Instant window experiment first ran the HideCursor procedure by itself (that's one). Then we put the ShowCursor command in and used the Do It button to run again. Because we still had the HideCursor command in the Instant window, it executed again (that's twice), while ShowCursor ran only once. So we put in *three* more ShowCursor calls. Why? Because if we'd used only two — as might at first seem right — and *then* clicked the Do It button, we'd have called HideCursor for a *third* time while we had only two balancing ShowCursor calls, so the cursor would have stayed hidden. In other words, three clicks on the Do It button of the Instant window called the HideCursor routine three times, so we needed three ShowCursor calls to balance them.

Just Go Away for a Moment

To hide the cursor temporarily and then get it back without issuing an explicit command, you can use the ObscureCursor procedure call. This procedure hides the cursor but only until the next time the mouse is moved or a button is clicked. Then the cursor reappears. Macintosh Pascal uses a lot of ObscureCursor calls; this is why the arrow disappears when we are editing programs in the Program window, for example.

Changing the Shape of the Cursor

It is possible to alter the shape of the cursor in Macintosh Pascal by using special codes to describe the desired shape and then calling the SetCursor procedure. The details of doing this are beyond the scope of this book. If you do change the shape of the cursor, you can get the northwest arrow back with the InitCursor command.

Window Pains

The three windows with which we've started each of our Macintosh Pascal sessions — the Program window, Drawing window, and Text window — can be maneuvered by our Mac Pascal programs. We have suggested frequently, for example, that before Running a program you click on the Drawing window's vanishing box and enlarge the Text window to accommodate more information on the screen. These things can all be done *from within Mac Pascal*.

Clearing the Desk for Action

One of the easiest window-managing commands to use is the one that simply removes all three windows from the Macintosh Pascal desktop. The command is written in programs like this:

```
HideAll;
```

This command, which we've seen once before without discussing it, leaves only the Menu bar and a clean desktop. Pull down the Instant window, type in the HideAll statement, and click on the Do It button to prove that this is the way it works.

Restoring Windows Using the Menu Bar

Now what? You can't program, and, if you could, you couldn't show the results anywhere but on the printer. Gotcha!

There are two ways to restore windows after they've been erased from the desktop. The first is to use the Windows option on the Menu bar, pull down the name of the window you want to restore, and release the button. The window will then reappear. Click on the Windows option on the Menu bar and observe that all three of the windows are listed there.

By the way, this is the only way to restore the Program window once we cause it to disappear. The other two windows, as you'll see in a moment, are accessible through Macintosh Pascal statements, but the Program window is not.

Restoring Windows in a Program

If you've caused the Text or Drawing window to vanish by using the HideAll procedure or some other means, you can use a Macintosh Pascal statement to restore either window to its original visible position. Again using the Instant window (clear it first of its previous contents, if any), type the following:

```
HideAll;
ShowDrawing;
```

Now click on the Do It button and notice what happens. You can do the same thing with the Text window by substituting ShowText for ShowDrawing.

These commands are sometimes handy. If you have written a program that uses the Drawing window only, for example, you can make the display much easier for the user to follow — and, incidentally, save on Mac memory space — by including the following lines in your program:

```
HideAll;
ShowDrawing;
{other program lines that use the Drawing window go here}
ShowText;        ← Restores things to original condition
```

Resizing and Positioning Windows

Sometimes you may want your programs to make the Drawing window much larger than its usual size. You may also want to move it on the screen so that it dominates the desktop more. Similarly, if you are producing output in the Text window from a disk file, you may want the Text window to be larger and more central or dominant. Macintosh Pascal provides a group of statements designed to facilitate these needs. We'll describe the ones for the Drawing window and let you experiment with the parallel ones for the Text window on your own.

The following program will enlarge the current Drawing window, relocating it in the process to occupy most of the screen. When the [Return] key is pressed, the Drawing window will return to its original size and position. Type the program in and Check it; there's no need to Save it unless you want to do so.

```
program MoveWindow;
 var
  OldDrawingRect,NewDrawingRect:rect;
  ans:string;
begin
  GetDrawingRect(OldDrawingRect);
  HideAll;
  NewDrawingRect.top: = 35;
  NewDrawingRect.left: = 10;
  NewDrawingRect.right: = 410;
  NewDrawingRect.bottom: = 335;
  SetDrawingRect(NewDrawingRect);
  ShowDrawing;
  Writeln('Press RETURN to go back to the original.');
  Readln(ans);
  SetDrawingRect(OldDrawingRect);
  ShowDrawing;
  ShowText;
end.
```

Now let's take a closer look at the new material presented in this sample program.

Saving the Original Size First

In order to be able to restore the Macintosh Pascal desktop to normal size when a program is finished, you need to save the original size and position of any windows that the program will move. This requires two steps: defining a variable to hold specifications for the current window and asking Mac Pascal to put that information into this variable for later use.

Since windows are always rectangles, their size and position on the screen are defined by the methods for handling rectangles described in Chapter 6. We define a variable either to be of type Rect or to contain four integer values that we then associate with the top, left, right, and bottom sides of the rectangle. We'll use variables of type Rect throughout this discussion.

Our sample program uses the Mac Pascal statement GetDrawingRect and assigns the current values of the Drawing window's location coordinates to the variable OldDrawingRect. The values will be available there at the end of the program, as we'll soon see.

Setting Up the New Location and Size

Next, we define the new size and location for the Drawing window using two sets of statements. We first assign each corner — top, left, right, and bottom — to the appropriate components of the variable of type Rectangle that we've defined to hold the new size and location. In this case, it's NewDrawingRect. This process involves a simple series of assignment statements.

Now we use the SetDrawingRect procedure to *set* the Drawing window to have these coordinates we've given our new variable. Finally, we order Mac Pascal to show us the new window, using the ShowDrawing procedure.

Restoring the Original Drawing Window

At the end of many programs, users want to go back to the main Mac desktop rather than to the program with its associated windows. Sometimes, however, you'll want a program to *restore* all windows to their original condition before going back to the main desktop. To do this, simply use the SetDrawingRect statement with the information about the size and position of the Drawing window that you saved at the outset of your work, then use ShowDrawing to redisplay the Drawing window in its original place and condition.

(Note that a Writeln statement in our program asks the user to press [Return] to go back to the original window. Note, also, that this line doesn't display, since there is no Text window for it to appear in. We did this to demonstrate that even when a window is not displayed, we can still *use* it; it just doesn't do much good.)

Moving and Resizing Rectangles

The ideas behind graphics on the Macintosh are very closely connected to rectangles. If you can learn to manipulate rectangles with ease and flexibility, you'll have gone a long way toward mastering even complex graphics on the Mac.

This section will introduce two new commands that permit us to do interesting things with rectangles. One of them moves rectangles around, and the other resizes them. We'll work with them in the Drawing window, but the principles described can be used equally well with the other windows.

The Incredible Moving Rectangle

The first procedure can be demonstrated easily. Just type it into the Program window, Save it, and Run it:

```
program OffsetShow;
 var
  MyRect:rect;
  count,z:integer;
 begin
  SetRect(MyRect,10,10,100,100);
  FrameRect(MyRect);
 repeat
  EraseRect(MyRect);
  OffsetRect(MyRect,1,2);        ← New procedure; discussed in text
  FrameRect(MyRect);
  for count: = 1 to 100 do
   z: = z + 1; {delay for a few moments so we can see what's going on}
 until button;
end.
```

When we run this program, a rectangle is formed in the Drawing window. After a few moments, it begins to move down and to the right at a nice, even pace. It continues until you press the button on the mouse to tell it to stop.

Note the new procedure called OffsetRect. In our example, it has three parameters: the name of the rectangle to be worked with (in this case, MyRect), the horizontal offset, and the vertical offset. The horizontal offset tells the Mac how far to relocate the rectangle horizontally. Similarly, the vertical offset tells the Mac how to reposition the rectangle vertically. Our example moves the rectangle one position to the right and two positions down each time through the loop.

A simple loop will move the rectangle smoothly in either small or large increments.

The Incredible Shrinking Rectangle

Besides repositioning a rectangle inside the Drawing window, we can grow or shrink them, again by using a loop. Observe the following program (and Run if you like):

```
program ShrinkingBox;
 var
  MyRect:rect;
  count,z:integer;
begin
  SetRect(MyRect,10,10,190,190);
  repeat
    FrameRect(MyRect);
    EraseRect(MyRect);
    InsetRect(MyRect,10,10);        ← New line; see text
    FrameRect(MyRect);
  for count: = 1 to 100 do
       z: = z + 1;
  until EmptyRect(MyRect);         ← New function EmptyRect; see text
end.
```

The program produces a shrinking box something like the one we encounter when we close a file on the Mac desktop. The shrinking effect is produced by a new statement, InsetRect. Like OffsetRect, this statement takes three parameters: the name of the rectangle to be reshaped, the horizontal shrinkage or expansion amount, and the vertical shrinkage or expansion amount. Every time InsetRect is executed and followed by FrameRect(myrect), the program draws another rectangle with the new specifications.

Note, too, the new function called EmptyRect. When called, it checks the rectangle to which it is applied to see if there are any spaces in it. If there are not, the rectangle is no longer visible, and EmptyRect returns a value of true. In this case, a value of True for EmptyRect stops the loop and the program.

Summary

In this chapter, you have learned how to hide, show, and obscure the cursor to make your desktop cleaner and easier to use. You've also learned how to make the Drawing, Text, and Programming windows disappear from

the desktop, and how to resize and reposition them. (For examples of using windows in larger programs, see Chapter 17.) Finally, you've learned two new procedures, OffsetRect and InsetRect, that can animate, resize, and manipulate rectangles in the Drawing window.

Mac-r-cises

There are no specific assignments for this chapter. You might, however, try using the OffsetRect and InsetRect procedures as starting points for graphics programs, combining them with things like PaintRect, InverseRect and FillRect to create startling visual effects. What happens to these commands and your use of coordinates as the Drawing window is moved around or resized with the window manipulation commands you learned in this chapter?

15

Introduction to Dynamic Data Structures

When you finish this chapter, you'll know

- **The meaning of "dynamic data" and "static data"**
- **How to use the built-in procedure "New"**
- **What pointer variables are and how to use them**
- **The meaning of a pointer value of "nil"**
- **The use and value of linked lists**

In this chapter, we'll discuss one drawback to all of the data structures we've encountered so far, and then we'll describe Macintosh Pascal's solution to that problem. When you've finished this discussion, you should be prepared to tackle just about any kind of data-handling assignment you might encounter.

Dynamic Data

All the types of data structure we've dealt with previously have had one common trait: they required us to know *in advance* how many items of information we were going to put into the structure. All lists and arrays are like that; for example, we define an array as having 32 possible integer values by writing:

```
array1:array[0..31] of integer;
```

If that array turns out to need only 15 variables, we're safe enough, although we've wasted the computer memory space that was set aside for the 17 unused variable values; but if we suddenly needed 33 or 50 or 120 values in this array, we'd have to revise the program.

Of course, in daily life it is fairly common to have variable amounts of data to work with. For example, if you were running a bookstore, you might have some very slow days, on which there were only a few transactions, and some very busy holiday shopping days, on which there might be hundreds of transactions. Now suppose you want to write a Macintosh Pascal program to record and analyze all of these transactions. Using what you know so far, you'd have to define a variable array to have enough "slots" to handle the busiest day you'd ever expect to have. This would make very inefficient use of the computer's memory, since almost every day's record would contain some wasted space. Alternatively, you could set up your variables for an average day and then, on busy days, call in the programmer to reprogram the computer for the larger number of transactions! What's needed is an expandable data structure which can take on different sizes, depending on how much data it has to hold.

Dynamic vs. Static

To solve this problem, you need to understand the difference between *dynamic* data and *static* data. The number of players on a baseball team, for example, is static: it's always nine. The number of hits, runs, errors, pitches, hot dogs, and beers involved in a baseball game, on the other hand, is never known in advance; such data is referred to as dynamic data.

In this book, we've often dealt with data that was really dynamic but treated it as if it were static at some arbitrary level. Thus, we designed a golf score program that was limited to ten rounds and a shopping list recorder that might not handle enough items for the next trip to our Macintosh dealer. We had to do that because introducing the idea of dynamic data structures would have been confusing until we had thoroughly discussed data structures in general. Now, though, you're ready for dynamic data structures.

When to Use Dynamic Variables

The dynamic variables you'll learn about in this chapter should be used any time you *don't know in advance* how many pieces of information of a particular kind you're going to have to deal with in a program. Because

of this element of uncertainty, programming using dynamic data structures tends to be a bit more cumbersome and complex than programming with simple static data structures. Therefore, it's not a good idea to use dynamic data structures if you don't have to, even though they would provide added flexibility.

A dynamic data structure differs from an array of a similar type of static data in the following key ways:

1. It has no fixed maximum length.
2. Its items are not referenced using subscripting.
3. There is no predefined data type name for such a structure in Macintosh Pascal.

An Example Program

The short example program we'll examine in this section will, at first glance, seem to prove very little about dynamic data structures. But hang in there, and we think you'll get the idea we're trying to explain. Type in the program, Check it, Save it, and Run it. Then see if you can figure out intuitively what's happening in it. (Please note that the little character that looks like a peaked hat is called a *circumflex*. It is formed by using the uppercase character on the number 6 key on the Mac keyboard. Be careful not to miss this character when it appears in any program line, because it is critical to the meaning of the program.) The screen display you end up with should resemble Figure 15-1.

```
program Dynamic1;
 var
  p1,p2:^char;
 begin
  New(p1);
  p1^:='A';
  p2:=p1;
  Writeln(p2^);
  p1^:='B';
  Writeln(p2^);
 end;
end.
```

You can certainly be forgiven if you find this program more than a little puzzling. In the main body of the program, we assign a value, "A", to what appears to be a variable with a strange-looking name, "p1^". We make the variable p2 take on the value of the variable p1, which it doesn't

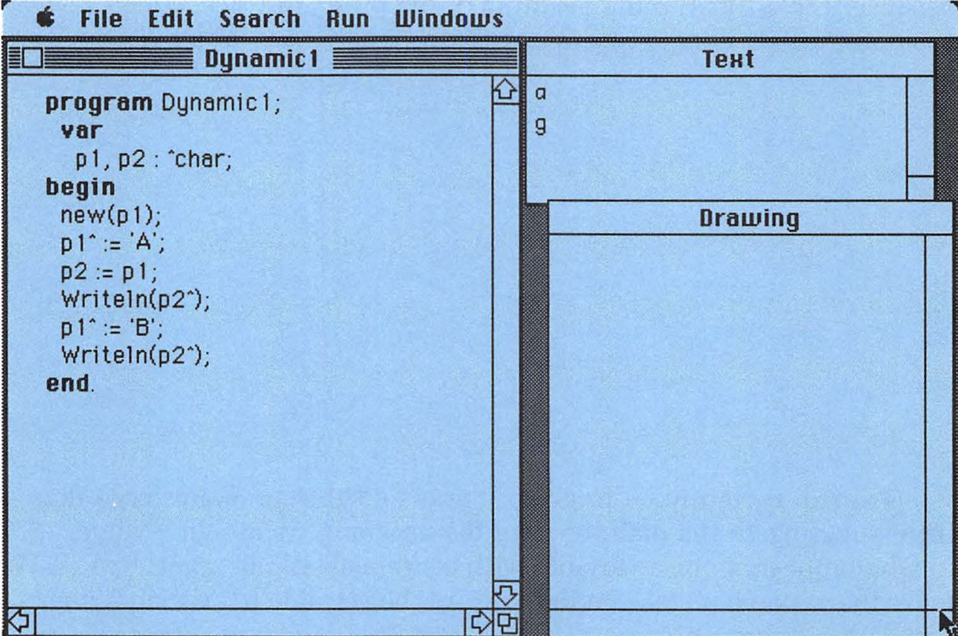
appear we've assigned a value to at all yet, and then we change the value of the first variable, p1^, to "B". Finally, we print out the value of p2^ and find that it has the value of p1^, even though we don't see any place where we have assigned that value to that variable. What's going on here?!

To understand what's happening and how it relates to dynamic data structures, let's slow down and take a look at two new ideas introduced in this program: creating dynamic data variables with the built-in procedure called New, and using the "pointers" that result from such creation.

Setting Up Dynamic Variables

The first new concept in the Dynamic1 program occurs in the Var declaration section, where we find two variables, p1 and p2, defined as being of type ^char, a type we've never encountered before. The circumflex that precedes the familiar data type Char is read as "pointer to". Thus we have defined two variables to be of type "pointer to character". We'll see in a few moments what that means.

Figure 15-1. Results of Running Dynamic1 Program



The Procedure "New"

The built-in Pascal procedure called "New" requires a single argument, the name of the pointer variable that points to the area in memory where the new record will be stored. In our sample program, the statement

```
New(p1);
```

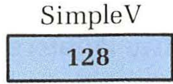
allocates space to a new variable *pointed to* by the pointer variable p1, which we have just defined. Note that this is *not* the same thing as a variable *named* p1. If you try to print or manipulate a variable called p1 inside the sample program, you'll find out that it doesn't work.

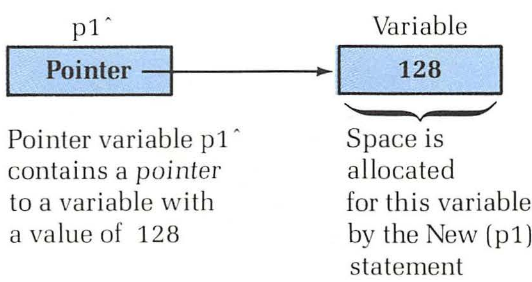
Figure 15-2 shows the difference between a simple variable, which *contains* a value, and a pointer variable, which *points* to a variable containing a value.

When we use the New procedure, we allocate space to a new variable that doesn't need a name since it is referenced, not by name or by indexing position in an array, but by the pointer defined for it. This approach to variable referencing is the key to allocating space for dynamic data structures.

Looking again at our sample program, we find that the next line allocates the value "A" to the variable pointer p1^. This causes the pointer

Figure 15-2. Simple vs. Pointer Variables





to point to a variable whose contents are now the character "A" (see Figure 15-3). The next line establishes the other pointer variable we've set up, p2, to point to the same variable *location* as the pointer variable p1. To prove this, we have the next statement

```
Writeln(p2^);
```

which dutifully produces the letter "A" on the display as the program runs. In other words, we have defined the pointer p2 so that it points to the same memory location as the pointer p1, and since we stored an "A" in that location a few moments ago, p2 now points to "A". Hardly surprising, right?

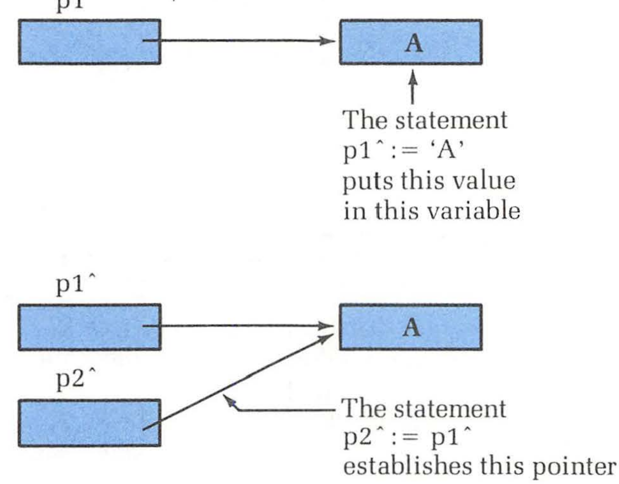
So far, we could have done the same thing with ordinary variables by assigning the same value to two different variables or by assigning the value of one variable to the value of another. The next line is the tricky part.

Manipulating Pointers

This line changes the value of the variable p1, while apparently leaving p2 alone. The line simply says:

```
p1^:='B';
```

Figure 15-3. Two Pointers to One Variable



It says nothing about p2. But when we print the value of p2^ on the next line, it, too, has the value "B". How come?

Since we set the pointer p2 to be the same value as the pointer p1, any value we place at the location pointed to by p1 will also automatically be the value pointed to by p2. Figure 15-4 should help to explain this idea.

A memory location is simply a physical space in the computer's memory area in which we can store a value. The memory location pointed to by the two variables in Figure 15-4 doesn't have any built-in value. We can "stuff" any value into that memory location with an assignment statement and that value will be pointed to by *both* p2 and p1.

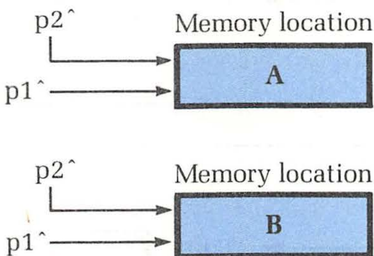
Each memory location is associated with a particular number, called its address. (You may remember that items in arrays had addresses, too. You could think of the Mac's memory as a very large array with one index.) Each such address is unique. The pointers we've been discussing are actually memory addresses in disguise. Figure 15-5, which is a reconstruction of Figure 15-2 using memory addresses, should help to clarify this point.

Accessing Dynamic Variables

Assigning a value to a dynamic variable, then, or reading data from a location pointed to by such a variable, can be done easily by using the pointer techniques shown in our sample program. Notice that the circumflex *precedes* the type of data to which the pointer is to point when we define the pointer, but it *follows* the name of the pointer when we wish to access the data pointed to by it. Writing a program statement like this

```
Writeln(^ p2);
```

Figure 15-4. Pointers and Variables



will produce a Bug box because the variable "^p2" makes no sense to Macintosh Pascal.

Limits of Dynamic Variables

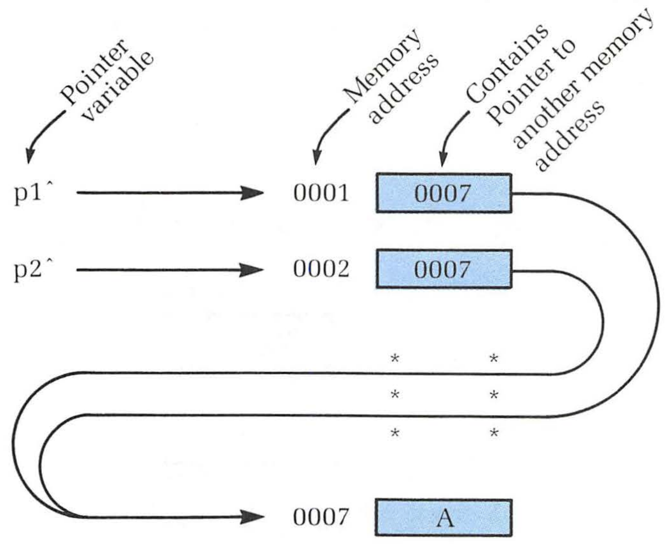
If we were going to work only with arrays or individual data items, we'd have little or no use for dynamic variables in most of our Mac Pascal programming. But the real strength of dynamic variables becomes apparent when we begin discussing linked lists.

Use of Linked Lists

A "linked list" is the most basic form of dynamic data structure. In essence, such a list consists of a sequence of items, each of which is linked to the item that follows it in the sequence. This linking is done by means of a linking field within the data item itself. This field contains the address of the next item in the list. In other words, it's a pointer. Figure 15-6 shows a simplified linked list in which the address at which each item is stored is shown on the left, followed by the two-field item itself.

Notice that each item in this group of data (which may be a file of records or an array of some sort) contains as its second field a pointer to

Figure 15-5. Pointer Variable Is Stored at Memory Address that Contains a Pointer to Another Memory Address



the next item in alphabetical sequence. The data items themselves are in no particular order.

Sorting these items into alphabetical sequence becomes a simple matter of shuffling the link addresses. Figure 15-7 illustrates what we mean by "shuffling the link addresses". It may be a good idea to follow along in that figure as we discuss the process.

The first record, which contains the value Able, also includes a pointer to the next item in alphabetical order. This pointer is called a link. If we sorted the items on the list in alphabetical order, the next one after *Able* would be *Baker*, which is located at memory address 0005. For this reason we put the value 0005 in the link field of the first record. The next item in alphabetical order is *Charlie*, at address 0003, so Baker's record contains a link value of 0003. Follow the rest of the list through and you'll find that it's all in alphabetical order.

Logically enough, linked lists most often appear in Macintosh Pascal programs as records in a data file or data structure where manipulation of records and record sequences is a major task of the program. Each time we add a record to the file of data being manipulated, we update these links to keep the file in the sequence we want. The beauty of this process lies in the fact that we don't have to physically relocate the records in the computer or on the disk; all we have to do is reset the values of the pointers in the link fields.

End of List

Note that *Pascal*, the last item in alphabetical order on our sample linked list, has as its link address a special Macintosh Pascal constant called

Figure 15-6. Typical Linked List

Memory location	Contents of Records	
	Data item	Link value
0001	Able	0005
0002	Dog	0007
0003	Charlie	0002
0004	Pascal	nil
0005	Baker	0003
0006	Fellini	0004
0007	Ezra	0006

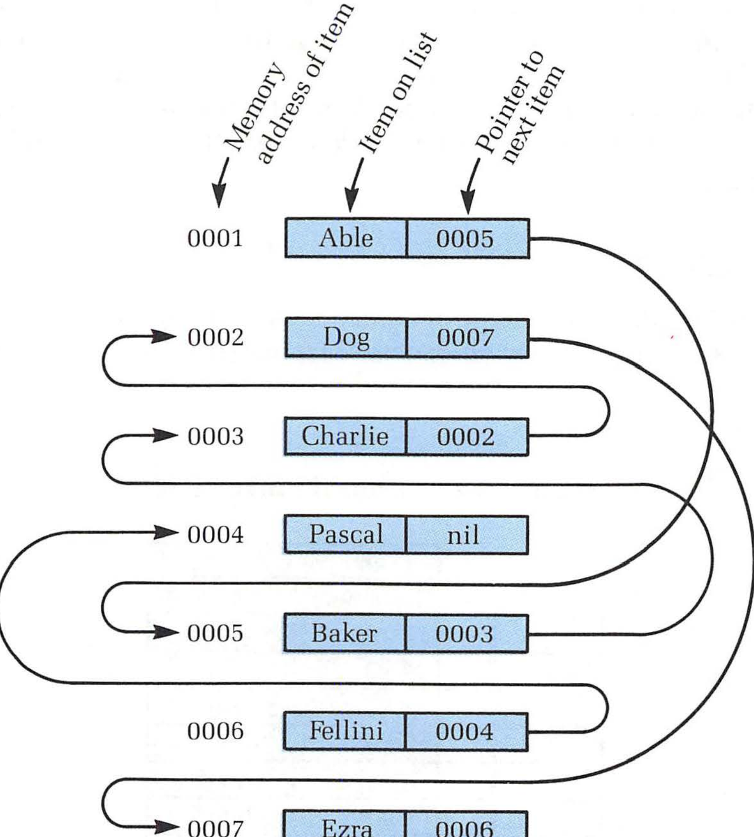
nil. This term literally means "nothing". Nil tells Mac Pascal that there is no link from this record to any other record in the file; in other words, the end of this particular sequence of links has been reached.

When dealing with dynamic data structures in a file, you should always write a final record containing the pointer Nil so that your program will know when it reaches the end of the file. That way, you don't have to know in advance how many records the program is going to have to read, display, or otherwise work with.

Pointers and Linked Lists

As we mentioned a few moments ago, the linking field in a linked list is a pointer; that is, it's a variable of type Pointer. When we define a record

Figure 15-7. Sorting a Typical Linked List



that is going to be used in a linked list, we define one field (usually, but not necessarily, the last field in the record) to be of type Pointer and then put data into it as appropriate to keep the list linked.

A Better Shopping List

To demonstrate more fully the use of pointers and dynamic data structures, let's go back to Chapter 11's shopping list program. For one thing, it is easier to see the advantages of such data structures in a program whose limitations you already know well than it might be in a new program. For another, this sample revision demonstrates an approach you might take in revising other programs in order to enhance them with this new feature of Pascal.

Type in the following program, Check, Save, and Run it.

```
program BetterShopper;
 type
 ItemField = string[24];
  CostField = real;
  ItemPointer = ^ ShoppingList;
  ShoppingList = record
    item:ItemField;
    cost:CostField;
    next:ItemPointer;
   end;
  var
   InitialPtr,CurrentPtr:ItemPointer;
   ThisItem:ItemField;
   ThisCost:CostField;
begin
 New(InitialPtr);
 CurrentPtr: = InitialPtr;
 Writeln('Enter items and prices.');
 Writeln('Type "stop" to quit.');
 Readln(ThisItem);
 Readln(ThisCost);
 while ThisItem<>'stop' do
  begin
   with CurrentPtr^ do
    begin {pointer update}
      New(next);
      CurrentPtr: = next;
    end; {pointer update}
```

```
    with CurrentPtr^ do
      begin {set up last record}
        item: = ThisItem;
        cost: = ThisCost;
        next: = nil;
      end; {set up last record}
    Readln(ThisItem);
    if ThisItem<>'stop' then
      Readln(ThisCost);
  end;
CurrentPtr: = InitialPtr^ .next;        ← "Next" avoids display problem
page;
while CurrentPtr<>nil do
  begin
      with CurrentPtr^ do
          begin {display list}
              Write (item);
              Writeln(', $',cost :1:2);
              Currentptr: = next;
          end;
      end; {display list}
end.
```

This program is similar to the shopping list program from Chapter 11, except that it doesn't limit us to three or any other arbitrary number of items. It keeps accepting entries until the user types "stop". (We've purposely omitted the totalling function and the review procedure here to keep the program short, but incorporating them would be very simple if you'd like the challenge.)

Use of Initial and Current Pointers

In the Var section of the program, we define two variables, InitialPtr and CurrentPtr to be of type ItemPointer. Previously, in the Type section, we described type ItemPointer to be a pointer to a shopping list record.

You'll recall from the earlier use of this program that we defined a record called ShoppingList, which contained an item and a cost. We've added a field called Next, which is our link address field, to the record format here.

Setting Up Initial Pointer Value

The first line in the main program loop says:

```
New(InitialPtr);
```

This statement sets up memory storage for the variable to be pointed to by InitialPtr. The next line assigns the value of the pointer variable Initialptr to the pointer variable CurrentPtr. Then the first item and cost are read into the program.

The next two groups of statements, the ones labeled "pointer update" and "set up last record", are the "brains" of the program. The first uses the statement

```
New(next);
```

to set up the memory location for the next pointer to be created by the next item added to the shopping list, then reassigns this pointer to CurrentPointer, which always points to the current item being accessed.

The next group of statements assigns the item and cost just entered to the Item and Cost fields in the record pointed to by CurrentPtr^ (remember, that's what the "with" statement does) and causes the Next or link field to contain the value Nil. This way, when the user ends the input by typing "stop", the last record is properly set up for the end-of-list condition we discussed earlier.

Displaying the Result Using Pointers

Finally, we use the CurrentPtr^ pointer variable to cycle through the list and display it on the Mac's screen.

Notice the statement that says:

```
while CurrentPtr<>nil do
```

which checks for the end-of-list condition. This statement points out that even though we've defined the field Next to be a pointer field, the value Nil will be understood by Pascal. Don't make the fairly common mistake of putting "nil" in quotation marks in the program, or you'll create some very interesting problems for yourself!

What Really Happens

Now that we've taken a close look at certain parts of the revised shopping list program, let's look at the program as a whole. We can describe what essentially happens as a list of processing steps that could also apply to any similar Macintosh Pascal program.

1. We set up an initial pointer (InitialPtr) with the New statement.
2. We assign the current pointer (CurrentPtr) to the initial pointer's value.
3. We update the current pointer each time through the entry process by using New(next) to create a new pointer for the next link and assigning its value to the pointer variable CurrentPtr.
4. We set up the latest record added to appear to be the last record in the list, just in case it is.
5. We go back and get another list entry from the user.
6. We've already set up the last item in the list, so when input is completed, the program simply stops.

Helpful Hint for Display Clean-Up

One other technique demonstrated in this program may prove useful to you in writing other programs involving dynamic data structures. Notice the line that says

```
CurrentPtr: = InitialPtr^ .next;
```

and has a comment about a display problem. If this line is omitted, and instead CurrentPointer is set to the value of InitialPointer, you will get a blank item field and a $0.00 cost field as the first line in your display. This is because the first record was skipped over and actually contains no data. Setting up the starting CurrentPointer value based on the value of the "next" record pointer in the InitialPointer value starts the display with the second record and solves this problem.

Summary

In this chapter, you've been introduced to the idea of dynamic data structures and have learned why and how they are used. You've dealt with the New procedure and with pointer variables, and you've learned to set up and manipulate linked lists of data.

The next chapter presents some intriguing aspects of Macintosh that you can now appreciate fully because you've been initiated into the wonders of the inner workings of the Mac.

16

Other Macintosh Goodies

When you finish this chapter, you'll know

- **How to set and use day, date, and time information**
- **How to use SysBeep in programs for simple sound**
- **How to create special sounds using Note**
- **How to make music**

This chapter will briefly examine two further interesting and useful aspects of Macintosh Pascal. You can then discover more of their many dimensions on your own. These two "goodies" are the built-in procedures and functions for setting and finding out the time and date from the Mac's built-in system clock; and ways of producing sound on the Mac.

Getting Time in Hand

As you know, the Macintosh comes equipped with a system clock. The battery-powered clock keeps the time and date current even when the Mac is turned off. Until now, this may not have seemed like an exceptionally useful feature. After all, most of us have other clocks, and very few of us (though there are exceptions) have trouble keeping track of what day it is. Of what value, then, is the fact that the Mac "knows" the day and time?

There will be many occasions when you'll want to record the date and time something happened in one of your programs. For example, you might want to know if the sales figures you're about to give your boss are the latest available. You might want to print the current date and time on reports you produce with the Mac so that you can refer later to this information to determine the currency of your data.

Getting Mac to Tell Us the Time of Day

Getting date and time information in Macintosh Pascal is extremely simple. The program below will retrieve the current day, date, and time from the system clock in the Mac and display it in a commonly acceptable format on the screen. Of course, once your program has learned the date and time, it can do other things with this information besides displaying it on the screen. Typically, the information would be stored in a data file, printed on the printer or saved on the disk.

Type this program into your program window, Check it, Save it, and Run it.

```
program DateTimer;
var
  DateInfo:DateTimeRec;
begin
GetTime(DateInfo);
Writeln(DateInfo.month:2,'/',DateInfo.day:2,'/',(DateInfo.year-1900):2);
Writeln(DateInfo.hour:2,'/',DateInfo.minute:2);
Write('It is ');
case DateInfo.DayOfWeek of
  1:
     Writeln('Sunday.');
  2:
     Writeln('Monday.');
  3:
     Writeln('Tuesday.');
  4:
     Writeln('Wednesday.');
  5:
     Writeln('Thursday.');
  6:
     Writeln('Friday.');
  7:
     Writeln('Saturday.');
  end;
end.
```

A quick look at this program will reveal how the Mac stores day, date, and time information in its internal memory. It uses a record of a predeclared type called DateTimeRec. There are six fields in this record: year, month, day, hour, minute, and a number between 1 and 7 representing the day of the week. The year is stored as a four-digit value (i.e., 1984, not 84). This is why we subtract 1900 from the year when we print the year in the usual mm/dd/yy format, where "mm" is the month, "dd" is the day, and "yy" is the last two digits of the year.

Obviously, we don't need to access all the fields of the DateTimeRec record. We can simply pick the portion we want. Thus, if the only thing of interest is the day of the week on which a program is being run, it's easy to simply check a variable called DateInfo.DayOfWeek (substituting, of course, the name of the record variable you've defined for your use of the word 'DateInfo').

Resetting Mac's Clock and Calendar

If for some reason we want to change date and time information stored in the Macintosh, we simply store new values in the variables that make up the fields of the DateTimeRec record. It is difficult to think of a reason for doing that (other than, perhaps, to play a prank on someone) but it's comforting to know that Mac gives us control over even the time of day — at least as far as the computer is concerned.

A word of caution: If you *do* change the Mac's date and time information for some reason, be sure to change it back when you're done. It's easy to reset the time from within the Apple accessories that come with the Mac, but you might forget to do so.

How Long Have I Been Working at This Lesson?

One more interesting thing the Mac can do in the realm of time is to tell us — with great precision — how much time has elapsed since we last turned the computer on or reset it. If you want to know how long you've been trying to solve that impossible problem your Computer Science instructor gave you this morning, you can divert yourself with a program that looks something like this:

```
program HowLong;
  var
    TotalTime:LongInt;
    hours,minutes,seconds:integer;
begin
    TotalTime: = TickCount;
  seconds: = trunc(TickCount/60);
  TotalTime: = seconds;
  seconds: = TotalTime mod 60;
  minutes: = trunc(TotalTime/60);
  TotalTime: = minutes;
  minutes: = TotalTime mod 60;
  hours: = trunc(TotalTime/60);
Writeln(hours:2,' hours,',minutes:2,' minutes,',seconds:2,' seconds.');
end.
```

The program doesn't require you to do anything. Just type it in, Check, Save, and Run it. The program will tell you precisely how long it has been since you turned on your Macintosh.

Checking the Ol' Ticker

Notice that the Var section of the program defines a variable called TotalTime to be of type LongInt. Later, we assign the value of a built-in variable called TickCount to it. TickCount contains the time, in sixtieths of a second, since the Mac was last turned on. Since it doesn't take very long for a ticker ticking at sixtieth-of-a-second intervals to get past the allowable upper limit for a variable of type Integer, we have to use a LongInt type variable to hold the value.

The rest of the program simply takes the time saved in TickCount and divides it successively by sixty to find the time in seconds, minutes, and hours. It then prints that information on the screen.

This ability of the Mac to keep track of elapsed time can be put to use in doing what are called "benchmark" programs. In such a program, the time it takes the computer to carry out a complex series of steps (or a great many simple steps) using a given programming language (for example) is recorded and used as a benchmark to measure other languages against. Obviously, all we'd have to do is to use TickCount, check it at the start and conclusion of the program, and perform a minor calculation on the difference between the two values to give us a time we could use to compare with other machines and Pascals to find out how fast Macintosh Pascal really is.

The Mac: A Sound Decision

Most people think of computers as providing only visual information to the user. However, many computers can also generate sound. Besides being a welcome change from visual information, sound has several advantages. For one thing, you don't need to be watching the screen to get the message. Sound can also reinforce visual images and lend excitement to a visual program.

Your Macintosh comes with a surprisingly sophisticated sound generation capability. In fact, as you probably know, it's possible to program the Mac to imitate the human voice! Explaining this particular feat is beyond the scope of this chapter, but we *are* going to introduce you to some of the more fundamental methods of sound generation.

Three basic ways of creating sound effects on the Macintosh are: manipulation of waveforms that contain a representation of data to be used by a program to produce sound; telling Mac to send a beep through its built-in speaker; and telling Mac to use its built-in speaker to sound a specific note at a specific volume for a specific time period. The first method is rather complex, so this section will look at methods two and three. First, though, let's talk briefly about some of the basics of sound on the Mac.

Three Factors in Sound

Any sound — whether created on a computer like the Mac or by the proverbial tree falling in the proverbial empty forest — has three characteristics: frequency (or pitch), volume, and duration. Other traits cause sounds to have different *qualities*, but we will confine our present discussion to these three important factors. We need to know these factors to describe any sound.

Volume and duration are easy to understand. You can tell how loud a sound is, at least relative to other sounds, simply by listening to it. You've undoubtedly experimented with the Mac control panel on the Apple pull-down menu and noted that you can control the general level of sound in the system over a wide range. Similarly, duration is relatively easy to judge. Although we may have difficulty quantifying the duration of a sound, we usually have no trouble deciding whether it is short, average in length, or long. Most people would call two billiard balls clicking together a short sound, for example, and a foghorn long.

But what about frequency? What is it, and how do we get some appreciation for it? Let's examine this important quality of sound.

Frequency and Sound Waves

All sound is made up of *waves*. Basically there are three different kinds: sine waves, square waves, and "freeform" or "noise" waves. The first two are regular waves, while the third is irregularly shaped. Regular waves produce sounds like music, pure tones, and even buzzing noises. Irregular waves are characteristic of human speech and noise (and isn't *that* an interesting coincidence?). Figure 16-1 depicts the three types of waves.

Regardless of the type of wave composing it, sound has a value called *wavelength*, which is the distance from the peak of one sound wave to the peak of the next. Wavelength is closely related to the *frequency* of a sound. Frequency is the number of waves or cycles produced by the sound per second. Another name for cycles per second is "Herz", abbreviated "Hz".

The frequency of a sound determines the sound's *pitch*. High C on a piano, for example, differs from middle C in pitch and frequency.

Beep-Beep!

With this information in mind, let's look at the two methods of sound production on the Mac that will concern us in this chapter. The first involves a procedure we've used several times in our Macintosh Pascal programs without discussing it. The procedure is called SysBeep, which, as you might guess, is short for "system beep".

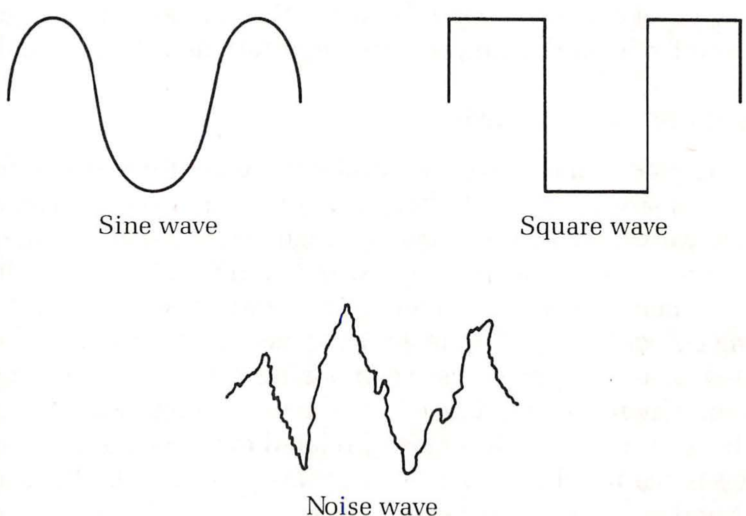
Whenever you turn on the Mac, make a mistake, or cause a Dialog box containing an important question to appear, you hear a SysBeep. If you want to produce a system beep, all you need to tell the procedure is the duration of the sound. The frequency is predetermined by Macintosh Pascal and is not subject to modification. The volume is determined by the setting on the Mac's control panel.

One line of Macintosh Pascal code like this will produce a system beep:

```
SysBeep(25);
```

A value of 25 for the duration of the beep produces a sound nearly like the one the Mac makes when we turn it on. Smaller values produce

Figure 16-1. The Three Types of Sound Waves



the shorter beeps we usually hear when we are running Mac Pascal and the Mac wants to attract our attention.

SysBeep and the Control Panel

You may have discovered that if the volume on the Mac's control panel is turned down to zero, the machine still makes a characteristic beeping sound when turned on but reacts differently to get our attention during a program run. If you've never done this before, give it a try now. Pull down the control panel on the Apple menu and move the slide on the sound indicator to the bottom position. If your eyes are quick, you'll notice something about the display as you release the mouse (we'll describe what you see in a moment). Now click the control panel away and type into the Instant window the following line:

```
SysBeep(50);
```

Click the Do It button in the Instant window and keep your eyes on the Menu bar. There! Did you see it? No? Try it again. Got it? Good.

If we call the SysBeep procedure while the Mac's volume is turned off, the Mac says to itself, "Whoops! If I try to get my user's attention with a beep now, it won't work. I'd better try something else." So it flashes the Menu bar once for each SysBeep call. Pretty clever, this Macintosh!

Be sure to reset the volume on the control panel before we go on. (We like ours set at 3, but tastes vary.) If you don't do this, the rest of this chapter could seem a bit mysterious!

When to Use SysBeep

You've probably guessed by now that the most frequent use of the SysBeep procedure in Macintosh Pascal programs is to get the attention of the user. Sometimes we use it to signify the end of a procedure, particularly a long one where the user might want to leave the computer and do something else until the program is finished.

But what if we want to do something more interesting with sound? What if, for example, we want to play a little song as an alarm clock or as a reward for a young user who just got fifty math problems right? That's where the next topic comes in.

Noteworthy Notes

To produce pure, more or less musical tones, Macintosh Pascal provides a procedure called Note. When called, this procedure produces a single square-wave tone. To see how this sounds, pull down the Instant window, clear it if necessary, and type in the following lines:

```
note(264,128,75);
note(528,128,75);
```

Click on the Do It button in the Instant window and listen to the result. The Mac should play two consecutive "C" notes. The first is middle C on a piano and the second is the C one octave above that. (If pianos and octaves don't mean anything to you, don't worry about it; you need not be a musician to use this procedure.)

Now, let's take the Note procedure apart and find out what really happens when it is called.

What's in the Note?

Notice that the Note procedure requires three parameters. These are, in order, frequency, amplitude (or volume for us laypeople), and duration. The first parameter, frequency, can take any value from 12 to 783,360. It therefore sometimes isn't an integer variable in Mac Pascal; instead, it may be of a type we've seen before called LongInt (for "long integer").

The second value in the Note procedure is the volume setting. This can be any value from 0 (which produces no sound, or what is called in music a "rest") to 255. We can't say exactly how loud a sound a value of (for example) 100 will produce, because the volume is, as we have seen, tied into the volume setting on the control panel. If the control panel volume is set at 3, we'll get a different volume of sound from a note with a volume parameter of 100 than we will if we have the control panel volume set at 6. We can say, however, that if we use a value of 100 for the volume parameter, we will get a sound half as loud as if we had used 200 and twice as loud as if we had used 50.

The final value given to the Note procedure is the duration value. In case you're interested (or want to impress someone), we can tell you that duration is measured in ticks (remember our friend TickCount?). Any value from 0 to 255 can be placed in the duration parameter.

A Note-able Experiment

Let's look at a program that will let us play around with the Note procedure's parameters. Type it in; it isn't necessary to save it, although you might want to do so because it can help you search systematically for the "right" sound to use in a particular situation.

```
program NoteTester;
var
  freq,vol,dur:integer;
begin
  repeat
    Write ('What frequency? ');
    Readln(freq);
    Write('What volume setting? ');
    Readln(vol);
    Write('How long a note? ');
    Readln(dur);
    note(freq,vol,dur);
  until freq = 0;
end.
```

The program asks for a frequency, a volume, and a duration for a note and produces the note described. It then asks the user for another note description and continues the process until a zero is typed in response to the question about frequency.

If you're like us, you'll quickly find comfortable volume and duration settings and then settle down to play with the frequency values. You'll probably find that any frequency value below about 150 sounds more like a buzz than a musical tone. Frequency values above about 4000 begin to get uncomfortably high-pitched. Anything between those two values, though, produces something more or less resembling musical notes.

Music, MacMaestro, Please!

Now let's put all this information about the Note procedure to some practical use by getting the Mac to play the old '20's tune, "Happy Days Are Here Again" — in celebration of itself, perhaps. The following program will do that. It's somewhat long, because each note's pitch and duration has to be entered into an array, but patience will be rewarded with an understanding of the way music is set up on the Mac. Type in the program, Check it, and Save it. Then Run it while you sit back and enjoy the symphony!

```
program HappyDays;
 const
  qn = 15; {duration value for a quarter note}
 var
  count:integer;
  tune:array[1..31,1..2] of integer;
 procedure SetTune;
 begin
  tune[1,1]: = 264;
  tune[1,2]: = qn;
  tune[2,1]: = 330;
  tune[2,2]: = qn;
  tune[3,1]: = 330;
  tune[3,2]: = qn*2;
  tune[4,1]: = 396;
  tune[4,2]: = qn*3;
  tune[5,1]: = 396;
  tune[5,2]: = qn;
  tune[6,1]: = 528;
  tune[6,2]: = qn;
  tune[7,1]: = 528;
  tune[7,2]: = qn*2;
  tune[8,1]: = 660;
  tune[8,2]: = qn*3;
  tune[9,1]: = 660;
  tune[9,2]: = qn;
  tune[10,1]: = 528;
  tune[10,2]: = qn;
  tune[11,1]: = 528;
  tune[11,2]: = qn*2;
  tune[12,1]: = 396;
  tune[12,2]: = qn*3;
  tune[13,1]: = 396;
  tune[13,2]: = qn;
  tune[14,1]: = 330;
  tune[14,2]: = qn;
  tune[15,1]: = 330;
  tune[15,2]: = qn*2;
  tune[16,1]: = 660;
  tune[16,2]: = qn*2;
  tune[17,1]: = 634;
  tune[17,2]: = qn;
  tune[18,1]: = 594;
  tune[18,2]: = qn;
  tune[19,1]: = 495;
  tune[19,2]: = qn;
  tune[20,1]: = 495;
  tune[20,2]: = qn*2;
  tune[21,1]: = 396;
```

```
    tune[21,2]:=qn*3;
    tune[22,1]:=396;
    tune[22,2]:=qn;
    tune[23,1]:=352;
    tune[23,2]:=qn;
    tune[24,1]:=352;
    tune[24,2]:=qn*2;
    tune[25,1]:=594;
    tune[25,2]:=qn*2;
    tune[26,1]:=660;
    tune[26,2]:=qn;
    tune[27,1]:=528;
    tune[27,2]:=qn*2;
    tune[28,1]:=528;
    tune[28,2]:=qn*2;
    tune[29,1]:=528;
    tune[29,2]:=qn*2;
    tune[30,1]:=528;
    tune[30,2]:=qn*2;
    tune[31,1]:=528;
    tune[31,2]:=qn*4;
  end; {SetTune}
begin
  SetTune;
  for count:=1 to 31 do
    if not button then
      note(tune[count,1],100,tune[count,2]);
end.
```

Phew! That's more typing than you've had to do for any other program in this book (except, perhaps, some of the Mac-r-cises). Be sure to proofread carefully the lines in the procedure SetTune, which is the heart of the program.

Setting Up Music on the Mac

You can use this program as a model for setting up almost any music on the Macintosh Pascal. First, get a piece of sheet music and translate the notes into letter notes and durations on a piece of paper. You can use abbreviations like "q" for quarter note, "e" for eighth note, "dh" for dotted half note, and so on — or whatever other abbreviations are comfortable. The first few notes in "Happy Days Are Here Again" look like Table 16-1 when we do that.

The next step is to translate the note values (C, B, G, E-flat, and so on) into frequency values for the Note procedure. We've helped with this by designing Table 16-2 which contains frequency equivalents for all the notes on the musical scale from one octave below middle C to two octaves above middle C. (Some notes are off just a tiny bit, though you probably

Table 16-1. Opening Notes of Song for Mac Music

Note	*Duration*
C	Q
E	Q
E	H
G	DH
G	Q
C↑	Q
C↑	H

(↑ = octave up)

Table 16-2. Mac Frequency Equivalents to Musical Notes

NOTE	*OCTAVE*			
	One below Middle C	*Middle C*	*One above Middle C*	*Two above Middle C*
C	132	264	528	1056
D♭	141	282	563	1126
D	149	297	594	1188
E♭	158	317	634	1267
E	165	330	660	1320
F	176	352	704	1408
F♯	188	376	751	1502
G	198	396	792	1584
A♭	211	422	845	1690
A	220	440	880	1760
B♭	231	462	924	1848
B	247	495	990	1980

won't even be able to detect it. That's because the precise values of the notes require the use of fractions that are not easily available in the Note procedure.)

Next, count the number of notes in the song you're playing. In the case of "Happy Days Are Here Again", we needed thirty-one notes to play the chorus. Create a two-dimensional array which is that many values by two (in this case 31 x 2) and define it to be an array of type Integer. (Yes, we remember that we said frequency can be a "long integer". Since for musical purposes no frequency value exceeds the range of Pascal integer values, however, we can get away with calling frequency an Integer variable here.

Now define each note's duration relative to a quarter note. Count a half note as twice the length of a quarter note, a dotted half note as three times a quarter note, and so on.

After all that preparation is done, you're ready for the exciting job of typing all the note frequency and duration values into your array. Things should be straightforward from there.

One more thing: We like to set up a constant in the program that puts the value of the quarter note into numeric terms. That way, if we want to vary the tempo of the song a little, we need to adjust only one number instead of all 31 (or 50 or 200).

That's all there is to it! With that information and some knowledge of how to read sheet music, you can translate any song into Mac tones.

Summary

In this chapter, we've looked at two different ways to produce sound on the Mac — the SysBeep and Note procedures. You've also learned a little about how the Mac keeps time and how to access time, day, and date information and use it in your programs.

Mac-r-cises

There are no formal Mac-r-cises for this chapter. However, you might consider writing a program that will let you *type in* frequency and duration values for any number of notes up to, say, 100; then play the resulting tune, and allow you to edit or adjust specific notes. Perhaps you can even have it print out the note values on the printer.

17

Three Complete Fun and Useful Programs

This chapter pulls together all you've learned about Macintosh Pascal by providing you with three complete programs to type into your Mac. We hope you will find them useful and entertaining additions to your Mac software library. In addition, they can afford an excellent learning opportunity.

Learning from Others' Programs

In the early days of computing, programs were shared via informal networks all over the country. There was little or no commercial market for such software, so pride of authorship was the only reward a programmer could expect to receive. Many people learned to program not from textbooks and instructional manuals (which were at that time largely nonexistent in any case) but by taking apart someone else's program, analyzing what the program did and how it did it, and modifying and adapting those techniques to their own programming needs.

As software sales mushroomed in the early 1980s, most commercial software began to be issued only in protected form — a form that computers could read but users could not. While this may have led to better protection for the programmers' products and therefore to better programs (who wants to spend several hundred hours doing a program just for people to give away?), it has also resulted in fewer learning opportunities. This chapter represents a small attempt at remedying that situation for readers of this book.

For a more extensive collection of programs, you can also refer to *Games and Utilities for the Macintosh* by Dan Shafer (New York: Plume/Waite, New American Library, 1985).

The Three Programs

The three programs furnished in this chapter are GridMaker, DumpFile, and ChuckALuck. We present them in order of increasing length and complexity.

GridMaker is a highly useful tool for programming graphics. If you've worked with some of the Quick-Draw commands in Chapter 6 and experimented with manipulating the Drawing window, you've probably encountered an interesting problem: Coming up with the proper positioning of shapes and lines to produce a particular picture in the Drawing window is often a hit-and-miss proposition. Our program produces a grid inside the Drawing window with numbered position lines, which you can then print out by using the Mac's [⌘] 4 key combination. With that in hand, you can sketch the drawing you want and know with great accuracy which coordinates are going to be needed to produce the desired graphic result.

Our DumpFile program will be quite valuable to you as you attempt to learn more about the way the Mac's disk files are organized and what's stored in them. It will display any text file — including a Macintosh Pascal program, which is a text file — either on the screen or through your printer, in both hexadecimal and human-readable characters. The latter are called "ASCII" (pronounced "ass-key") characters and are those you are used to seeing on the screen of your Mac. This program also demonstrates the use of disk files, which were introduced in Chapter 13.

ChuckALuck is a simulated dice game. It gives you a chance to see how graphics can be put to work in a program that is a lot of fun to play.

We will provide a brief explanation of each program and then reproduce its listing. We will not attempt to explain each procedure and statement in depth, however. Not only would that take many more pages than we have available — it would also defeat the main purposes of this chapter, which are to help you learn by analyzing programs and to give you a feeling for the overall organization of a Macintosh Pascal program.

GridMaker

This program uses only one built-in procedure that we have not introduced in this book. That is the procedure GlobalToLocal, which is called twice in the procedure Resize Drawing. The GlobalToLocal procedure enables us to work with the Drawing window in the context of the larger world that exists in the Mac's memory but can't be shown at one time on the display. It ensures that the grid produced in the Drawing window will be correct in its coordinates no matter where on the desktop the Drawing window may be located.

Here's the listing of the program GridMaker.

```
program GridMaker;
{ Calls the procedure DrawGrid to draw a grid on the Drawing Window. }
{ Program by Chuck Blanchard }
 var
  DrawingRect : Rect;
 procedure ResizeDrawing;
  { Resize the drawing window to the size given }
 begin
  SetRect(DrawingRect, 1, 39, 510, 340);
  SetDrawingRect(DrawingRect);
{ Now convert screen coordinates to window coordinates. }
  GlobaltoLocal(DrawingRect.TopLeft);
  GlobaltoLocal(DrawingRect.BotRight);
 end;
 procedure DrawGrid;
  const
   inc = 20;
  var
   i : integer;
 begin
  TextSize(9);
  i := inc;
  repeat
   MoveTo(0, i);
   Line(512, 0);
   MoveTo(1, i);
   DrawString(StringOf(i : 1));
   MoveTo(i, 0);
   Line(0, 350);
   MoveTo(i + 1, inc div 2);
   DrawString(StringOf(i : 1));
   i := i + inc;
  until i > 512;
 end;
begin
 HideAll;
 ResizeDrawing;
 ShowDrawing;
 DrawGrid;
end.
```

Type in and Save the program in the usual way. To Run it, simply Open it and choose the Go option from the Run menu. In a few moments, a grid like the one shown in Figure 17-1 will appear on your screen. Hold down the ⌘ and 4 keys simultaneously, and your printer will generate a "hard copy" of this handy-dandy, programmer's tool.

As you can see from Figure 17-1, the upper left corner of the window has the coordinates (1,39), while the lower right is located at (510,340). If you use the Size box in the lower right corner of the window to shrink the window, the grid stays visible. If, however, you shrink the window and then expand it again, the grid is not redrawn.

To make GridMaker's grid really useful, we recommend that you make a printout of the pattern, put it on a clipboard, and cover it with acetate. Now you can use a grease pencil or some other erasable marker to create shapes. The grid will tell you what values to give your program's QuickDraw command parameters so that you'll get the drawing you want.

DumpFile

This program will permit you to examine the contents of any *text* file on any Macintosh diskette (unless it's somehow protected against such examination). When it is run, it will display in the Mac's Drawing window the hexadecimal and ASCII values of each byte on the diskette. As you go on in programming, you may find this tool useful for examining files created by other people's programs or for making sure your own programs are creating the files they're supposed to.

Figure 17-1. Result of Running GridMaker

What Values of Whose "Bytes"?

If the concepts of hexadecimal notation and bytes are totally unfamiliar to you, it might be a good idea to read an introductory book on computers and how they work for more information. If the ideas are familiar but you can't quite remember what they mean, the next few paragraphs should refresh your memory.

Spelling Out "Hex"

Hexadecimal notation is a way of representing values in the number base of 16 instead of our traditional base of 10. To the ten digits (0-9) of the decimal number system we use every day, hexadecimal notation adds six new ones (using, arbitrarily, the letters A-F), thus giving us enough digits to represent numbers from 0-15 in each position of a value.

In decimal notation, the number 123 is interpreted as meaning 3 times 1 (which is 10 raised to the "zero power") plus two times 10 (which is 10 raised to the first power) plus 1 times 100 (which is 10 raised to the second power, or squared). Broken down in this way, the number 123 would look like this:

N = 123 (decimal)

$N = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$

$N = 1 \times 100 + 2 \times 10 + 3 \times 1$

N = 123 (decimal)

On the other hand, the number 123 in *hex* (that's the programmer's shorthand for hexadecimal notation), stands for a *decimal* value of 291! The same principles of raising each position to a power apply, but the *base* is now 16, so the breakdown looks like this:

N = 123 (hexadecimal)

$N = 1 \times 16^2 + 2 \times 16^1 + 3 \times 16^0$

$N = 1 \times 256 + 2 \times 16 + 3 \times 1$

N = 291 (decimal)

When you run the DumpFile program, the left side of your screen or printed text will show the hex values for all the characters in the file you're examining. Hex is much closer than decimal to the computer's numerical "native language", so hex values can be useful as you go on to more advanced programming and deepen your understanding of the Mac.

ASCII = Human Readability

ASCII is an acronym for American Standard Code for Information Interchange. In this widely used code, alphabetic, numeric, and certain other characters are given particular numerical values. For example, the decimal number 66 equals a capital *B* in ASCII, while a lowercase *b* is decimal 98. These numerical values can also, of course, be represented in hexadecimal.

Table 17-1 shows the hexadecimal equivalents for the most commonly used ASCII codes.

When DumpFile is run, the right side of the screen or printout displays the ASCII characters represented by the hex values on the left side, as the sample display in Figure 17-2 shows. You can see that these are (for

Table 17-1. ASCII Codes Expressed in Hexadecimal

Hex	*ASCII*	*Hex*	*ASCII*	*Hex*	*ASCII*	*Hex*	*ASCII*	*Hex*	*ASCII*	*Hex*	*ASCII*
20	Space	30	0	40	@	50	P	60	‘	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	:	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	}	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	rubout

the most part) readable characters; you can recognize them and understand what they are. Where a character is stored on the disk in such a way that there is no ASCII equivalent, the program prints a period in that position.

Running DumpFile

Type in the program from the listing and Save it on your disk. After you have Opened DumpFile in the usual way and selected the Go option from the Run menu, you will be shown the same sort of file selection Dialog box that you've seen on the regular Pascal File menu. Open the file you want to examine by using a double-click on the file name or a click on the file name followed by a click on the Open button. (If the file you want to see isn't shown in the directory, it may be because it isn't a type of file that can be dumped with this program. But you can still attempt to do so by using the Cancel button and then supplying the file's name when asked for it.)

After a file is selected, the program asks if you want it to "Send output to printer?" If you type a "Y" here, the program will dump the file being

Figure 17-2. Display Produced by DumpFile

```
 File  Edit  Search  Run  Windows
                         Drawing
0BF0: 74 72 69 6E 67 28 27 20 20 20 27 29 3B 0D 20 20 tring('   ');.
0C00: 66 6F 72 20 69 20 3A 3D 20 30 20 74 6F 20 28 66 for i := 0 to (f
0C10: 69 6C 65 61 64 64 72 20 2D 20 31 29 20 6D 6F 64 ileaddr - 1) mod
0C20: 20 6E 75 6D 61 63 72 6F 73 73 20 64 6F 0D 20 20  numacross do.
0C30: 20 77 72 69 74 65 61 73 63 69 69 28 61 73 63 69  writeascii(asci
0C40: 69 5B 69 5D 29 3B 0D 20 65 6E 64 3B 20 20 7B 20 i[i]);. end;  {
0C50: 44 75 6D 70 46 69 6C 65 20 7D 0D 0D 20 70 72 6F DumpFile }.. pro
0C60: 63 65 64 75 72 65 20 52 65 73 69 7A 65 44 72 61 cedure ResizeDra
0C70: 77 69 6E 67 3B 0D 20 20 7B 20 52 65 73 69 7A 65 wing;.  { Resize
0C80: 20 74 68 65 20 64 72 61 77 69 6E 67 20 77 69 6E  the drawing win
0C90: 64 6F 77 20 74 6F 20 74 68 65 20 76 61 6C 75 65 dow to the value
0CA0: 73 20 67 69 76 65 6E 20 61 73 20 6C 69 74 65 72 s given as liter
0CB0: 61 6C 73 2E 20 7D 0D 20 62 65 67 69 6E 0D 20 20 als. }. begin.
0CC0: 53 65 74 52 65 63 74 28 44 72 61 77 69 6E 67 52 SetRect(DrawingR
0CD0: 65 63 74 2C 20 31 2C 20 33 39 2C 20 35 31 30 2C ect, 1, 39, 510,
```

examined to the printer for later review; otherwise, it will display the file contents on the Mac's screen. While the file is being displayed on the screen or printed, you can pause the output by clicking the mouse button anywhere in the Drawing window and holding it down. Releasing the button will make the display resume.

Here's the listing of the DumpFile program.

```
program DumpFile;
{ Dump a file to the screen in hexadecimal and ASCII format. }
{ Program by Chuck Blanchard }
 uses
  QuickDraw1, QuickDraw2;
 const
  Monaco = 4;
  PenLeft = 0;
  PenTop = 0;
 type
  bytetype = 0..255;
 var
  FileName : string;
  ch : char;
  LookFile : file of char;
  printer : Text;
  DrawingRect : Rect;
  LineHeight : integer;
  FInfo : FontInfo;
  print : boolean;
 procedure WriteString (str : string);
{Write the given string to the screen and to the printer if it is on.}
 begin
  DrawString(str);
  if print then
   Write(printer, str);
 end;   { WriteString }
 procedure CRLinefeed;
  var
   pen : Point;
   tempRgnH : RgnHandle;
 begin
  GetPen(pen);
  if pen.v + LineHeight > DrawingRect.bottom - 32 then
   begin   { must scroll }
    tempRgnH := NewRgn;
    ScrollRect(DrawingRect, 0, -LineHeight, tempRgnH);
    MoveTo(penLeft, pen.v);
    DisposeRgn(tempRgnH);
   end
```

```
  else
    MoveTo(penLeft, pen.v + LineHeight);
  if print then
    Writeln(printer);
 end;

 procedure WriteByte (byte : bytetype);
{ Given a byte (range 0 to 255), write it to the screen in }
{ hexadecimal notation.   Pause if the mouse is down. }
  const
    hexs = '0123456789ABCDEF';
 begin
  WriteString(copy(hexs, (byte div 16) + 1, 1));
  WriteString(copy(hexs, (byte mod 16) + 1, 1));
  while button do
    ;
 end;

 procedure WriteASCII (ch : char);
 { Given a character, write it to the screen if it is a printable }
 { character, otherwise write a period. }
 begin
  if not ((ord(ch) >= 32) and (ord(ch) <= $d8)) then
    ch := '.';
  WriteString(ch);
 end;

 procedure DumpFile;
  { The file LookFile has been opened.   Dump the contents to the }
  { screen as 8 bit hex bytes, followed by the ASCII representation. }
  const
    numacross = 16;
    numacrossminus1 = 15;
  var
    fileaddr, fileword : longint;
    ascii : packed array[0..numacrossminus1] of char;
    i : integer;
    bytevalue : bytetype;

  function bytenum (value : longint;
              count : integer) : integer;
{ Return the count'th byte in value.   Count is in range 0 to 3. }
  begin
    if count > 1 then
      value := BitShift(value, -16);
    if count mod 2 = 1 then
      value := BitShift(value, -8);
    bytenum := BitAnd(value, $ff);
  end;
```

```
  begin { DumpFile }
    fileaddr := 0;
    for i := 0 to numacrossminus1 do
      ascii[i] := ' ';
    while not Eof(LookFile) do
      begin
        if (fileaddr mod numacross) = 0 then
          begin
{ We have printed numacross digits.   Show the ASCII }
{ representation and start a new line with the file addr. }
            for i := 0 to numacrossminus1 do
              Writeascii(ascii[i]);
            CRlinefeed;
{ write the file address. }
            for i := 1 downto 0 do
              writebyte(bytenum(fileaddr, i));
            WriteString(': ');
          end;
        bytevalue := ord(LookFile^ );
        Writebyte(bytevalue);
        WriteString(' ');
        ascii[fileaddr mod numacross] := chr(bytevalue);
        fileaddr := fileaddr + 1;
        get(LookFile);
      end;   { while not End of File }
    for i := (fileaddr - 1) mod numacross to numacross - 2 do
      WriteString('    ');
    for i := 0 to (fileaddr - 1) mod numacross do
      Writeascii(ascii[i]);
  end;

  procedure ResizeDrawing;
    { Resize the Drawing window to the values given as literals. }
  begin
    SetRect(DrawingRect, 1, 39, 510, 340);
    SetDrawingRect(DrawingRect);
{ Now convert screen coordinates to window coordinates. }
    GlobaltoLocal(DrawingRect.TopLeft);
    GlobaltoLocal(DrawingRect.BotRight);
  end;

begin
  HideAll;
  ResizeDrawing;
  ShowDrawing;
  TextFont(Monaco);
  TextSize(12);
  GetFontInfo(FInfo);
    { Set the initial location of the graphics pen to the top of the window. }
```

```
MoveTo(PenLeft, PenTop);
  { Calculate the height of a line. }
with FInfo do
  LineHeight := Ascent + Descent + Leading;
  { Get the filename from the user. }
FileName := OldFileName('File you wish to examine?');
if FileName = '' then
  begin
    ShowText;
    Writeln('Do you wish to see a file that wasn''t');
    Writeln('in the directory? ');
    Readln(FileName);
    HideAll;
    ShowDrawing;
  end;
if FileName <> '' then
  begin { We have a filename, so try to open it. }
    reset(LookFile, FileName);
    if Eof(LookFile) then
      begin { There is nothing in this file. }
        DrawString(FileName);
        DrawString(' is empty');
      end
    else { Something in this file, dump it. }
      begin
        ShowText;
        Write('Send output to the printer? ');
        read(ch);
        print := ch in ['y', 'Y'];
        if print then
          ReWrite(printer, 'Printer:');
        ShowDrawing;
        DumpFile;
      end;
  end; { if the user specified a  filename }
end.
```

There are only a couple of really new ideas in this program. One is the concept of "regions" (Rgn) in the procedure CRLinefeed, and the other is the variable type *Packed* array. Neither of these needs to concern us here. Just type the instructions in as they are presented, and the program should run fine. If you are burning with curiosity about those two constructs, check your *Macintosh Pascal Reference Manual*.

ChuckALuck

This is a graphic computer version of an old dice game. Basically, the computer will roll three dice. You bet that a particular number will come up. If it comes up on one die, you win your bet back (that is, break even). If it appears on two dice, you double your money. If it's on all three dice, you receive three times your bet.

Figure 17-3 shows what the screen looks like as you play Chuck-ALuck. The nearest faces of the three dice show the numbers that count. At each turn, you'll be told how much money you have left (you'll also be told when you're broke!) and asked for a bet and a number. Your bet can be anything from 1 to the number of dollars left, and the number can, of course, be any value from 1 to 6. The dice are rolled, and you are then told how much you won or lost and asked if you want to try again.

All the constructs in this program have been introduced in our book, so it should be an easy program to follow as well as a good learning experience. Oh, of course, it's also a good game that will keep anyone with gambler's blood busy for hours — but that's secondary, right?

Figure 17-3. Sample Run of ChuckALuck

Following is the listing of the ChuckALuck program.

```
program ChuckALuck;
{ Program by Chuck Blanchard }
 const
  DieOffset = 150;
  DieZeroStart = 75;
  DotSize = 6;
  StartMoney = 1000; { Dollars to start with }
 type
  FaceType = (front, top, left);
  NumType = (mid, half, offset, slant);
 var
  HorzInfo, VertInfo : array[FaceType, NumType] of integer;
  OtherFace : array[1..6, FaceType] of integer;
  DrawingRect, TextRect : Rect;
  FrontRect, FrontFaceRect : array[0..2] of Rect;

 procedure DrawCube (dienum : integer);
 begin { Given which die to draw, draw the lines of the cube. }
  PenSize(2, 2);
  FrameRect(FrontRect[dienum]);
  MoveTo(FrontRect[dienum].right - 2, FrontRect[dienum].top);
  Line(-25, -25);
  Line(-48, 0);
  Line(0, 48);
  Line(25, 25);
  MoveTo(FrontRect[dienum].left, FrontRect[dienum].top);
  Line(-25, -25);
 end;

 procedure DrawFace (num, dienum : integer;
            face : FaceType);
{ Given the die number and which face, put dots on the face. }
  var
   i, halfhorz, halfvert, midhorz, midvert, horzoffset, vertoffset : integer;
   hslant, vslant : integer;
   r : Rect;
  procedure DrawDot (h, v : integer);
  begin { Draw a dot at the indicated location. }
   SetRect(r, h - halfhorz, v - halfvert, h + halfhorz, v + halfvert);
   FillOval(r, black);
  end; { DrawDot }
  procedure DrawPair (firstvert, secondvert : integer);
  begin { Draw a pair of dots, each in vertical pos -1 to 1. }
   DrawDot(midhorz - horzoffset + hslant * firstvert, midvert + firstvert * vertoffset - vslant);
   DrawDot(midhorz + horzoffset + hslant * secondvert, midvert + secondvert * vertoffset + vslant);
  end;
```

```
begin { DrawFace }
 halfhorz := HorzInfo[face, half];
 halfvert := VertInfo[face, half];
 midhorz := HorzInfo[face, mid] + dienum * DieOffset + DieZeroStart;
 midvert := VertInfo[face, mid];
 horzOffset := HorzInfo[face, offset];
 vertOffset := VertInfo[face, offset];
 hslant := HorzInfo[face, slant];
 vslant := VertInfo[face, slant];
 if odd(num) then { There is a dot in the middle. }
  DrawDot(midhorz, midvert);
 for i := 1 to num div 2 do
  case i of
   1:        { Draw topleft, lowerright dots for die 2 or 3. }
    DrawPair(1, -1);
   2:            { Draw lowerleft, upperright dots for die 4 or 5. }
    DrawPair(-1, 1);
   3:        { Draw middle pair of dots for die 6. }
    DrawPair(0, 0);
  end; { for i case i }
end;

function RandomRange (min, max : integer) : integer;
begin { Return a random integer between min and max inclusive. }
 RandomRange := abs(random mod (max - min + 1)) + min;
end;
procedure Initialize;
 var { Initialize info to draw all three dice. }
  faceindex : FaceType;
  numindex : NumType;
  i, j : integer;
begin
 for j := 0 to 2 do
  begin { Set up rectangle to frame and erase the front face. }
   i := DieZeroStart + j * DieOffset;
   SetRect(FrontRect[j], i, 75, i + 50, 125);
   FrontFaceRect[j] := FrontRect[j];
   InsetRect(FrontFaceRect[j], 2, 2);
  end;
 HorzInfo[front, half] := Dotsize div 2;
 HorzInfo[front, mid] := 25;
 HorzInfo[front, offset] := 15;
 HorzInfo[front, slant] := 0;
 VertInfo[front, half] := Dotsize div 2;
 VertInfo[front, mid] := 100;
 VertInfo[front, offset] := 15;
 VertInfo[front, slant] := 0;
 for faceIndex := top to left do
  for numindex := half to slant do
```

```
      begin
        HorzInfo[faceindex, numindex] := HorzInfo[front, numindex];
        VertInfo[faceindex, numindex] := VertInfo[front, numindex];
      end;
    HorzInfo[top, mid] := 12;
    HorzInfo[top, slant] := 7;
    HorzInfo[top, offset] := 13;
    VertInfo[top, half] := round((DotSize div 2) * 0.5);
    VertInfo[top, mid] := 62;
    VertInfo[top, offset] := 6;
    HorzInfo[left, mid] := -12;
    VertInfo[left, slant] := 7;
    HorzInfo[left, half] := round((DotSize div 2) * 0.5);
    VertInfo[left, mid] := 87;
    HorzInfo[left, offset] := 6;
    VertInfo[left, offset] := 13;
{ For numbers 1-6, describe which numbers are shown on the other faces. }
    OtherFace[1, top] := 3;
    OtherFace[2, top] := 4;
    OtherFace[3, top] := 6;
    OtherFace[4, top] := 1;
    OtherFace[5, top] := 3;
    OtherFace[6, top] := 4;
    for i := 1 to 6 do
      OtherFace[i, left] := 5;
    OtherFace[2, left] := 6;
    OtherFace[5, left] := 6;
  end;

  procedure ResizeDrawing;
    { Resize the Drawing window to the values given as literals. }
  begin
    SetRect(DrawingRect, 1, 39, 510, 340);
    SetDrawingRect(DrawingRect);
{ Now convert screen coordinates to window coordinates. }
    GlobaltoLocal(DrawingRect.TopLeft);
    GlobaltoLocal(DrawingRect.BotRight);
  end;

  function RollDie (dienum : integer) : integer;
    var { Roll the given die and return the number rolled. }
      j, k : integer;
  begin
    DrawCube(dienum);
    for j := 0 to RandomRange(2, 5) do
```

```
    begin
      EraseRect(FrontFaceRect[dienum]);
      k := RandomRange(1, 6);
      DrawFace(k, dienum, front);
      Note(10000, 20, 1);
    end;
  DrawFace(OtherFace[k, top], dienum, top);
  DrawFace(OtherFace[k, left], dienum, left);
  RollDie := k;
end;

procedure TryYourLuck;
  var { Play the game until the user wants to quit or runs out of money. }
    Money, WonLost, bet, guess, matched, i : longint;
    ch : char;
  function GetNum (prompt : string;
               min, max : integer) : integer;
    var
      i : integer;
  begin
    repeat
      Write(prompt);
      Readln(i);
        if (i < min) or (i > max) then
        Writeln('The number must be between ', min : 1, ' and ', max : 1, '.');
    until (i >= min) and (i <= max);
    GetNum := i;
  end; { GetNum }
begin
  money := StartMoney;
  Writeln('You have $', Money : 1, '.');
  repeat
    bet := GetNum('How Large a Bet? ', 1, money);
    guess := GetNum('Number to match? ', 1, 6);
    EraseRect(DrawingRect);
    matched := 0;
    for i := 0 to 2 do
      if RollDie(i) = guess then
        matched := matched + 1;
    if matched = 0 then
      matched := -1;
    WonLost := bet * matched;
    if WonLost < 0 then
      Writeln('You lost $', -WonLost : 1, '.')
    else
      Writeln('You won $', WonLost : 1, '.');
    Money := Money + WonLost;
    if Money <> 0 then
```

```
      begin
        Writeln('You have $', Money : 1, '.');
        Write('Try again? ');
        Read(ch);
        Writeln;
      end
    else
      Writeln('Sorry, you''re broke.')
  until not (ch in ['y', 'Y']) or (Money = 0);
 end;  { TryYourLuck }
begin { Main }
 HideAll;
 ResizeDrawing;
 ShowDrawing;
 SetRect(TextRect, 1, 190, 510, 340);
 SetTextRect(TextRect);
 ShowText;
 Initialize;
 TryYourLuck;
end.
```

A Friendly P.S.

Our joint Mac Pascal adventure has now come to an end, but don't let your own adventure end here. There are many fascinating and useful things you can do with your Macintosh computer now that you have a good grounding in Macintosh Pascal; a whole new world awaits you!

We have not, of course, covered everything there is to know about Macintosh Pascal. We *have* tried to cover the basic material: the most important procedures, functions, statements, and concepts. But Mac Pascal is a particularly rich language, and no one book can do justice to it. We hope we have given you the tools to continue your exploration of the language in all of its depth.

Solutions to Selected Mac-r-cises

CHAPTER 2

1. Only the semicolon following the line "x: = x + 5" can be safely eliminated. The lines that appear between the second Begin and the first End statement comprise a *compound* statement in which the *last* statement before the "end" may be entered without the semicolon.

2. The value 280 (approximately) will result in the rectangles disappearing off the bottom of the Drawing window, but you can go to nearly 500 (actually somewhere around 480, depending on how accurately you fill the space within the Drawing window) before the rectangles' sides disappear off the right edge of the window.

CHAPTER 3

1. The following list describes which identifiers are acceptable or not to Mac, and why:
   a. This variable is OK, even though its name may not make you think so!
   b. Unacceptable. Starts with a number.
   c. Unacceptable. Has a space in it.
   d. Unacceptable. The hyphen is a special character.
   e. OK.
   f. Unacceptable. The question mark is a special character.
   g. OK.
   h. OK. Numbers are permitted in names except for the first position.
   i. OK.

2. The program has four errors in it: (1) There is no *var* identifier to tell the program that x, y, and 1name are being declared as variables. (2) 3name is an illegal variable name because it begins with a number. (3) The comment line lacks a closing curly bracket, which means that the program will keep looking for the end of the comment and, not finding it, will produce an error. (4) There should not be a semicolon after the word *begin*.

3.

a. Integer. Golf scores are always hole...er, *whole*...numbers.
b. Real. Dollars and cents values almost always have decimal points.
c. String. Even though a Social Security number is made up mostly of numbers, the fact that it has two hyphens or dashes in it makes it alphabetic and not numerical.
d. Integer. Any time we are concerned with *counts*, integer is the right variable type because we cannot count a fractional part of a loop.
e. Real. In all likelihood, the average word length in Pascal's famous philosophical treatise won't be an exactly whole number.
f. Character or String. Any time a character variable will work, so will a string, but the opposite is not true.
g. Real. Pi is 3.14159 and a lot more numbers, clearly a decimal value that would require a variable of type Real to hold it.

4. This program is not really so difficult, but at this stage in our Macintosh Pascal adventure, it uses things you may not yet understand. If you got it, bravo! If not, don't worry about it. It will soon seem trivial to you.

```
program PoundsToKilograms;
var
  weight:real;
begin
  Writeln('Enter the weight in pounds.');
  Readln(weight);
  Writeln('In kilograms, that will be:');
  Writeln(weight*0.453592);
end.
```

CHAPTER 4

1. This modification is relatively straightforward. We define a new variable, AgeInHours to be of type Real. Then we multiply AgeInDays by 24, put the result in the new variable AgeInHours, and add a Writeln statement after the current Writeln statement that tells the user his or her age in days. (We could also define a new constant, HoursInDay, to be 24 and use it instead of the number 24 in the calculation part of the program.) The newly finished program would look like this:

```
program HourTeller;
 const
  DaysInYear = 365.25;
 var
  name:string[80];
  age:integer;
  AgeInDays,AgeInHours:real;
```

```
begin
  Writeln('What is your name, please?');
  Readln(name);
  Writeln('How many years old are you, ',name,'?');
  Readln(age);
  AgeInDays: = age*DaysInYear;
  AgeInHours: = AgeInDays*24;
  Writeln('That means you are about ',AgeInDays,' days old,');
  Writeln('which translates to ',AgeInHours,'hours!');
end.
```

2. One possible solution:

```
program CheckCheck;
var
  hours,rate,gross,net,tax,deductions:real;
begin
  Write('How many hours: ');
  Readln(hours);
  Write('Hourly rate: $');          ← Note that we supply the dollar sign, so user doesn't
  Readln(rate);
  Write('What tax percentage? (enter as a decimal): ');
  Readln(tax);
  Write('Total of other deductions = $');
  Readln(deductions);
  gross: = hours*rate;
  net: = gross-(gross*tax)-deductions;
  Writeln('Your gross pay was $',gross:1:2);
  Writeln('Your check should be $',net:1:2);
end.
```

Note that we supplied the dollar sign in several places where we asked the user for input. If we hadn't done so, some user might type in the dollar sign (or try to); what would be the result? This provides a good example of using good question formatting to elicit the answers we want from users in the form in which the program is expecting those answers.

3. Here's one way to do this:

```
program Divider;
  var
    number1,number2:integer;
  begin
    Write('First number: ');
    Readln(number1);
    Write('Second number: ');
    Readln(number2);
    Writeln((number1/number2):1:2);
  end.
```

4. We have to add a variable of type Real to hold the result, since a division is not guaranteed to produce a whole-number answer. (In Chapter 8, we'll learn more about number manipulation and some other division tricks.)

```
program Divider2;
 var
  number1,number2:integer;
  quotient:real;
 begin
  Write('First number: ');
  Readln(number1);
  Write('Second number: ');
  Readln(number2);
  quotient: = number1/number2;
  Writeln(quotient:1:2);
 end.
```

5. This program is difficult only because we haven't yet talked about things like While statements and how they work. If you got something close to this answer, consider yourself ahead of the crowd!

```
program UserFirst;
 var
  x,limit,increase:integer;
 begin
  Writeln('To what limit in the Drawing window');
  Writeln('would you like me to draw?');          ← Notice multiple-line question
  Readln(limit);
  Writeln('How much space between squares (1-20)? ');     ← Helping the user
  Readln(increase);
  {rest of program is same as "First"}
  while x< = limit do
     begin
        FrameRect(0,0,x,x);
        x: = x + increase;
     end;
 end.
```

CHAPTER 5

1. The key to the "I don't know you" part of the program, which is really the only tricky part, is the use of a Repeat...Until loop. Here's our solution (there are, of course, others).

```
program AgeTeller2;
 const
  person1 = 'Ludwig von Beethoven';
  person2 = 'Ernest Hemingway';
```

```
  person3 = 'Betsy Ross';
  age1 = 215;
  age2 = 96;
  age3 = 233;
 var
  name:string;
 begin
  repeat
   begin
    Writeln('What is your name?');
    Readln(name);
    if name = person1 then Writeln ('You are ',age1,' years old.')      ← No semicolon
     else if name = person2 then Writeln ('You are ',age2,' years
     old.')
     else if name = person3 then Writeln ('You are ',age3,' years
     old.')
        else Writeln ('I guess I don''t know you.');
    end;
   until name = 'Stop';
 end.
```

2. The key idea here is that of *swapping* variables. To accomplish this, we need a third, intermediate variable which we here call temp. Our solution:

```
program SortNumbers;
 var
  smaller,larger,temp:integer;
 begin
  Write('Enter the first number: ');
  Readln(smaller);
  Write('Enter the second number: ');
  Readln(larger);
  if smaller>larger then
   begin
     temp: = larger;
     larger: = smaller;
     smaller: = temp;
   end;
  {no Else needed here}
  Writeln ('The larger number: ',larger);
  Writeln ('The smaller number: ',smaller);
 end.
```

3. The program would run as follows:

```
program PrintThrees;
var
  value:integer;
```

```
begin
  for value: = 1 to 30 do
   Writeln(3*value);
end.
```

4. Here's one way of approaching the problem.

```
Program BuyFillets;
 const
  PricePerPound = 4.89;
 var
    pounds,ounces:integer;
    bill,PricePerOunce:real
    answer:char;
 begin
  PricePerOunce = PricePerPound/16;
    repeat
      Writeln('How many pounds of fillets');
      Write('do you have?');
      Readln(pounds);
      Write('And how many ounces?');
      Readln(ounces);
      bill: = (((16*pounds) + ounces)*PricePerOunce);
      Writeln('That will be $',bill:1:2);
      Writeln;
      Writeln('Is that OK?');
      Readln(answer);
     until answer  = 'y';
  Writeln('Thank you very much!');
end.
```

5. This one is actually much easier than it sounds, but it does use the idea of letter comparison, which we haven't yet discussed; that's why we rated it difficult. Here's our solution:

```
program WhichHalf;
var
  word:string[20];
begin
  repeat
    Writeln('Enter your next word (Use capitals):');
    Readln(word);
    if word >'M' then Writeln ('That is in the second half of the
    alphabet.')
       else Writeln ('That is in the first half of the alphabet.');
  until word = 'STOP';
end.
```

(If the first letter of the word entered is after the letter *M* in the alphabet, the word is in the second half of the alphabet. Words must be entered with initial capital letters or in all capital letters, since you haven't yet learned how to differentiate between upper- and lowercase for the purpose of string comparison.)

CHAPTER 6

1. Here's our solution. Yours may, of course, vary in terms of variable names, size and position of oval and mouth, and so on. The important question is, "Does it work?"

```
program Talker;
var
  x,y,z:integer;
begin
  PaintOval(75,75,190,155);
  for x: = 1 to 10 do
    begin
      InvertOval(150,90,170,140);
      for y: = 1 to 100 do        ← This is a "delay loop"
        z: = z + 1;
      InvertOval(150,90,170,140);
      for y: = 1 to 100 do
        z: = z + 1;
    end;
end.
```

2. Here's one way to approach it.

```
program StackOfBoxes;
begin
  PaintRect(20,20,120,120);
  EraseRect(30,30,110,110);
  PaintRect(40,40,100,100);
  EraseRect(50,50,90,90);
  PaintRect(60,60,80,80);
  EraseRect(65,65,75,75);
end.
```

The planning comes in when you decide where to begin the series of boxes and the amount by which to vary the border limits so the finished drawing will have a neat appearance. How does yours look?

3. No solution furnished. (This Appendix *is* called "Selected" solutions, after all!)

CHAPTER 7

1. The changes are relatively simple: Add a variable to which the word to be deleted can be assigned, a line to ask for, and a line to input that word. Change the Omit line so that it uses the Length procedure to determine where the word and its following blank occur.

2. Here's how we solved this one:

```
program MakePalindrome;
var
  phrase1,phrase2:string;
  len1,count:integer;
begin
  phrase2:=' ';          ← Initialize the variable Phrase2 to be one blank long
  Writeln('Enter the phrase:');
  Readln(phrase1);
  len1:=length(phrase1);
  for count:=len1 downto 1 do
   phrase2:=concat(phrase2,(copy(phrase1,count,1)));
      {start at last position; reassemble letters in reverse}
  phrase2:=copy(phrase2,2,count); {get rid of blank we started with}
  phrase1:=concat(phrase1,phrase2);
  Writeln(phrase1);
end.
```

3. We leave this one to you. Basically, it will require you to do only one thing we haven't done yet: Copy the string that is entered over to a new string, one character at a time. In the process, check to be sure you are copying only letters (using the ">=" and "<=" logical operators). Use the new string to create a third string that reassembles the letters in reverse order. Finally, check the two new strings to see if they're equal.

4. One way to approach this assignment follows.

```
program BuildText;
 var
  c:char;
  line:string;
 begin
   line:=' ';
   repeat
    read(c);
    line:=concat(line,c);
    if length(line)=80 then
     begin
      Writeln(line);
      line:=' ';
    end;
```

```
  until c = '}';,        ← This is how we handled the hint
  Writeln;
  Writeln(line);
  Writeln(Done!);
end.
```

CHAPTER 8

1. This solution will work:

```
program CoinToss;
var
  totheads,tottails,numtosses,count,number:integer;
begin
  totheads: = 0;
  tottails: = 0;
  Writeln('How many coin tosses');
  Write('would you like? ');
  Readln(numtosses);
  for count: = 1 to numtosses do
    begin
      number: = (random mod 2) + 1;
      if number = 1 then
          totheads: = totheads + 1
        else tottails: = tottails + 1;
    end;
  Writeln('Out of ',numtosses,' coin tosses,');
  Writeln('I got ',totheads,' heads and ',tottails,' tails.');
end.
```

2. The first modification requires that you declare and initialize a variable (call it, say, Totright) to keep track of the total right answers. Be sure to initialize this variable to be zero when the program starts each time. Then calculate a percentage by dividing the Totright by the total number of problems and multiplying the result by 100. The second modification requires only that you set up a For loop with a counter from 1 to 3 that checks after each answer to see if it's right before printing the "wrong" message. We'll leave these modifications to you.

3. Here's our dice-rolling program. There are thousands of variations on this theme.

```
program RollTheDice;
var
  die1,die2,total:integer;
  ans:char;
```

```
begin
  repeat
    die1:=(random mod 6)+1;
    die2:=(random mod 6)+1;
    total:=die1+die2;
    Writeln('You rolled a ',die1,' and a ',die2);
    Writeln('for a total of ',total,'.');
    Writeln;
    Write('Another roll? ');
    Readln(ans);
  until ((ans='N') or (ans='n'));
end.
```

4. The general form for a quadratic equation is: $ax^2 + bx + c = 0$. In solving such an equation, we are given the values of a, b, and c and our job is to find the value of x. Actually, x can always have two values. The formula for solving such an equation is:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The program we wrote for solving this kind of equation follows:

```
program Quadratic;
var
  a,b,c,x,root1,root2:real;
begin
  Write('Coefficient a: ');
  Readln(a);
  Write('Coefficient b: ');
  Readln(b);
  Write('Coefficient c: ');
  Readln(c);
  root1:=-b+(sqrt(b*b-(4*a*c)))/2*a;
  root2:=-b-(sqrt(b*b-(4*a*c)))/2*a;
 Writeln('The roots are:');
 Writeln(root1   :1);
 Writeln(root2:1);
end.
```

CHAPTER 9

1. Here's how we solved this puzzler.

```
program WordCounter;
var
  s:string;
function countit(s:string):integer;
  var
  cnt,num:integer;
  begin
  repeat
  begin
    cnt:=cnt+1;        ← We know we have at least one word
    num:=pos(' ',s);       ← Find the space
    s:=copy(s,num+1,length(s)-num+1);       ← Drop the word we've counted
  end;
    until num = 0;
    countit:=cnt;      ← Assign count to variable to be displayed
 end; {countit}
 begin {main program}
      Writeln('Type any sentence up to 255 characters:');
      Readln(s);
      Writeln;
    while length(s)>0 do
      begin
       Write(countit(s):1:1,'words.');        ← Explicit function call
       Writeln;
       Writeln('Type another:');
       Readln(s);
      end;
end.
```

2. Naturally, the answer will depend on the program you picked!

3. This one's too much fun for us to spoil it for you.

CHAPTER 10

1. Rather than reproduce the entire program again, we'll just show how we modified the main program to call the new procedure and then give you the new procedure. As usual, there are many ways of solving this problem, so don't be concerned if yours is different — as long as it works.

   Here's our modification to the main program. In place of the line that now says:

```
Writeln('Give someone the axe?');
```

we code the following:

```
Writeln('Type "Y" to give someone the axe');
Writeln('or "L" to see the list.');
```

Then we modify the If construction just below the line that does a Copy of the answer to extract the first character and come up with this:

```
if ((ans = 'Y') or (ans = 'y')) then
    fickle
  else if ((ans = 'L') or (ans = 'l')) then
      ListFickle
  else
      Writeln('Done!');
```

The procedure ListFickle is fairly straightforward. We "fancied it up" a bit so it prints a heading, which is, of course, optional:

```
procedure ListFickle;
    begin

    Writeln('Name                          Age    Field');
    Writeln('-------------------------------');
    Writeln;
    for row: = 1 to 5 do
      Writeln(favorites[row,1],favorites[row,2]:2,favorites[row,3]);
  end;
```

(You may have to enlarge the Text Window to get this to display properly, especially if you've entered some names or fields that are quite long.)

2. Here's our solution to this one.

```
program TableToList;
var
  row,column,count:integer;
  table:array[1..5,1..3] of integer;
  list:array[1..15]of integer;
procedure MakeTable;
begin
  for row: = 1 to 5 do
    begin
      for column: = 1 to 3 do
        begin
          Write('Enter a variable: ');
          Readln(table[row,column]);
      end; {columns}
    end; {rows}
```

```
   end;{MakeTable}
procedure MakeList;
begin
   count: = 1;
   for row: = 1 to 5 do
     begin
       for column: = 1 to 3 do
       begin
          list[count]: = table[row,column];
          count: = count = 1;
          end;{column}
       end;{row}
   end;{MakeList}
procedure ShowList;
begin
  for count: = 1 to 15 do
    Write(list[count]:2,' ');
end;{ShowList}
begin{main program}
  MakeTable;
  MakeList;
  ShowList;
end.
```

3. We won't show the whole solution here, since the process of creating and displaying the table should by now be old hat to you. But here's the procedure we called Transpose:

```
procedure Transpose;
begin
   for row: = 1 to 4 do
     begin
      for column: = 1 to 4 do
        table2[column,row]: = table[row,column]; {table2 is the one being created}
     end;
end;{procedure}
```

We assume that Row and Column are defined as integers and that both Table1 and Table2 are defined as 4x4 arrays.

4. This one is labeled difficult only because it requires the translation of numbers to pictures, a process that some people find intuitively hard. Here's one solution that works.

```
program BarGraph;
var
   count,top,bottom:integer;
   numbers:array[1..6] of integer;
```

```
begin
  for count: = 1 to 6 do
    begin
      Writeln('Give me a number');
      Write('between 1 and 20: ');      ← See text for why we chose 20 here
      Readln(numbers[count]);
    end;
  for count: = 1 to 6 do
    begin
    top: = count*20;        ← Ensures spacing between bars
    bottom: = top + 10;       ← All bars are same depth
    PaintRect(top,0,bottom,numbers[count]*10);      ← See text
  end;
end.
```

We chose to limit input values to 1 to 20 and then to scale them by 10 so that we'd stay inside the normal boundaries of the Drawing window and still be able to detect differences between values. It is possible, for example, to permit values from 1 to 200 and do no scaling, but detecting the difference between 140 and 141 in that situation may take better eyesight than most of us have. (In doing any kind of graph or chart, the scaling factor is the decision most crucial to the readability of the output.)

CHAPTER 11

1.

a. This set consists of people who are *both* men *and* women. No such person is in our sets.

b. This set will contain Ron, Sam, Norma, and Meredith (that is, all the people in the "Strangers" set).

c. This set will also have only the strangers in it. The intersection of the sets of friends, women, and men is an empty set (because, remember, nobody is both a woman and a man in our sets). When we find the union of that empty set and the set containing strangers, we get only the strangers.

2. Here is one way to solve the problem.

```
program ChessCount;
  type
  ChessSet = record
    piece:char;        ← Could use string here and full names if desired
    value:integer;
      end;
      var
      setup:array[1..5] of ChessSet;
      points, count, num, mypoints, yourpoints:integer;
```

```
    procedure PointCount;
    begin
      points: = 0;
      for count: = 1 to 5 do
        begin
          Write('How many ',setup[count].piece,'?');
          Readln(num);
          Points: = points = (num*setup[count].value);        ← See text end;
  end;{PointCount}
 begin {main program}
  setup[1].piece: = 'Q';
  setup[2].piece: = 'R';
  setup[3].piece: = 'N';
  setup[4].piece: = 'B';
  setup[5].piece: = 'P';
  setup[1].value: = 10;
  setup[2].value: = 5;
  setup[3].value: = 3;
  setup[4].value: = 3;
  setup[5].value: = 1;
  Writeln('First we''ll do your pieces....');
  PointCount;
  mypoints: = points + 2;        ← Could've initialized in procedure, too!
  Writeln('You have ',mypoints:2,' points.');
  Writeln('Now your opponent....');
  PointCount;
  yourpoints: = points + 2;
  Writeln('Your opponent has ',yourpoints:2,' points.');
end.
```

Note the line that was marked as the key line. This statement uses the index variable Count to locate the value of the piece in the record Setup and multiply it by the number of those pieces the player says he has. This total is then added to the current running total.

3. We'll let you work out your own solution to this one!

CHAPTER 12

1. No solution suggested.

2. Here's how we solved this one (but see the note at the end of the program for a better idea!):

```
program ButtonMaker;{because the box looks like a Mac button}
 var
  x,y:Integer;
```

```
begin
    FrameRoundRect(25,0,65,150,15,25);
    moveto(2,50);
    textfont(0); {Chicago font...pick whichever you like!}
    textsize(18); {18-point Chicago just fits, though!}
    drawstring('This is a button');
    repeat
      GetMouse(y,x);
 until button;
  if ((y> = 0) and (y< = 150) and (x> = 25) and (x< = 65)) then
    InvertRoundRect(25,0,65,150,15,25)          ← Remember: no semicolon!
 else
   sysbeep(5);
end.
```

There is a procedure called PtInRect (described in Appendix C of the *Macintosh Pascal User's Guide*) which can be used to determine whether a given point (such as the place pointed to by the mouse) is inside the bounds of a rectangle. We haven't explored that procedure, but its use could shorten this program somewhat.

CHAPTER 13

No solutions.

CHAPTER 14

No solutions.

CHAPTER 15

No solutions.

CHAPTER 16

No solutions.

CHAPTER 17

No solutions.

Index

*, 150
+, 150
-, 150
/, 150
:=, 82
<, 75, 92, 150
<=, 75, 92, 150
<>, 150
= operator, 91
=, 75, 92, 150
>, 75, 92, 150
>= operators, 92
>=, 75, 150
∧, 251, 253

ASCII, 279, 281, 282
Abs, 145, 146
Absolute value, 146
Abstract art, 114
Accessing dynamic variables, 255
Addition, 150
Address, 255
Advanced graphics, 239
AgeTeller program, 53, 50, 52, 54, 55, 56, 57, 60, 67, 96
Alert box, 13, 19
Alphabetical order, 257
Alphanumeric, 125, 126, 58
American Standard Code for Information Interchange, 281
Amplitude, 270
Animation, 112
Annuity, 153, 154
Apostrophe, 65
Apple menu, 269
Arctan, 152
Arctangent, 152
Arithmetic operators, 150
Array, 139, 177, 180, 181, 182, 183, 184, 185, 189, 208, 209, 212, 249, 250, 251, 256
  of records, 209, 210, 212
ArrayDismay program, 183
Art work, 118, 121
Artistic, 114
Assignment
  operator, 52
  statement, 183, 212
Auto repeating key, 225, 226
Average, 88, 89
Averages program, 88

BASIC, 29, 31
BadExample program, 48
Bar
  graph, 98
  Menu, 3, 5, 13, 21, 85, 87
  Scroll, 23
Beep, 34, 268, 269
Begin, 100, 106, 112, 113, 118, 119, 12, 121, 127, 129, 18, 19, 30, 31, 33, 34, 36, 38, 4, 52, 77, 78, 83, 84, 88, 89, 90, 91, 95, 99
Begin-End, 35
  pair, 33, 78
BetterShopper program, 259
Blackjack, 93
Blank line, 75
Boldface, 4, 12, 17, 18
Bool1 program, 91
Boolean, 90, 91, 146, 218, 283
  experience, 93
  type, 91, 92
  variable, 92
Boundary coordinate, 97
Box, 108
  Alert, 13, 19
  Bug, 33, 35, 45, 57, 60, 77, 80, 165, 171
  Catalog, 23
  Dialog, 8, 21, 22, 23
  Program, 21
  Size, 4, 28
Brackets, 42, 46
Bug, 24
  box, 33, 35, 45, 57, 60, 77, 80, 165, 171
Button, 218, 219
  Discard, 23
  Do It, 64, 65, 240
  Mouse, 95
  Not, 90
  OK, 22
  Save, 21
Bytes, 280

Calculate, 78, 81
Capital, 71
Case construct, 80
Case, 113
  of, 79
  statement, 78, 79, 80, 112, 115, 116, 117

Catalog box, 23
CatchTheThe program, 135
Char, 42, 43, 44, 74, 77, 91, 119
  type, 252
Character variables, 58
Characters, 132, 133
Check option, 7, 19, 43, 51, 53, 79
ChuckALuck program, 277, 287, 288
Circle, 109, 172, 174
Circumflex, 251, 252, 255
Cleaning up, 236
Clear, 5
Clipboard, 5
  window, 8
Clock, 263
Close option, 23, 37, 231, 232, 234, 235
Colon, 18, 37, 41, 70, 79, 145, 232
Column, 179, 210
Combining strings, 131
Comma, 66, 77, 129, 132
Command
  Cursor, 239
  Do, 82
  Drawing, 102
  Graphics, 97
  HideAll, 122
  Invert, 111
  options, 5
  PenPat, 112
Comments, 97, 164
Compare strings, 129
Compiler, 12
Complex programs, 78
Compound, 154
  interest, 153
  statement, 83, 84, 94
Computer
  animation, 112
  art, 97
  program, 11, 13
Concat, 131, 135, 139
Concatenation, 131
Condition, 70, 71, 72, 73, 75, 93
  expression, 70
  matching, 73
Conditional
  processing, 70
  program statement, 70
  statements, 78

Const, 193
Constant, 31, 54, 58, 172
  values, 174
Control
  loop, 93
  panel, 269
  statements, 69
Coordinates, 90
Copy, 132, 135, 139
  hard, 22
Cos, 152
Count, 85
  Backwards, 86
CountTo10 program, 82, 83
CountWords, 158
Counter variable, 166
Create, 112
Cross hair, 87
Curly brackets, 16, 32, 33, 34, 120
Current pointer (CurrentPtr), 262
Cursor, 239, 240, 242
  commands, 239
  manipulation, 239
Cut option, 13, 17
Cycles, 267

Data, 59, 65, 98, 125, 183, 230
  file, 257
  items, 230
  structure, 257
  types, 190, 196
Date, 264
DateTimeRec, 264, 265
DateTimer program, 264
Day, date and time, 263
DayFinder program, 79, 198
Debugging, 24, 28
Decimal point, 142, 143, 144
Declaration statements, 42
Declarations, 30, 36
Declare, 36, 37, 41, 171
Default pen size, 103
Delete, 16, 17, 37, 125, 135, 136
Designing arrays, 189
Desktop, 30, 33, 34, 35, 38, 43, 64, 67, 90, 106, 242, 244, 245, 247
Dialog box, 8, 21, 22, 23
Dice, 287
Difference of two sets, 203
Difference operator, 203
Discard, 23, 37
  button, 23
Disk, 23, 27, 50, 51, 235, 236
  drives, 50
  file, 230, 234, 237
DiskRead program, 235
Diskette, 229, 237
  inserting, 1
Diskwriter program, 234, 235
Display, 64
Displaying program results, 98
Division, 67, 150
Divisor, 148
Do command, 82, 83, 85, 86, 88, 89, 90, 95, 106, 112, 113
Do It, 10, 11, 13
  button, 64, 65, 240
Double apostrophe, 75
Double precision, 144
Down-pointing thumb, 12, 19, 35
Downto, 86
Draft quality print, 22
Dragging program, 220
Draw, 116
DrawString statement, 118, 119, 120, 123
DrawTriangles program, 169
Drawing, 122
  command, 102
  window, 3, 4, 8, 20, 23, 27, 28, 85, 100, 102, 103, 106, 109, 112, 114, 115, 119, 120, 121, 122, 123, 239, 243, 244, 245
DumpFile program, 277, 279, 281, 282, 283
Duration, 267, 270, 275
Dynamic data, 249, 250, 251
  structure, 251, 252, 256, 258, 259, 262
  data variables, 252
Dynamic variable, 255, 250, 256
Dynamic1 program, 251

EOF, 236
Edit menu, 5, 6, 13
Editing, 19
Else...If...Then, 73, 74, 75, 76, 77, 80
Else, 72, 74, 75, 76, 77, 84, 92
EmptyRect, 247
End, 4, 12, 18, 30, 31, 33, 34, 35, 36, 38, 56, 77, 78, 79, 80, 82, 83, 84, 88, 89, 90, 92, 95, 99, 106, 107, 112, 113, 118, 121, 127, 129, 100
  of file, 236
Enumerated data type, 196, 197
Equal, 74
Erase, 109, 110, 111, 123
EraseOval, 10, 111
EraseRect, 111, 246, 247
EraseRoundRect, 111
Erasing, 112
Error checking, 165
Error message, 65
Even, 83
Event, 221
  Manager, 215, 224, 225, 227, 228
  code, 222, 226
  queue, 215, 221, 222, 223, 224, 226
  record, 222
EventAvail, 227, 228
EventQueue program, 226
EventRecord, 227
Everywhere option, 6
Execute, 85, 93
Execution, 69, 81
Exp function, 152
ExploreDraw program, 99, 100, 101, 102, 103
Exponent, 142, 143
Exponentiation, 153
Expression, 37
Extended precision, 144
Extract1 program, 133

Factor, 151
False, 91, 92
FickleFavorites program, 187
Field, 206, 207, 210, 230, 261
  name, 210
Fields within records, 31
File, 208, 229, 233, 234, 235, 236, 258
  Data, 257
  Disk, 230, 234, 237
  Font, 118
  menu, 2, 5, 6, 21, 22, 23, 26, 33, 37, 63, 121
  of records, 256
  of text, 230
  Printer and disk, 229
  Text, 231, 233, 237
Fill, 109, 111, 123
FillOval, 111
FillRect, 111
FillRoundRect, 111, 121
Find option, 6
FindMouse program, 90, 91
FindString program, 129
First, 16, 37
  program, 18, 20, 21, 25, 27, 38, 40, 41
FixMul, 152
FixRatio, 152
FixRound, 152
Fixed-Point arithmetic, 152
Flasher program, 111, 112
Flow, 69
FlyingCircles program, 172, 173
Font files, 118
Fonts, 118
For, 82, 83, 84, 85, 86, 88, 106, 112, 113
For construct, 87
For...DownTo, 86

For Loop, 82, 83, 84, 85, 86, 87, 88, 89, 95, 178, 187
For...To...Do, 82, 85, 88, 112, 113, 132, 133
Frame, 108, 110, 111, 123
FrameOval, 111, 113, 115
FrameRect, 85, 109, 111, 113, 118, 119, 121, 122, 246, 247
FrameRoundRect, 111, 113
Frequency, 267, 270, 271, 274, 275
Friendly, 77
Function, 31, 136, 141, 145, 156, 157, 158, 159, 174, 175

Games and Utilities for the Macintosh, 276
GetDrawingRect, 244, 245
GetMouse, 90, 99, 160, 169, 224
GetNext Event function, 224, 226, 227
Global, 224
  setting, 224
  variables, 174
Go option, 7, 20, 26, 34, 81
Go-Go, 7
GolfScores1 program, 180, 182, 183
Grapes program, 159, 176
Graph, Bar, 98
Graphic shapes, 170, 174
Graphics, 1, 85, 86, 97, 98, 103, 105, 111, 122, 123, 224, 287
  commands, 97
  mode, 4
  Program, 97
  QuickDraw, 20, 97, 115, 116, 117, 123
Graphs, 98
Greater than or equal to, 74
GridMaker program, 277, 278
GuessIt program, 75, 76

Halt option, 21, 87
Hand, 12
"Happy Days Are Here Again", 271, 273
HappyDays program, 272
Hard copy, 22
Herz, 267
Hex, 280, 281
Hexadecimal, 153, 279, 281
  notation, 280
HideAll command, 121, 122, 123, 242, 244
HideCursor, 240, 241
High quality print, 22
Higher-level computer languages, 41
Horizontal coordinate, 90
Housekeeping, 54, 236
HowLong program, 265
Hz, 267

I/O, 50, 57
Icon, 1, 2, 33, 98, 101
Identifier, 31, 32, 55
If statement, 69, 70, 71, 72, 74, 75, 77, 78, 80, 84, 91, 92, 95, 99, 100
If...Then...Do, 91
If...Then...Else, 72, 74, 75, 76, 77, 84, 89, 92, 96
If...Then...Else...Until, 130
Imagewriter printer, 50, 229, 230
"In" operator, 203
Include, 136, 137, 139
Incompatible Variable Types, 60
Index, 181, 184, 139
Infinite loop, 87
Information, 41
InitCursor, 242
Initial and current pointers, 260
Initial pointer (InitialPtr), 262
  value, 260
Initialize, 52, 53, 59, 61
Input, 49
Insert, 136
Insertion, 125
Instant option, 9
Instant window, 8, 9, 10, 11, 12, 13, 64, 110, 141
Integer, 19, 41, 42, 45, 46, 55, 56, 57, 60, 79, 82, 83, 84, 88, 89, 90, 99, 142, 100, 107, 112, 113, 118, 119, 134, 144, 164
Intersection of two sets, 202
Invert command, 109, 110, 111, 112, 113, 116, 123
InvertOval, 111, 112
InvertRect, 111
InvertRoundRect, 111

Job application, 39

Key pressed, 225, 226
Key released, 225, 226
Keyboard, 59, 67
Keyword, 4, 70

Length, 127, 128, 139
Less than or equal to, 74
LetterChecker program, 70, 72, 74
LineItUp program, 216, 218
LineTo statement, 101, 102, 103, 104, 106, 107, 121, 123
Link, 257
Linked lists, 256, 257, 258
List, 179, 181, 186
Ln function, 152
Local variable, 166, 170, 171
Logarithms, 152
LongInt, 144, 270
Loop, 82, 84, 86, 87, 88, 90, 92, 93, 94, 116, 163, 178, 182
Loops inside loops, 162
Lowercase, 71, 72

MacDraw, 220
Macintosh Pascal Reference Manual, 99, 116, 123, 144, 152, 225
MacPaint, 4, 5, 8, 97, 98, 99, 101, 122, 220
MacStillLife program, 120, 121, 122
MacWrite, 5, 6, 8, 16, 19, 22, 125
Macasso program, 112, 113, 114, 115, 116, 117
Macintosh Pascal Technical Appendix, 154
Manipulating
  pointers, 254
  sets, 200
  strings, 132
Mantissa, 142, 143
Mask codes, 224, 225
Masked event, 224
Math, 141
Membership in a set, 103
Memory location, 255
Menu, 1, 3, 5
  Apple 5
  bar, 3, 5, 13, 21, 85, 87
  Edit, 5, 6, 13
  File, 2, 5, 6, 21, 22, 23, 26, 33, 37, 63, 121
  Run, 6, 7, 11, 12, 19, 20, 23, 25, 34, 37, 43, 51, 67, 81
  Search, 6
  Windows, 8, 37
Message
  code, 222
  field, 226
Metal slide cover, 1
MicroSort program, 130
Mod, 75, 76, 106, 107, 108, 113, 115, 134, 146, 147
Mode, 5
Mode,
  Graphics, 4
  Text, 4
Modem, 50
Modifier, 222
  Field, 226, 227
Mouse, 1, 2, 4, 6, 9, 20, 94, 100, 102, 103, 215, 216, 217, 218, 219, 221, 222, 228
Mouse button, 95
  Down, 225, 226
  Up, 225, 226

MoveTo, 99, 101, 102, 104, 106, 107, 118, 119, 120, 121, 123
MoveWindow, 244
Moving rectangle, 246
Moving, 112
MultTables program, 151
Multiplication, 150
Murphy's Laws, 24
Music, 263
Musical
notes, 274
scale, 274

Name, 32
Naming, 31
Natural logarithm, 152
Negative, 82
Nested
For loops, 186, 187
If statements, 78
loop, 164
New, 249, 252, 259, 260, 261, 262
line, 84
NewAgeTeller program, 63
NewDrawingRect, 244, 245
Nil, 249, 258, 260, 261
Noise wave, 268
Not, 271
button, 90
Note, 270, 275
values, 274
NoteTester program, 271
Null event, 225
Numbers, 45, 64, 67, 74, 79, 75, 105, 142, 143, 144
Numeric variable, 58, 60

OK button, 22
ObscureCursor, 240, 241
Observe option, 37, 61
Observe window, 8, 12, 37, 38, 40, 61
Odd, 83, 145, 146
or even, 141
OddCheck program, 148
OffsetRect, 246
OffsetShow program, 246
OldDrawingRect, 245
Omit, 135, 136, 137, 139
One-dimensional array, 186
Open option, 2, 23, 37
Opening the file, 235
Operations, 113
Option,
Check, 19, 43
Close, 23
command, 5
Cut, 13, 17
Everywhere, 6
Find, 6
Go, 20, 26, 34
Halt, 21
Instant, 9
Observe, 37, 61
Open, 23
Pause, 87
Print..., 22
Quit, 5, 121
Replace, 6
Reset, 10, 11, 23, 25, 67
Revert, 5, 6, 26
Save, 21
Save As..., 21
Select All, 6
Step, 25
Undo, 6
What to Find, 6
Windows, 85, 242
Ord, 198
Origami, 170
Otherwise statement, 80
Output, 49, 50
Formatting, 232
To the printer, 232
Oval, 111, 117, 123, 174
OvalHeight, 117
OvalWidth, 117

Packed array type, 286
Page, 237
Paint, 109, 110, 111, 112, 116, 123
PaintOval, 10, 111, 112, 121
PaintRect, 111
PaintRoundRect, 111
Paper-folding art, 170
Parameter, 101, 102, 104, 113, 117, 118, 127, 128, 129, 136, 156, 166, 168, 169, 174, 270
list, 172
passing, 129, 168
Parenthesis, 65
Pause option, 21, 87
Pen, 109
Patterns, 112, 116, 117
PenNormal, 122, 123
PenPat command, 112, 113, 116, 121, 123
PenSize, 99, 101, 102, 104, 106, 107, 108, 108, 121, 123
Periods, 77
Permanent, recoverable storage: disk files, 233
Picture, 120
Pitch, 267, 268
Planning, 27
Point, 216, 217
Pointer, 43, 66, 87, 255, 258, 261
type, 258, 259
Value, 249
Variable, 253, 254, 249
Pointing hand, 12, 25
Pos, 128, 135, 139
Procedure, 129
Positioning windows, 243
Pred, 198, 199
Print, 114
Draft quality, 22
High quality, 22
Standard quality, 22
... option, 22
PrintIt program, 231
Printer, 230, 231, 235, 236, 237, 282, 283
and disk files, 229
Procedure, 31, 78, 126, 128, 136, 141, 145, 156, 157, 158, 159, 161, 163, 164, 166, 167, 168, 169, 171, 172, 174, 181, 253
Program, 4, 30
AgeTeller, 50, 52, 53, 54, 55, 56, 57, 60, 67, 96
ArrayDismay, 183
Averages, 88
BadExample, 48
Bool1, 91
Box, 21
CatchTheThe, 135
ChuckALuck, 277, 287, 288
Computer, 11, 13
CountTo10, 82, 83
DateTimer, 264
DayFinder, 79, 198
Development, 28
DiskRead, 235
Diskwriter, 234, 235
Dragging, 220
DrawTriangles, 169
DumpFile, 277, 279, 281, 282, 283
EventQueue, 226
ExploreDraw, 99, 100, 101, 103
Extract1, 133
FickleFavorites, 187
FindMouse, 90, 91
FindString, 129
First, 18, 20, 21, 25, 27, 38, 40, 41
Flasher, 111, 112
Flow, 69
FlyingCircles, 172, 173
GolfScores1, 180, 182, 183
Grapes, 159
Graphics, 97
GridMaker, 277, 278
GuessIt, 76
HappyDays, 272
HowLong, 265
LetterChecker, 70, 72, 74
LineItUp, 216, 218
MacStillLife, 120, 121, 122

Macasso, 113
MicroSort, 130
MultTables, 151
NewAgeTeller, 63
NoteTester, 271
OddCheck, 148
OffsetShow, 246
PrintIt, 231
RandomArt1, 106, 108
RandomArt2, 106, 107, 108
ReplaceIt, 137, 138
ScatterLetters, 134
Seasons, 205
SelfImage, 132, 133
Shopper, 210
Show, 3, 36, 43
ShowReals, 143, 144, 145
ShrinkingBox, 247
SquaresTo10, 83, 84, 85, 86
StickStrings, 131
StringLength, 127
Structure, 29
Table1, 186
TextWork, 118, 119
Untitled, 21
WhileAverages, 89
WhileMouse, 95
Window, 3, 16, 30, 35, 37, 64, 106, 118, 121, 243
Programmer's Key, 87, 138
Programming, 26
structured, 78
Window, 23
Programs, complex, 78

Question, 66, 70
Queue, 221, 222, 227, 228
QuickDraw Graphics, 20, 97, 99, 111, 115, 116, 117, 118, 123
Quit option, 5, 33, 121
Quotation mark, 51, 64, 65, 66, 77, 132

Radians, 152
Random, 105, 106, 107, 108, 123, 141, 146, 147
Art, 104, 105,
mod, 105, 113, 148
number, 75, 123
RandomArt1 program, 106, 108
RandomArt2 program, 106, 107, 108
Randomness, 112
Read, 59, 66, 67, 233
from diskette, 229
Readability, 168
Reading, 235
Readln, 49, 51, 59, 60, 61, 62, 63, 66, 67, 70, 74, 75, 79, 88, 89, 91, 118, 119, 127, 129
Real, 42, 46, 54, 55, 56, 57, 60, 88, 89, 142, 143, 144, 164
Record, 193, 206, 207, 209, 212, 230, 257
definition, 208
of information, 230
Rect, 115, 123
Rectangle, 111, 123
round-cornered, 111, 123
Regions, 286
Relational operator, 73, 74, 75, 129
Relays, 24
Remainder, 147
Repeat statement, 69, 94, 95, 99, 100
Repeat...Until, 93, 94, 95, 99, 100, 131
RepeatMouse program, 95
Replace option, 6
ReplaceIt program, 137, 138
Reserved words, 4
Reset option, 7, 10, 11, 20, 23, 25, 45, 67, 87, 235, 236,
Resetting Mac's clock and calendar, 265
Resizing windows, 243
Restoring windows, 243
Revert option, 5, 6, 26
Rewrite, 231, 234, 235, 236, 237
Roots, 149
Round rectangles, 117
Round-cornered rectangles, 111, 123
RoundRect, 116, 123
Shapes, 117
Routing output to screen, 237
Run, 1, 20, 27, 35, 51, 53, 56, 61, 66, 67, 70, 72, 79, 95, 102, 121
menu, 6, 7, 11, 12, 19, 20, 23, 25, 34, 37, 43, 51, 67, 81
window, 10

SANE, 153, 154
Save, 23, 70, 79
button, 21
Save As... option, 21, 63
Scalar, 197
ScatterLetters program, 134
Scientific notation, 143, 145
Scroll bar, 23
Search menu, 6
Search-and-replace, 131
Seasons program, 205
Select All option, 5, 6, 13
SelfImage program, 132, 133
Semicolon, 9, 16, 17, 18, 19, 28, 37, 51, 64, 65, 73, 77, 84
Set, 193, 206
SetCursor, 242
SetDrawing, 121
SetDrawingRect, 122, 123, 244, 245
SetRect statement, 115, 121, 123, 246, 247
SetTune, 273
Sets, 199, 201, 202
Shape, 108, 112, 113, 115, 116, 117
coordinate variables, 116
Shopper program, 210
Show program, 34, 36, 43
ShowCursor, 240, 241
ShowDrawing, 121, 122, 123, 243, 244
ShowReals program, 143, 144, 145
ShowRect, 123
ShowText, 243, 244
Shrinking rectangle, 247
ShrinkingBox program, 247
Sin, 152
Sine, 152
wave, 268
Single-Step, 25, 37, 38
Size box, 4, 28
Social Security numbers, 39
Sound, 255
effects, 267
generation, 266
waves, 267, 268
Speaker, 50, 267
Sqr, 149
Sqrt, 149, 157, 158, 159
Square brackets, 209
Square root, 150, 157, 159
Square wave, 268
tone, 270
Squares, 85, 149, 150
SquaresTo10 program, 83, 84, 85, 86
Standard Apple Numeric Environment, 153
Standard quality print, 22
Statement, 17, 31, 41, 70, 78, 83, 94
Assignment, 212
Case, 78, 79, 80, 112, 115, 116, 119
Control, 69
DrawString, 119
If, 69, 70, 71, 72, 75, 78, 95
LineTo, 102
Nested If, 78
Repeat, 69
SetRect, 115
Then and Else, 72
While, 69
With, 213
Static data, 249, 250
Step option, 7, 12, 25
Step-Step, 7, 12
Stepping, 26
StickStrings program, 131
StillDown, 219, 220

Stops In, 7
Store information, 229
String, 42, 44, 55, 57, 60, 63, 64, 77, 118, 119, 125, 126, 127, 128, 129, 131, 132, 134, 135, 137, 138, 139,
  data, 141
  manipulation, 126, 131
  variables, 125
StringLength program, 127
Structure, 12, 29, 31, 163, 164, 223
Structured
  data type, 180
  programming, 78
Subrange data type, 189, 190, 191, 196, 197
Subtraction, 150
Succ, 198, 199
SysBeep sound, 173, 263, 268, 269
System clock, 263

Table, 179, 181, 185, 186, 187
  array, 187
Table1 program, 186
Tabular reports, 232
Text
  files, 231, 233, 237
  Mode, 4
  type, 231
  window, 3, 4, 8, 10, 23, 43, 64, 61, 64, 65, 67, 75, 100
TextFile1, 235
TextFont, 118, 119, 123
TextSize, 118, 119, 123
TextWork program, 118, 119
Then, 71, 73, 74, 75, 76, 77, 80, 84, 92, 99, 100
  and Else statements, 72
  do, 91
Thumb, down-pointing, 12, 19, 25, 35
TickCount, 266, 270
Time, 263
  of day, 264
To, 82, 83, 112, 113
Triangle, 170, 171
Trigonometric functions, 152
True, 91, 92
Trunc, 164
Two-dimensional array, 185, 186, 187
Two-part record, 216
Type, 31, 144, 193, 194
  Char, 42, 43, 44, 252
  Definition, 195
  Integer, 42, 45, 56
  Packed array, 286
  Pointer, 258, 259
  Real, 42, 46, 54
  Section, 194, 195
  Size, 8
  String, 42, 44, 57
  Text, 231
  Variable, 40, 42

Undo option, 6
Until, 94, 95, 99, 100
Untitled, 30, 31
  program, 21
  window, 3, 8, 16, 33

Value, 60, 66, 82, 85, 87, 90, 157
  out of range, 190
Var, 17, 37, 41, 53, 55, 75, 76, 77, 79, 82, 83, 84, 88, 89, 90, 91, 99, 100, 106, 107, 112, 113, 118, 119, 121, 127, 129, 134, 193, 195
Variable, 17, 29, 31, 37, 39, 40, 41, 47, 49, 52, 53, 54, 55, 56, 57, 59, 60, 61, 62, 66, 67, 82, 85, 86, 87, 90, 91, 108, 115, 117, 128, 133, 169
  assignment operator, 54
  declaration, 167
  declaration section, 41
  name, 41, 42, 66
  parameter, 169
  types, 41, 42, 91
Vertical coordinate, 90
Volume, 267, 270

Wavelength, 267
What field, 225, 226, 227
What to Find option, 6
What value, 226
Where fields, 224
While, 89, 95
  ...Do, 135, 136, 137
  loop, 89, 92, 93, 94
  not...Do, 90, 92, 94, 95
  statement, 69
WhileAverages program, 89, 90
WhileMouse program, 95
Whole number, 41
Window, 1, 3, 4, 16, 242
  Clipboard, 8
  Drawing, 3, 4, 8, 9, 10, 11, 12, 13, 16, 20, 23, 27, 28, 30, 35, 37, 38, 40, 61, 64, 85, 100, 102, 103, 106, 109, 110, 112, 114, 115, 118, 119, 120, 121, 122, 123, 141, 239, 243, 244, 245
  Menu, 8, 37
  Option, 85, 242
  Run, 10
  Text, 3, 4, 8, 10, 23, 43, 61, 64, 65, 67, 75, 100
  Untitled, 3, 4, 8, 16, 33
With
  operator, 213
  statement, 213
Word processor functions, 131
Write, 66, 67, 74, 84, 89, 99, 230, 231, 233
  data, 234
  to diskette, 229
Writeln, 49, 51, 61, 63, 64, 65, 66, 67, 71, 73, 75, 77, 79, 83, 84, 88, 90, 91, 92, 95, 118, 119, 127, 129, 145, 156, 157, 230, 231
Writing in graphics screens, 118, 119

Your declarations, 16, 32

Look for these forthcoming Plume/Waite titles on the Macintosh®:

- ☐ **Games and Utilities for the Macintosh® by Dan Shafer.** Thirty exciting games and useful utility programs in Macintosh Pascal, ready for you to type in and run. Something for everyone, from "Crypto-quotes," "Parachute Man," and "Logic Probe," to sort routines and icon and menu constructors. Full-sized and expertly written, these programs are not only entertaining and useful, they are also a valuable education in the finer points of Macintosh programming.

- ☐ **Hidden Powers of the Macintosh® by Christopher L. Morgan.** This unique, authoritative book takes you behind the Mac's user-friendly facade and shows you how the machine *really* works. Starting simply, the book explains all you need to know to write serious application programs; including QuickDraw and Toolbox routines, windows, pictures, bit maps, regions, events, menus, files, RAM and ROM organization, and much more. Essential for serious programmers, this is the book Apple should have written.

- ☐ **Basic Primer for the Macintosh® by Emil Flock and Miriam Flock.** Apple's own Macintosh Basic is one of the best-structured, fastest, easiest-to-learn versions of Basic ever developed. Using entertaining, carefully graded programming examples, this book takes the complete novice from simple one-line programs to full mastery of the language. Later chapters cover such advanced topics as sound, files, and using the Mac's QuickDraw and Toolbox routines.

- ☐ **Assembly Language Primer for the Macintosh® by Keith Ma** Many serious application programs must be written in assembly language, which alone has the speed and versatility to handle tough problems. Assuming no previous knowledge of assembly language, this book shows you, in easy, step-by-step style, how to master 68000 code, and at the same time, how to access all of the Mac's features from your programs: windows, the mouse, text editing, and more.

Other Plume/Waite books available from New American Library:

- ☐ **Introducing the TRS-80® Model 100, by Diane Burns and S. Venit.** This book, intended for newcomers to the Model 100, offers simple step-by-step explanations of how to set up your Model 100 and how to use its built-in programs: TEXT, ADDRSS, SCHEDL, TELCOM, and BASIC. Specific instructions are given for connecting the Model 100 to the cassette recorder, other computers, the telephone lines, the optional disk drive/video interface, and the optional bar code reader. (255740—$15.95)
- ☐ **Mastering BASIC on the TRS-80® Model 100, by Bernd Enders.** An exceptionally easy-to-follow introduction to the built-in programming language on the Model 100. Also serves as a comprehensive reference guide for the advanced user. Covers all Model 100 BASIC features including graphics, sound, and file-handling. With this book and the Model 100 you can learn BASIC anywhere! (255759—$19.95)
- ☐ **Games and Utilities for the TRS-80® Model 100, by Ron Karr, Steven Olsen, and Robert Lafore.** A collection of powerful programs to enhance your Model 100. Enjoy fast-paced, exciting card games, arcade games, music, art, and learning games. Help yourself to practical utilities that let you count words in a text file, turn your Model 100 into a scientific calculator, show file sizes, and generally increase your Model 100's usefulness, and your own grasp of programming. (255775—$16.95)
- ☐ **Practical Finance on the TRS-80® Model 100, by S. Venit and Diane Burns.** The perfect book for anyone using the Model 100 in business: investors, real estate brokers, managers. Contains short but powerful programs to perform production planning, and access financial and other information from CompuServe® and the Dow Jones News/Retrieval® service. (255767—$15.95)
- ☐ **Hidden Powers of the TRS-80® Model 100, by Christopher L. Morgan.** This amazing book takes you deep inside the Model 100 to reveal for the first time how it really works. You'll learn about the amazing power buried in the ROM, and how to use this power in your own programs. You can print in reverse video, prevent any screen lines from scrolling, dial the telephone from BASIC, control external devices from the cassette port, and discover many other fascinating secrets hidden within your Model 100. (255783—$19.95)

To order, use the convenient coupon on the next page.

Ⓟ

Other Plume/Waite books available from New American Library:

☐ **BASIC PRIMER for the IBM® PC and XT by Bernd Enders and Bob Petersen.** An exceptionally easy-to-follow entry into BASIC programming that also serves as a comprehensive reference guide for the advanced user. Includes thorough coverage of all IBM BASIC features: color graphics, sound, disk access, and floating point. (254957—$16.95)

☐ **DOS PRIMER for the IBM® PC and XT by Mitchell Waite, John Angermeyer and Mark Noble.** An easy-to-understand guide to IBM's disk operating system, versions 1.1 and 2.0, which explains—from the ground up—what a DOS does and how to use it. Also covered are advanced topics such as the fixed disk, tree-structured directories, and redirection. (254949—$14.95)

☐ **PASCAL PRIMER for the IBM® PC by Michael Pardee.** An authoritative guide to this important structured language. Using sound and graphics examples, this book takes the reader from simple concepts to advanced topics such as files, linked lists, compilands, pointers, and the heap. (254965—$17.95)

☐ **ASSEMBLY LANGUAGE PRIMER for the IBM® PC and XT by Robert Lafore.** This unusual book teaches assembly language to the beginner. The author's unique approach, using DEBUG and DOS functions, gets the reader programming fast without the usual confusion and overhead found in most books on this fundamental subject. Covers sound, graphics, and disk access. (254973—$24.95)

☐ **BLUEBOOK OF ASSEMBLY ROUTINES for the IBM® PC and XT by Christopher Morgan.** A collection of expertly written "cookbook" routines that can be plugged in and used in any BASIC, Pascal, or assembly language program. Included are graphics, sound, arithmetic conversions. Get the speed and power of assembly language in your program, even if you don't know the language! (254981—$19.95)

All prices higher in Canada.

Buy them at your local bookstore or use this convenient coupon for ordering.

NEW AMERICAN LIBRARY
P.O. Box 999, Bergenfield, New Jersey 07621

Please send me the PLUME BOOKS I have checked above. I am enclosing $______ (please add $1.50 to this order to cover postage and handling). Send check or money order—no cash or C.O.D.'s Prices and numbers are subject to change without notice.

Name______________________________

Address______________________________

City______________ State______________ Zip Code______________

Allow 4-6 weeks for delivery
This offer subject to withdrawal without notice.

Ⓟ

Computer Guides from PLUME

☐ **THE COMPUTER PHONE BOOK by Mike Cane.** The indispensable guide to personal computer networking. A complete annotated listing of names and numbers so you can go online with over 400 systems across the country. Including information on: free software; electronic mail; computer games; consumer catalogs; medical data; stock market reports; dating services; and much, much more. (254469—$9.95)

☐ **ALMOST FREE COMPUTER STUFF FOR KIDS by Linda Gail Christie and Gary Bullard.** Hundreds of companies across the country offer a tremendous array of products for computer fun and educational challenge at startlingly low prices—or even no cost—if you know where to write. This book tells you all the things you can get and provides the send-away-for coupons you need to enjoy special discounts on everything from software to T-shirts. (255619—$9.95)

☐ **THE COMPUTER FREELANCER'S HANDBOOK: Moonlighting with Your Home Computer by Ardy Friedberg.** This practical guide will show you how you can use your personal computer for extra income. Step-by-step advice, a wealth of real-life success stories, and inspiring ideas offer all the information you'll need for choosing the right home-based business, figuring prices, attracting customers, and growing as much and as fast as you want. (255627—$10.95)

☐ **DATABASE PRIMER: AN EASY-TO-UNDERSTAND GUIDE TO DATABASE MANAGEMENT SYSTEMS by Rose Deakin.** The future of information control is in database management systems—tools that help you organize and manipulate information or data. This essential guide tells you how a database works, what it can do for you, and what you should know when you go to buy one. (254922—$9.95)†

☐ **BEGINNING WITH BASIC: AN INTRODUCTION TO COMPUTER PROGRAMMING by Kent Porter.** Now, at last, the new computer owner has a book that speaks in down-to-earth everyday language to explain clearly—and step-by-step—how to master BASIC, Beginner's All-Purpose Symbolic Instructional Code. And how to use it to program your computer to do exactly what you want it to do. (254914—$10.95)

Prices higher in Canada.

†Not available in Canada.

Buy them at your local bookstore or use this convenient coupon for ordering.

NEW AMERICAN LIBRARY
P.O. Box 999, Bergenfield, New Jersey 07621

Please send me the PLUME BOOKS I have checked above. I am enclosing $__________ (please add $1.50 to this order to cover postage and handling). Send check or money order—no cash or C.O.D.'s. Price and numbers are subject to change without notice.

Name ____

Address ______________________________________

City ________________ State __________ Zip Code ________

Allow 4-6 weeks for delivery. This offer is subject to withdrawal without notice.

COMPUTER • Z5640 • $1[illegible]
CANADA • $2[illegible]

The Waite Group

PASCAL PRIMER FOR THE MACINTOSH®

Whether you're a beginning programmer or an experienced Pascal user on other computers, this book will help you take advantage of all of Macintosh Pascal's exciting special features. Step by step, it shows you how to use its various menus and windows to write statements, create your own programs, and make use of the Macintosh's amazing graphics capabilities.

You'll learn how easy it is to create shapes, fill them with patterns, draw with graphics "pens" of different sizes, and even create your own computer art. You'll also learn how to use such unique Macintosh features as windows, the mouse, the event queue (which can control the flow of your program in new ways), and the built-in clock, and to manipulate the style and size of the letters your program prints on the screen.

Exercises with solutions test your knowledge before you go on to the next chapter. You'll find it easy to master the machine and the exciting new language that are taking homes, schools, and offices by storm.

The Waite Group is a Sausalito, California-based producer of high-quality books on personal computing. Acknowledged as a leader in the industry, the Waite Group has written and produced over thirty titles, including such best-sellers as *Assembly Language Primer for the IBM® PC & XT*, *Bluebook of Assembly Routines for the IBM® PC & XT*, *DOS Primer for the IBM® PC & XT*, and *CP/M® Primer*. Internationally known and award-winning, Waite Group 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. Mr. Waite, President of the Waite Group, has been involved in the computer industry since 1976, when he bought his first Apple I computer from Steven Jobs.

ISBN 0-452-25640-2