

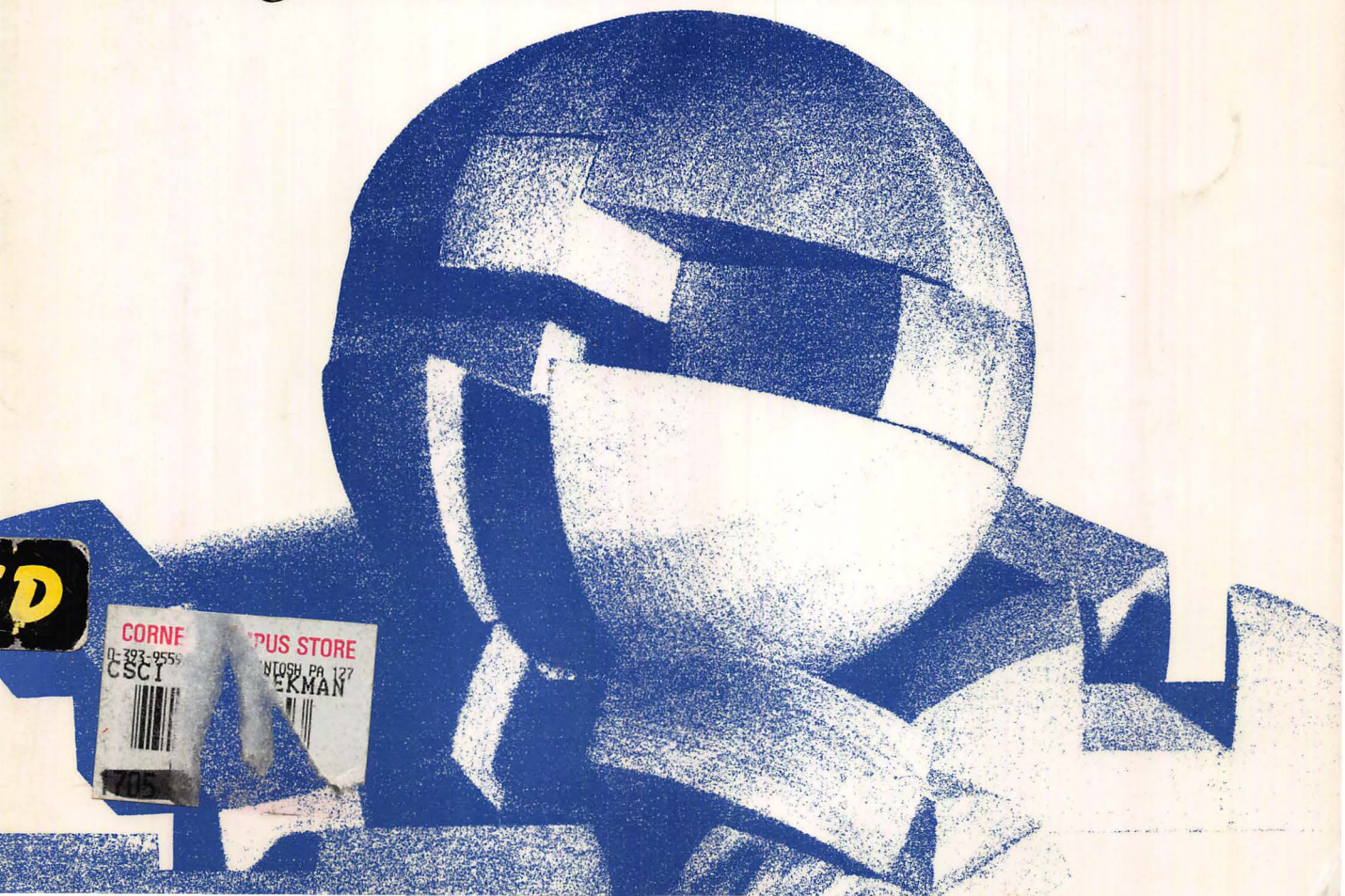
Oh!

by Doug Co

Macintosh

Oh! Pascal!

George Beekman / Michael Johnson





Macintosh

Oh! Pascal!

George Beekman / Michael Johnson

For our families

Susan, Ben, and Josie

Kathleen, Rosemary, Marjorie, Michael, Joanne, and Diane

Macintosh

Oh! Pascal!

George Beekman / Michael Johnson



W·W·NORTON & COMPANY
New York and London



This book was designed and typeset by George Beekman and Michael Johnson using a Macintosh Plus and an Apple LaserWriter printer.

© Copyright 1986 by W. W. Norton & Company, Inc.

ISBN 0-393-95598-2

Table of Contents

	Preface	vii
	A Word to the Student	ix
1	Getting Acquainted with Macintosh Pascal	1
	What is Mac Pascal? 1	
	Programming for Output 2	
	Variables and Input 3	
	Antibugging and Debugging 4	
2	Programming Calculations and Graphics	7
	Assignments and Expressions 7	
	Built-In Functions and Procedures 8	
	Introducing QuickDraw Graphics 9	
3	Procedures and Functions for Problem Solving	17
	Functions as Subprograms 17	
	Antibugging and Debugging 17	
4	Taking Control of Execution: the for Statement	21
	for Statements and Program Actions 21	
	One-Dimensional Arrays* 22	
5	Making Choices: the case Statement	24
	The case Statement 24	
6	Programming Decisions: the if Statement	29
	<i>boolean</i> Expressions and if statements 29	
7	Making Actions Continue: The Conditional Loops	30
	The repeat and while statements 30	
	Antibugging and Debugging 32	
8	Character-Oriented Computing: Text Processing	33
	Text Processing 33	
	The File Window and External Files* 37	
9	Extending the Ordinal Types	43
	Enumerated Ordinal Types 43	

Table of Contents

10	Software Engineering	44
	Writing Programs for People	44
11	Arrays for Random Access	46
	Focus on Programming: Arrays	46
	Strings as Arrays	50
12	E Pluribus Unum: Records	53
	Predefined Record Types	53
13	Files and Text Processing	56
	Making and Using Textfiles	56
	Direct Access Files	58
	Antibugging and Debugging	64
14	Collections of Values: The set Type	66
	Defining and Programming Set Types	66
15	Abstract Data Structures Via Pointers	67
	Pointers in Macintosh Pascal	67
16	Advanced Topics	68
	Macintosh Pascal Extensions	68
	Advanced Programming Resources	70
	Beyond Macintosh Pascal	71
	Appendix: A Hands-On Introduction	73
	Session 1: Meet Mac Pascal (to accompany Chapter 1)	73
	Session 2: Tools and Tricks (to accompany Chapter 2)	87
	Index	97
	Error Messages and Explanations	Inside back cover
	Macintosh Desktop Command Summary	Reference card
	Macintosh Pascal Functions and Procedures	Reference card

Preface

Oh! Pascal! is an exceptionally readable, thorough, and educationally sound introduction to Pascal programming and problem solving. When we first adopted *Oh! Pascal!* at Oregon State in 1982, we were using an aging CDC mainframe running an antiquated version of Pascal. In 1985 we moved our introductory programming classes onto a bushel of shiny Macintoshes. The course will never be the same. The Macintosh Pascal interpreter, with its graphic user interface, smart editor, multiple windows, and powerful debugging options, provides a nearly ideal environment for teaching beginning programming.

Our switch to Macintosh Pascal created one problem: textbook incompatibility. *Oh! Pascal!* teaches Standard Pascal, a language that only roughly resembles implementations used on most of today's microcomputers. String data types, direct access files, and graphics – all common in microcomputer versions of Pascal – aren't included in the original Pascal Standard – or the Cooper and Clancy text. Many features that *were* included in the Standard have been modified and improved in Mac Pascal. These changes make Pascal easier to use and more powerful, but they're hard to teach without a supporting text.

There are at least a dozen Macintosh Pascal books on the market, but *Oh! Pascal!* provides a far better introduction to Pascal, programming, problem solving, and computer science than any of these Mac-specific alternatives. Rather than settling for a book that compromises on Pascal in order to focus on the machine environment, we decided to start with the best available Pascal textbook and develop a bridge between that book and the Macintosh environment.

Oh! Macintosh Pascal! is that bridge. It includes:

- A gentle hands-on introduction to the Macintosh Pascal programming environment.
- Topic-by-topic discussions of the differences between Standard Pascal and Macintosh Pascal.
- Thorough introductions to the most important special features of Mac Pascal, including string processing, file manipulation, and graphics.
- A wealth of programming examples in Macintosh Pascal.
- Supplementary exercises for Mac Pascal programmers.
- A reference section including explanations of common error messages.
- A tear-out quick reference card.

Preface

- An optional supplementary disk containing all of the programs in *Oh! Macintosh Pascal!*, many of the programs in *Oh! Pascal!*, and bonus programs that illustrate with Macintosh graphics some important programming concepts from the text.

Oh! Macintosh Pascal! is designed to accompany *Oh! Pascal!* Topics are generally covered in the same sequence. And like *Oh! Pascal!*, this supplement need not be read in a strict sequential order. Some instructors, for instance, may want to cover the section on using external files earlier than Chapter 8 so they can use data files in their programming assignments. We are careful to introduce Macintosh Pascal's special features where they'll fit most appropriately in an introduction to Pascal. Strings and graphics, for example, come early because they provide powerful, interesting, and understandable tools for learning about procedures and functions.

We're interested in hearing of your experiences with *Oh! Macintosh Pascal!* We can be reached on CSNET at Oregon State (beekman@orstcs, mpj@orstcs) or by good old-fashioned US Mail in the Computer Science Department at Oregon State University, Corvallis, OR 97330.

Acknowledgments

First, thanks to Doug Cooper and Michael Clancy, whose excellent book inspired this effort. We particularly appreciate Doug's encouragement and advice in the early stages of this project. On the editorial end, it's been a pleasure working with Jim Jordan and his staff at W. W. Norton & Co; their fine editing and professional support made our work much easier – and this book much better. We also appreciate the help and information we received from Anjali Magana and the friendly folks at Apple Computer. Locally, we are especially indebted to Karen Meyer-Arendt, Mark Borgerson, Susan Beekman, and Kathy Johnson for their many editorial contributions to this text. And finally, to our families – who have endured so much and seen us so little over the last few months – love and thanks.

*George Beekman
Michael Johnson
October, 1986*

A Word to the Student

You may be wondering why you've been assigned a separate book on the subject of Macintosh Pascal. The answer is that you're lucky...lucky to be learning Pascal on a system that's more interesting (and fun) than the kind of computer system for which the text *Oh! Pascal!* – and the original Pascal language – were conceived. Because you'll use the Macintosh to learn programming, you'll also learn a different – and much easier – way of working with a computer than the one assumed in *Oh! Pascal!* *Oh! Macintosh Pascal* is your guide to the special features of the Macintosh system; it's an essential supplement to the text.

Oh! Pascal! describes what is regarded by the International Standards Organization (ISO) as the Pascal Standard, and every program in *Oh! Pascal!* is a Standard Pascal program. All Standard Pascal programs can be run using Macintosh Pascal (occasionally with minor modifications, as discussed in this supplement). However, since Macintosh Pascal is designed to allow you to write programs that take advantage of the Macintosh computer's special features and capabilities – like its graphics and sound generation hardware – not every program you'll write in Macintosh Pascal will be a Standard Pascal program.

The two main objectives of *Oh! Macintosh Pascal!* are to make clear which elements of Macintosh Pascal are different from Standard Pascal, and to explain the special features of Mac Pascal that you need to know about to be fully conversant with the Macintosh.

You'll find learning Mac Pascal easier if you follow these procedures:

1. Read your assignment in *Oh! Pascal!* That's where you'll learn the nuts and bolts of programming, problem solving, and Standard Pascal.

2. Read the corresponding chapter in *Oh! Macintosh Pascal!* for details of your brand of the language. Each chapter contains notes that modify or expand on statements made in the text. For example, when *Oh! Pascal!* tells you on page 4 that a Standard Pascal identifier may contain any series of letters or digits, *Oh! Macintosh Pascal!* adds that it's OK in Macintosh Pascal to include underscore characters in identifiers. Marginal sideheads with parenthesized page references, like the one to the left of this paragraph, make it easy to find the corresponding passages in *Oh! Pascal!* For sideheads that deal with Macintosh-specific topics that aren't covered in *Oh! Pascal!*, we substitute a tiny Macintosh for the page reference number.

How you read these notes will depend on your personal learning style. Most readers prefer to make an uninterrupted pass through each chapter of *Oh! Pascal!* before consulting *Oh! Macintosh Pascal!* Some read the two books side by side from the beginning, checking for notes and additions as they go along. A few read it both ways. Have it your way....

3. Get your hands dirty. Learning to program is like learning to ride a bicycle; you can't do it without doing it. The two-part tutorial in the appendix is designed to get you up and running on the Macintosh quickly and painlessly. Work through these tutorial sessions after reading Chapters 1 and 2, respectively. But don't stop there. Do the exercises. Try out program examples from the text, the supplement, and the *Oh! Macintosh Pascal!* diskette. Experiment. Have fun!

Learning Mac Pascal

identifiers
(4)



The White Rabbit put on his spectacles. "Where shall I begin, please your majesty?" he asked.

"Begin at the beginning," the king said, very gravely, "and go on till you come to the end: Then stop."

*—Lewis Carroll
Alice in Wonderland*

Getting Acquainted With Macintosh Pascal

MEET MAC PASCAL, the friendliest member of the ever-growing Pascal family. Mac Pascal is particularly nice to strangers. You'll find as you read this chapter and do the followup hands-on session in the Appendix that Mac Pascal does everything short of holding your hand as you take your first programming steps.

What Is Mac Pascal?



language
extensions

user interface

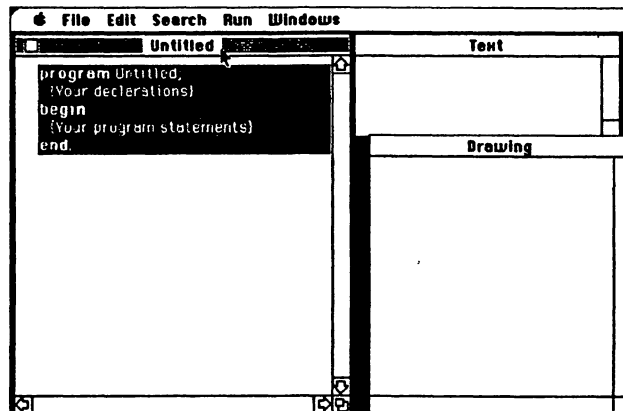
Macintosh Pascal differs from the Standard Pascal described in *Oh! Pascal!* in two ways: the *language extensions* and the *user interface*.

Like most modern versions of Pascal, the Macintosh Pascal language includes several extensions of the Standard. Some of these changes make programming easier and more intuitive; others allow your programs to do things that simply can't be done with Standard Pascal as Wirth defined it. The price you pay for taking advantage of these extensions is a lack of portability. Since the details of language extensions are implementation dependent, programs written to take advantage of Mac Pascal's (or *any* Pascal's) non Standard features may have to be modified considerably to run with other Pascal interpreters or compilers.

Macintosh Pascal's most striking feature is its distinctive *user interface* – the way it communicates with you. Like the Macintosh itself, Mac Pascal is designed to be, above all else, easy to learn and easy to use. But it's also packed with features to make advanced programmers more productive. We discuss general features of the Mac Pascal user interface here; the details are in the hands-on sessions in the Appendix.

The Mac Pascal screen generally displays three standard *windows*:

- the program window, where you'll type your programs and edit (modify) them. This window will display the name of your program, "Untitled" in this example.



- the Text window, which displays textual output from your programs. This window is the place where output from FirstRun, and all other programs in *Oh! Pascal!*, will appear.
- the Drawing window, which displays whatever pictures you create with your programs.

1 Getting Acquainted with Macintosh Pascal

Across the top of the screen is a series of pull-down menus containing commands you'll use to tell Mac Pascal what to do. The hands-on session will show you how easy it is to use those menus.

steps for running
programs
(7)

Macintosh Pascal includes a built-in editor that can be used to enter and edit (modify) programs in the program window. The editor automatically formats each program as you type it in, by taking care of indentation and spacing, and by **boldfacing** reserved words so you don't need to make decisions on those matters. What's more, the Mac Pascal editor points out many of the most common errors as you type your program in, so you can correct them immediately.

Mac Pascal is an interpreter rather than a compiler, which means that each line of your program is translated into machine language as the program is run. This kind of translation-on-the-fly makes programs run slower, but it also saves you the trouble of going through an extra translation step each time you change your program. The interpreter also includes several powerful debugging tools that make it easier to find errors in your programs. These tools will be introduced in later chapters.

We'll explore the Mac Pascal programming environment further in the Antibugging and Debugging section of this chapter. For now, let's focus on extensions to the language itself.

Programming
for Output
(4)

identifiers
(4)

Macintosh Pascal programs look pretty much like Standard Pascal programs. One noticeable difference is the inclusion of the underscore (_) as a valid character in an identifier. Underscores can appear anywhere in an identifier, except at the beginning. For example, in Mac Pascal you can name an identifier *Batting_Average*; standard Pascal insists on *BattingAverage*. Most modern versions of Pascal allow underscores to be included inside identifiers. This extension to the Standard Pascal definition allows you to write programs that are more readable without changing the way they work at all. Of course, programs that include underscores in identifiers won't run properly in older, less permissive versions of Pascal.

Besides accepting the underscore character, Macintosh Pascal also treats identifiers with uppercase and lowercase characters as identical. These are all the same to Mac Pascal:

highnumber

HIGHNUMBER

HighNumber

(The identifier *High_Number* is *not* identical to any of these.)

Comments
(5)

Comments may be surrounded by braces {...} or by {...*}. To enter a multiple-line comment, you can just type a carriage return at the end of each line of the comment, putting a closing brace only on the last line. Mac Pascal will automatically surround each line with braces. (Don't try to insert spaces to make the closing braces line up in the same column; it won't work in Mac Pascal.)

Comments should never be longer than about twenty lines. Longer comments can be typed in, but they may cause serious problems later when you try to run the program.

program heading
(6)

The (input, output) part of the program heading is optional in Macintosh Pascal.

Variables
and Input

Besides the simple types found in the Standard – *integer*, *char*, *boolean*, and *real* – Macintosh Pascal allows several additional types:

- (9) *longint*: *longint* extends the range of integers from -32768 to 32767 for *integer* to -2,147,483,648 to 2,147,483,647. This is still not enough to compute the national debt, but it does allow integer arithmetic over a much larger range than does *integer*.
- Standard types (10) *double*: *double* extends the range and precision of type *real*. The largest *double* number that can be stored is 1.7 times 10 to the power of 308. The precision with which real values can be stored is increased from 7 digits for *real* to 15 digits for *double*.
- extended*: *extended* extends the range and precision of type *double*. The precision is now 19 digits and the largest value is 1.1 times 10 to the power of 4932. Use this one to compute the national debt.
- string*: *string* is a type which allows from 0 to 255 characters to be associated with a single identifier, so you can store a complete name or phrase. **String** is the only type identifier which is also a reserved word, which is why it's printed in boldface. As you'll see in chapter 11, type **string** has many properties of a structured data type, like an array. But until then, we'll just think of it as another simple type.

ordinal types
(10)

The ordinal types in Macintosh Pascal are *integer*, *longint*, *char*, and *boolean*.

String Input
and Output

Of all of the non Standard data types included in Macintosh Pascal, **string** is by far the most important. Today's computers work with text at least as much as they work with numbers, and a string data type is an important feature of a text-processing language. The string-handling capabilities built into Macintosh Pascal can make short work of programming tasks that are at best tedious with Standard Pascal.*

String variables can be read from the keyboard and written to the text window much like the other variable types you've seen:

```
readln (string_variable);
writeln (string_variable);
```

String output is straightforward enough; all of the characters in the string are written just as if they had been stored and written as individual characters. But string input is complicated by the fact that a **string** variable may be any length up to 255 characters. How does the computer know where the string stops? The answer is simple: it keeps reading until it hits the end of the line (a **Return**) and counts everything up to that point as part of the string. The next *read* or *readln* starts reading on the next line, so a second string could be read from the second input line with a second *readln*.

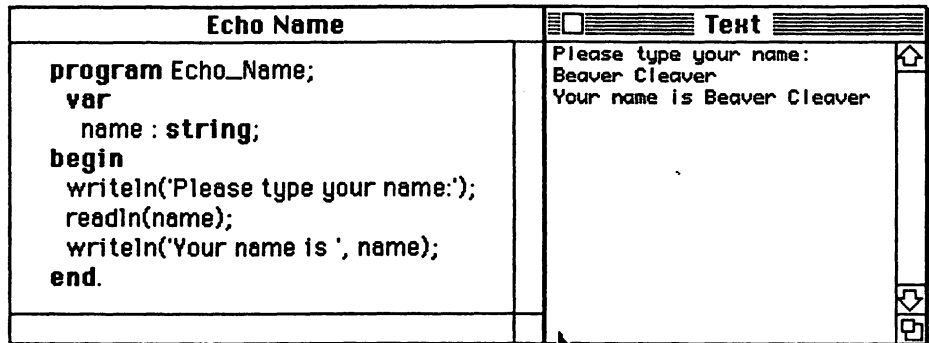
* In fact, strings have been incorporated into most microcomputer versions of Pascal.

1 Getting Acquainted with Macintosh Pascal

(The *read* statement also reads all of the characters until the end of the line and places those characters in the string variable. But it's not usually a good idea to read a string with a *read*, because it doesn't automatically skip over the **Return** at the end of the line.)

Because *readln* with a **string** variable as a parameter puts all remaining characters on the line into that variable, you can't read two strings from the same line, and you therefore shouldn't try to include two strings in one *readln*. It's perfectly OK to include variables of other types in a *readln* with a **string** variable, but the string should be the last variable in the list. (Why?)

Here's a simple program that illustrates basic string input and output:*



Antibugging and Debugging (22)

The Macintosh Pascal interpreter translates each line and checks its syntax as soon as you finish typing it. If it can't understand what you have typed, it **outlines** the text, starting with the point at which the program stops making sense. Suppose you're typing in your first program and you spell *begin* incorrectly. The part of the program that follows is displayed in **outline** print.

translation bugs (22)

```
program First_Run;  
{This is our first program.}  
begn writeln ('Hello. I love you.')
```

The bug is easy to spot because *begin* is not written in **boldface** as it would be if the interpreter recognized it as a reserved word. To correct the bug, just insert an "i" with the editor and move on; the outline type should disappear and "begin" will be written in **boldface**. (The details of this kind of editing are explained in the first hands-on session.)

```
program First_Run;  
{This is our first program.}  
begin  
  writeln ('Hello. I love you.')
```

* We've changed the window sizes here; you'll see how in the second hands-on session.

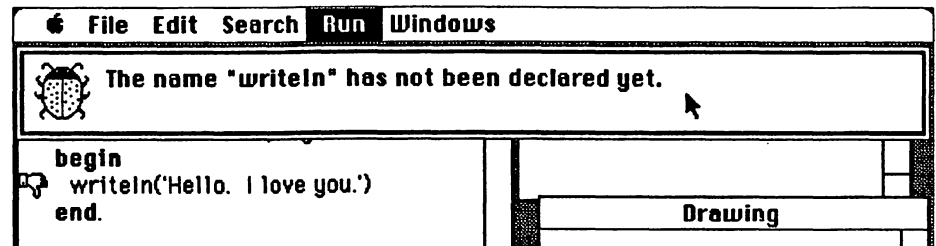
spurious bugs
(23)



Some syntax errors are not immediately recognized by the interpreter. Suppose that we type the capital letter "T" instead of the lowercase letter "t" in *writeln*. The interpreter doesn't immediately detect this as an error. If we choose the Check option in the Run menu, the interpreter checks through the entire program for errors and finds the incorrect spelling of *writeln*. Two things happen when a bug like this is detected:

1. The line in which the error occurs is marked with a thumbs down.
2. A ladybug appears in a dialog box along with an error message to give you an idea of what went wrong.

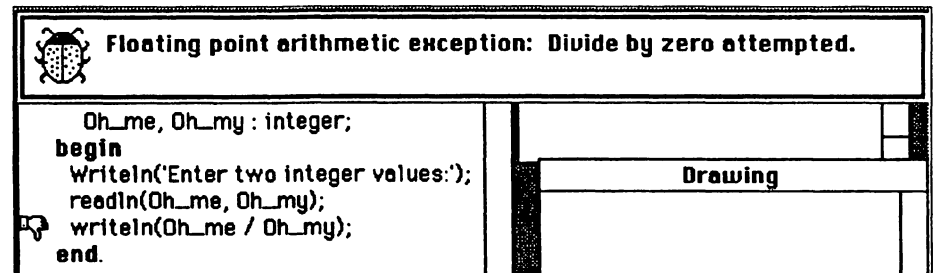
Before you can fix this kind of error, you'll have to acknowledge and clear the error message by clicking somewhere in the error box or hitting the **Return** key.



Be careful about taking error messages too literally. Many error messages are hard to understand; others can be downright misleading. They don't always describe the error accurately, and they sometimes point you to the wrong line of the program. Interpret them as important clues rather than absolute truths.

run-time bugs
(24)

Some bugs won't be detected by the interpreter until the program is run. In this example a value of zero was read for the variable *Oh_my*, causing this error message:



order of input
(25)

In many cases, Macintosh Pascal is more forgiving than other Pascals. For example, an input type clash won't terminate the program; the machine just beeps and waits for a correctly-typed value.

Common error messages are listed and explained in the reference section of this book.

1 Getting Acquainted with Macintosh Pascal

system bugs



More serious errors may cause a bomb, rather than a bug, to appear, along with an error message that may be so vague that it's useless. Sometimes a bomb appears because you've made a programming error, but more often it means that you've uncovered a bug in the Mac Pascal interpreter or the Macintosh operating system. Ominous visual image notwithstanding, a bomb does not mean that your machine is damaged or in danger; it just means that the Macintosh operating system is terribly confused by what just happened. You may never see a bomb; they're fairly rare, especially in mature software. But when a bomb *does* appear, it may leave you with no alternative but to restart the machine. That can be a minor inconvenience or a major setback, depending on how long it's been since you saved the program you were working on.

get a listing
(24)

A good way to find bugs in a longer program is to make a printed listing so you can examine it in total and make written notes and corrections. To list the entire program, select the Print option from the File menu, as described in the hands-on tutorial.

write test
programs
(26)

In addition to writing and testing short programs to see how things work (or don't work), you can take advantage of Macintosh Pascal's Instant window and Observe window to test features of the language and debug your programs. The next two chapters will show you how to use these powerful tools.

Still More
Exercises

1-32 Complete the first hands-on session in the Appendix.

1-33 Type in, edit, and run the example programs from Chapter 1 of *Oh! Pascal!*

1-34 Write a program that reads a value into an integer variable. Have your program write a prompt to the text window before the read statement. Run the program and type a letter instead of a number. What happens?

1-35 Different implementations of Pascal use different default output formats. Write a program that determines the default format for real and integer variables in Mac Pascal. Change the program so that it writes the result of the expression $10/8$ so that all of the significant places are written.

Programming Calculations and Graphics

IN CHAPTER 1 of *Oh! Pascal!* you were introduced to *write* and *writeln* two standard *procedures* that allow you to turn program values into text output. In Chapter 2, you met several standard *functions* that calculate and return simple values to your programs. In this chapter, you'll meet several Macintosh Pascal procedures and functions that can be used to produce pictures, rather than numbers and words. Graphics is an exciting application of computer science that's completely ignored in Standard Pascal; we'll compensate by giving it lots of attention here. But first, we'll discuss some simple extensions to those facets of Pascal discussed in Chapter 2 of *Oh! Pascal!* (When you've finished this chapter, don't forget the second hands-on session, *Tools and Tricks*, in the Appendix.)

Assignments and Expressions (34)

initializing variables (35)

Let's start by discussing a potentially hazardous language extension: Mac Pascal's automatic initialization of variables. Each time a program is run, Macintosh Pascal initializes all numeric variables to zero, all characters to blanks, all **string** variables to empty (zero characters), and all *boolean* variables to false. That means you don't need to explicitly initialize those variables yourself in the program – *but you should anyway!* Programs that don't explicitly initialize their own variables can cause subtle and terrible bugs if they're ever run on other systems, and programmers who don't learn good initialization habits can wreak havoc when unleashed in a non-initializing system. We therefore strongly recommend that you ignore this automatic initialization feature and *always initialize your own variables*.

String variables can be assigned values just like other variables. Like *char* variables, strings are assigned values enclosed in single quotes:

string assignments (36)

```
String_Var_1 := 'B';
```

But unlike character variables, strings aren't limited to one character's worth of data, so it's OK to say

```
String_Var_2 := 'What's up, Doc?';
```

Of course, it's perfectly OK to assign one string to another, or to assign a *char* to a **string** variable, or even to assign a string to a *char*, provided that the string contains only one character:

```
Char_Var := 'Y';
String_Var_3 := String_Var_2;
String_Var_2 := Char_Var;
Char_Var := String_Var_1;
```

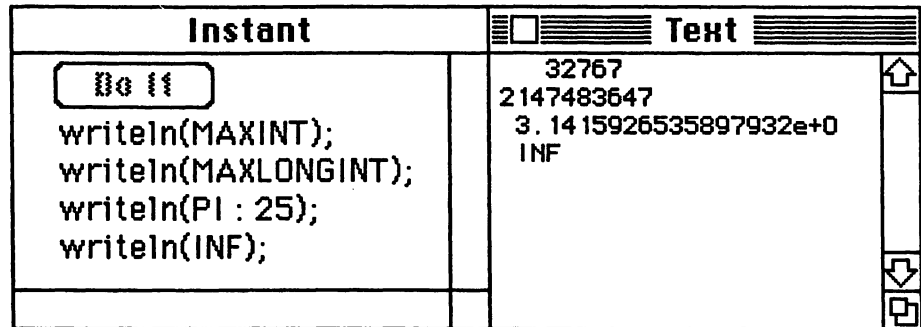
2 Programming Calculations and Graphics

MAXINT
and other
constants
(45)

Let's look at Macintosh Pascal's built-in numeric constants. The Standard Pascal constant *MAXINT* has the value of 32,767 in Mac Pascal; that's the largest legal *integer* value. A non Standard constant, *MAXLONGINT*, represents the largest possible value of a variable of *longint* type: 2,147,483,647. *PI* is another built-in constant that provides an extremely accurate value for that old favorite from geometry, π . *INF* can be used to represent infinitely large real values in arithmetic expressions; when it's printed out by itself, it's just "INF."

Suppose in the middle of writing a program you need to check the values of those four constants. If you don't have a reference manual handy, you can ask Mac Pascal via one of its friendliest tools: the Instant window.

the
Instant
window

This window allows you to execute one or many Pascal statements without having to write an entire program. Once you've selected Instant from the Windows menu, all you need to do is type in the statements that you want to try and click the "Do it" button in the window. (It's a good idea to save your program before you Do it, just in case your Instant code bombs the system.) The Instant window is especially handy for experimenting with small chunks of Pascal code to test them without writing a complete program. (Since the standard editing features are available inside that window, you can Copy statements that you've tested there directly into your program window.)

Built-In
Functions and
Procedures
(51)

In addition to the Standard Pascal functions, Macintosh Pascal includes a particularly useful arithmetic function called *random*. *Random* returns a random integer in the range -32768 to 32767.* It has no arguments. Examples:

```
writeln(Random:10);
```

or

```
writeln(Random mod 10.0: 3) {for a random digit 0-9}
```

Besides *random*, Mac Pascal includes several functions for working with the added predefined types (*longint*, *extended*, *double*, and *string*) and a variety of other special-purpose functions. In addition, Macintosh Pascal

* Random numbers are discussed in *Oh! Pascal!*, pages 144-6

includes predefined *procedures* for working with strings, files, sound, graphics, and the Macintosh user interface. Unlike the built-in functions, these procedures do more than return simple values. Each predefined procedure is a complete Pascal statement that tells the interpreter to do something. You've already worked with *readln*, *read*, *writeln*, and *write*, four procedures built into Standard Pascal. Standard Pascal has a few more; Macintosh Pascal has dozens.

Two simple procedures, *ShowText* and *ShowDrawing*, allow you to open the Text window and the Drawing window, respectively, from inside any program. It's important to use those procedures at the beginning of any program that depends on the visibility of those windows so the program will work properly even if they're closed or covered when it's run.

Introducing QuickDraw Graphics



When Wirth designed Pascal, almost all computer output was in the form of text or numbers. By contrast, output from the Macintosh is as likely to be pictures as words or numbers. Programming the Macintosh means programming with graphics, and Macintosh Pascal includes a QuickDraw library of procedures and functions to make graphics programming easier.

The entire Macintosh screen is composed of *pixels* (picture elements) – tiny dots that can be displayed as either black or white. Pixels can be combined in patterns that look like icons, words, numbers, space invaders, or just about anything else. (The same principle is used by card-carrying football cheering sections to make stadium-sized pictures and messages.)

In Mac Pascal, graphics output is shown in the drawing window. Building graphics means making pixel patterns. A single black pixel on a white pixel background makes a point, a row of black pixels on a white background looks like a line, four lines can make a rectangle, and so on. To allow you to position figures in the window, each pixel has an address that indicates its vertical and a horizontal position in the window. These positions are represented as integer values. The upper left hand corner of the window has vertical and horizontal positions of zero (0,0). (If you're comfortable with Cartesian coordinates, you'll have to readjust your orientation so that bigger means down, not up.) The horizontal positions increase as you move to the right to a maximum of 495 pixels, and the vertical positions increase as you move down the window to a maximum of 295 pixels. These maximum values assume that you've stretched the drawing window to fill the entire screen.*

QuickDraw Procedures

Creating graphics would be tedious, indeed, if we had to individually position each pixel on the screen. Fortunately, Macintosh Pascal has a library of built-in QuickDraw graphics procedures which can be used in programs to produce pictures. Most of these QuickDraw procedures have arguments that specify pixel addresses. There are more than 100 QuickDraw graphics procedures, but only a few are necessary for the drawings you'll be doing here.

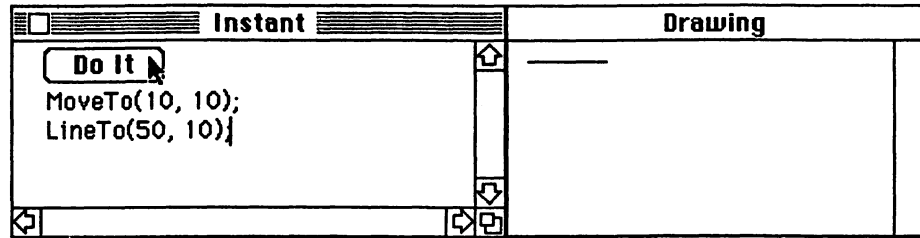
To understand how QuickDraw procedures work, imagine that you have a magic pen that draws on the screen wherever you tell it to. Drawing

* The hands-on session will show you how to manually change the window size. In Chapter 12 you'll see how you can make it even larger by using a procedure to resize it.

Drawing Lines

a line is a simple two-step process: move the pen to the starting point of the line and draw to the ending point. Suppose we want to draw a line 40 pixels long and 10 pixels from the top of the window. We'll use two QuickDraw procedures: *MoveTo*, which positions the pen at the starting point for the line (10 over and 10 down from the upper left corner), and *LineTo*, which moves the pen to the end (50 over and 10 down), drawing as it moves. The Instant window is ideal for short graphics tests like this:

procedures
MoveTo and
LineTo



Notice that each of these procedures has two arguments: a horizontal coordinate and a vertical coordinate. These parameters are like the ordered pairs used in analytic geometry, in that the horizontal coordinate *must* be listed first. (But remember that the Y coordinate measures *down*.) The two arguments must be of type integer, because points in the graphics window are represented by integer coordinates. They may be integer values, integer expressions, or integer variables, as long as they're in the right order.

The line is solid black and one-pixel wide, as if it were drawn with a black fine-tipped pen. As it turns out, you can change the width and pattern (color) of your pen with these procedures:

QuickDraw
pen-changing
procedures

***PenSize*(width,height):** Adjusts pen thickness. Default is (1,1).

***PenPat*(pattern):** Selects the pattern for the pen. The choices are black (default), white, gray, ltgray, dkgray.

***PenNormal*:** Sets pen to default size and pattern.

***HidePen*:** and ***Showpen*:** These turn the pen off and on respectively and are very useful for drawing dashed lines.

MoveTo and *LineTo* send the pen to absolute coordinate positions. QuickDraw has two other pen-moving procedures that allow you to move the pen a specified number of pixels from its current location. Here's a summary of the pen-moving procedures:

QuickDraw
pen-moving
procedures

***MoveTo*(horizontal, vertical):** Moves pen (without drawing) to the specified position. Move absolute.

***LineTo*(horizontal, vertical):** Moves pen (drawing all the way) to the specified position.

***Move*(horizontal,vertical):** Moves pen (without drawing) to the point *horizontal* pixels right or left and *vertical* pixels up or down. Move relative.

***Line*(horizontal, vertical):** Moves pen (drawing all the way) to the point *horizontal* pixels right or left and *vertical* pixels up or down.

Drawing Shapes

Using *MoveTo* and *LineTo*, we could draw a rectangle like this:

```
MoveTo(30,40);
LineTo(30,100);
LineTo(150,100);
LineTo(150,40);
LineTo(30,40);
```

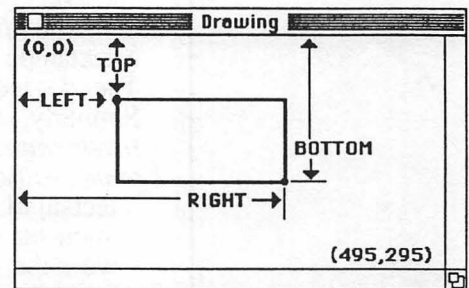
but, as it turns out, we can do the same thing with a single procedure call:

```
FrameRect(40,30,100,150);
```

The four arguments of *FrameRect* represent, in order:

1. the distance, in pixels, from the top of the window to the top of the rectangle (40)
2. the distance from the left of the window to the left side of the rectangle (30)
3. the distance from the top of the window to the bottom of the rectangle (100)
4. the distance from the left of the window to the left side of the rectangle (150).

procedure
FrameRect



(Cartesian thinkers might prefer to think of the first two arguments as an ordered pair representing the upper left corner of the rectangle and the last two as the lower-right corner.)

We could have also written

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

assuming that *top*, *left*, *bottom*, and *right* had already been assigned values of 40, 30, 100, and 150, respectively. The names of the arguments aren't important to Pascal; we could just as easily have written

```
FrameRect(john, paul, george, ringo);
```

or even

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

and gotten the same results, as long as the four arguments in parentheses had been assigned values of 40, 30, 100, and 150, when reading from left to right. In other words, Pascal pays attention to the order, rather than the names, of the arguments, in procedures. In *FrameRect*, the first argument *always* represents the top of the rectangle, the second the left side, and so on. To preserve your sanity, use argument names that correspond to the actual meanings.

2 Programming Calculations and Graphics

Here are the most important QuickDraw shape-drawing procedures:

QuickDraw
shape
procedures

FrameRect(top,left,bottom,right): Draws a rectangle with the specified bounds.

PaintRect(top,left,bottom,right): Draws a solid black rectangle (unless you specify another fill pattern using the *PenPat* procedure).

FillRect(top,left,bottom,right,pattern): Fills a rectangle with the specified pattern. *FillRect* allows the fill pattern to be different from the border pattern; *PaintRect* does not. The *pattern* argument may be any *black*, *dkgray*, *gray*, *lgray*, or *white*.

EraseRect(top,left,bottom,right): Erases (turns white) everything inside the specified rectangular area.

InvertRect(top,left,bottom,right): Changes every black pixel to white and every white pixel to black in the specified area.

FrameRoundRect(top,left,bottom,right,oval_width,oval_height): Draws a rectangle with the specified boundaries and with rounded corners. The degree of rounding is specified by the last two arguments. Similarly, *PaintRoundRect*, *EraseRoundRect*, *FillRoundRect*, and *InvertRoundRect* work like their square-cornered counterparts.

FrameOval(top,left,bottom,right): Draws an oval in the specified rectangular area. *FrameOval*(10,10,110,110) draws a circle with a diameter of 100. *PaintOval*, *FillOval*, *EraseOval*, and *InvertOval* work the same way.

FrameArc(top,left,bottom,right,start_angle,arc_angle): Draws an arc within the specified rectangle, of a size specified by *arc_angle*, beginning at *start_angle*. The angles are specified in Integer degrees, and zero degrees is twelve o'clock. *PaintArc*, *FillArc*, *EraseArc*, and *InvertArc* also exist.

Labeling
Graphics

One picture may be worth a thousand words, but it's often worth even more if it's got a few well-chosen words of its own. It's easy to add text to your graphics by writing text output directly to the Drawing window. The procedure *WriteDraw* works just like *write* except that it writes to the Drawing window instead of the Text window. The lower left corner of the first character of the text begins wherever the pen rests. (If the pen isn't where you want it, use *MoveTo* to get there.)

procedure
WriteDraw

```
WriteDraw('FERTILITY RATE, ', year1:4, 'to ', year2:4);
```

The problem with *WriteDraw* is that it only allows you to specify the position of the lower left corner of text, so it's difficult to center labels on graphs and diagrams. Fortunately, there's another built-in procedure, *DrawString*, that makes it easier to center labels.

procedure
DrawString

```
DrawString(label);
```

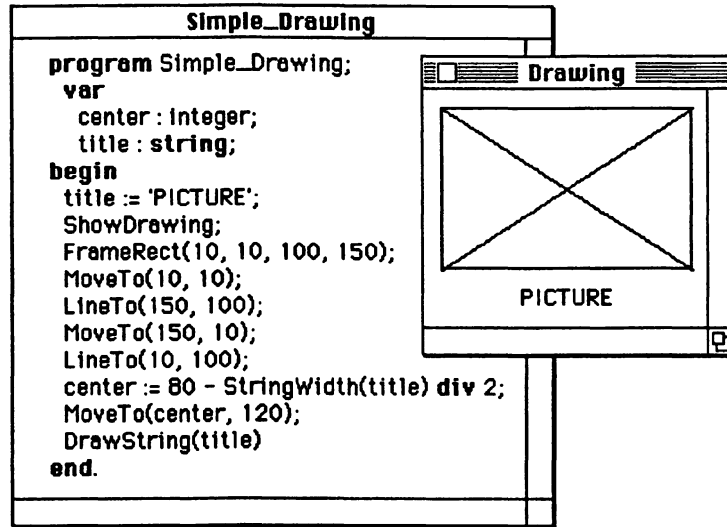
The argument *label* is of type **string** and may be any **string** value, variable or expression that you want to display in the Drawing window.

Like *WriteDraw*, *DrawString* starts the text at the current pen position. But since *DrawString* is writing out exactly one string, you can center the label if you can determine the width of that string in pixels. The function *StringWidth* returns an integer value equal to the width of the label in pixels:

procedure
StringWidth

```
Label_Size:= StringWidth(label);
```

This short example shows how *StringWidth* and *DrawString* can be used to center a picture. (The Drawing window, like the Text window, is cleared whenever Go or Reset is selected from the Run menu.)



Fine, but suppose we want to include a numeric variable in our centered label? In order to use *StringWidth* for centering we need to limit ourselves to a label that can be expressed as a single string argument. Fortunately, there's a built-in function called *StringOf* that works just like write, except that it "writes" the indicated values into a string:

function
StringOf

```
String_Var := StringOf('FERTILITY RATE,', year1:4, 'to ', year2:4);
```

If *year1* is an integer variable with a value of 1950 and *year2* has a value of 1990, *String_Var* now has a value of 'FERTILITY RATE, 1950 to 1990'. Since it's a string, *String_Var* can be used as an argument for *StringWidth* and for *DrawString*.*

* For the record, procedure *ReadString* goes in the other direction, allowing you to break a string into a collection of numeric, character, or other values. It works just like *read*, except that its first argument is a string from which the remaining arguments can be read.

2 Programming Calculations and Graphics

Here's a summary of the most important functions and procedures for working with text in the Drawing window:

QuickDraw
text
procedures
and functions

WriteDraw(expression_1, expression_2, ..., expression_n): Writes the specified expressions to the drawing window. The pen begins at the lower left of the first character.

DrawString(string): Writes the specified string of characters in the drawing window at the current pen position. The pen begins at the lower left of the first character.

DrawChar(character): Writes a single character to the drawing window in the same manner as *DrawString*.

StringWidth(string) and **CharWidth(character)** are functions that return integer values giving the width in the number of pixels.

StringOf(expression_1, expression_2, ..., expression_n): Not actually a part of the QuickDraw library, but often used to convert a series of expressions into a string so that they can be written with *DrawString*.

TextFace(face): Selects the style of the characters. *face* must be [bold], [italic], [underline], [outline], [shadow], [extend], or [condense]. The square brackets must be included: *TextFace([shadow])*. To indicate two or more styles, include both in brackets separated by commas: *TextFace([bold, italic])*. Indicate plain text with empty brackets [].

TextSize(size): Selects the size of the characters. *size* is an integer value and is the *point size* of the characters, which is roughly their height in pixels. Characters look best when they're written in sizes that are stored in the system file; otherwise they're likely to have jagged edges resulting from scaling up or down. (9 and 12 are usually safe; 12 is the default.)

TextFont(font): Selects the character font (typestyle) from among those available in your system file. *font* must be an integer value. If you want to use this one, you'll need to experiment or ask for help to determine the numbers of the system fonts on your disk.

Let's finish up with a short program that illustrates several of the QuickDraw procedures and functions we've discussed. This program isn't as easy to read as the non-graphic programs we've seen so far, mostly because of all the numeric arguments in the QuickDraw procedures. If you find it confusing, try stepping through the program listing one statement at a time using a piece of graph paper to represent the Drawing window. If that seems too tedious, the second hands-on session the Appendix will show you how Macintosh Pascal can step through the program for you so you can actually watch the drawing develop one statement at a time.

QuickDraw
graphics
demo
program

```

program Draw_a_Package; {to demonstrate QuickDraw routines.}
const
  CENTER = 107; {Horizontal center of box.}
var
  Labels : string;
begin
  ShowDrawing;
  PenSize(4, 4);
  FillOval(5, 5, 56, 210, dkgray);           {Shadow box top.}
  FrameOval(5, 5, 56, 210);                   {Outline box top.}
  FrameArc(210, 5, 261, 210, 90, 180);        {Draw box bottom.}
  MoveTo(5, 32);                              {Draw sides of box.}
  Line(0, 200);
  Move(201, 0);
  LineTo(206, 32);
  PaintRoundRect(70, 12, 120, 203, 20, 20); {Draw brand patch.}
  EraseOval(75, 22, 115, 193); {Clear place for brand name in patch.}
  TextSize(18);                               {Set up and draw brand name.}
  TextFace([shadow]);
  Labels := 'MAC PASCAL';
  MoveTo(CENTER - StringWidth(Labels) div 2, 101);
  DrawString(Labels);
  TextFace([]);                               {Clear TextFace.}
  TextFace([bold, italic]); {Draw product description.}
  Labels := 'FORTIFIED';
  MoveTo(CENTER - StringWidth(Labels) div 2, 140);
  DrawString(Labels);
  TextSize(10);
  Labels := 'Builds Better';
  MoveTo(CENTER - StringWidth(Labels) div 2, 155);
  DrawString(Labels);
  Labels := 'Programs';
  MoveTo(CENTER - StringWidth(Labels) div 2, 167);
  DrawString(Labels);
end. {Draw_a_Package}

```



2 Programming Calculations and Graphics

Still More Exercises

- 2-31 Complete the second hands-on session, *Tools and Tricks*, in the Appendix.
- 2-32 Sales of Fortified Mac Pascal are slipping. Redesign the packaging to rekindle consumer interest.
- 2-33 Write a program to draw a square inscribed in a circle inscribed in an equilateral triangle. Have your program accept as input the diameter of the circle and calculate the dimensions of the other two figures.
- 2-34 Write a program that takes the length of the two perpendicular sides of a right triangle as input, draws the triangle, and labels the length of each of the three sides.
- 2-35 Pie charts are used frequently to pictorially represent percentages. These diagrams are circles divided sliced into pieces. The size (arc) of each piece is proportional to the percentage represented by that piece. Write a program that reads three values which represent the number of registered voters who are Republican, Democrat, and Undecided. Your program should then determine the percent for each party and draw a pie chart to illustrate those percentages. Add labels to each piece of the pie.
- 2-36 Write a program which reads digital data representing the time of day in hours, minutes, and seconds and converts the time to analog data in the form of a clock face with hands. The clock face should have numbers at 3, 6, 9 and 12 o'clock.
- 2-37 Do exercise 2-24, representing graphically the time on Ziggy's 24-hour clock and on his new Mercurichrone
- 2-38 Write a program that draws a simple house. Label your house with an address centered beneath it.
- 2-39 Repeat your solution to 2-26 with the output going to the Drawing Window instead of the text window.
- 2-40 Write a program to draw a picture of a Macintosh computer. (Can this be considered a self portrait? Is it foreshadowing for recursion?)

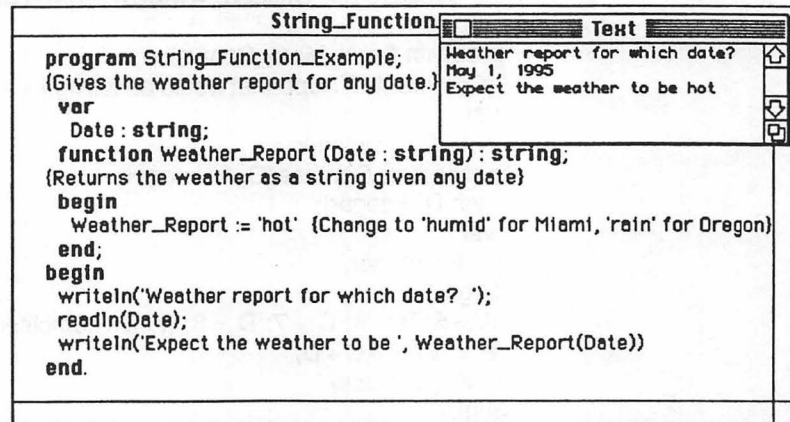
Procedures and Functions for Problem Solving

THIS IS ONE OF THE MOST IMPORTANT – and demanding – chapters in *Oh! Pascal!* Fortunately, Macintosh Pascal conforms closely to Standard Pascal in its treatment of procedures and functions, so you won't be saddled with much additional material here. The one exception – functions of type **string** – is straightforward and intuitive. If you found procedures, parameters, and functions generally confusing, Mac Pascal's Observe window, introduced in this chapter, will come to the rescue. This innovative and powerful tool, when used with the examples and exercises in *Oh! Pascal!*, will help make subprograms and parameters crystal clear.

Functions As Subprograms (82)

As you know, Standard Pascal functions may be of result type *char*, *boolean*, *integer*, and *real*. Macintosh Pascal allows functions to be of type **string**, too. Here's an admittedly silly example. Later, when you've been introduced to Mac Pascal's advanced string-handling capabilities, you'll see how useful string functions can really be.

function type (82)



Antibugging and Debugging (90)

Using the Observe Window




As we saw in *Oh! Pascal!*, bugs associated with parameters often result from confusing value and variable parameters, and programs with lots of procedures and parameters can be tedious to debug by hand. Fortunately, Mac Pascal's Observe window allows you to observe the value of any variable or expression at any point in the execution of a program. It's an important tool for debugging – or just understanding – longer programs.

The general procedure for using this tool is to select Observe from the Window menu and then type in the expressions you wish to observe. The usual editing features are available, so you don't need to retype long names; you can just Copy them from your program listing and Paste them in. The values of those variables will be continuously displayed in that window as your program executes.

3 Procedures and Functions for Problem Solving

If you select Go, the program will probably run too fast for you to observe the changes in your selected variables. There are three ways to slow or stop the execution so you can check the values of your variables.

- | | |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Step | 1. Instead of selecting Go, select Step (or  -S). You probably used this technique to step through a graphics program one statement at a time in the last hands-on session; it works particularly well with the Observe window for tracking variable changes. But for longer programs, single-stepping can be tedious. |
| Step-Step | 2. To speed things up, select Step-Step from the Run menu. This auto-stepping command won't wait for you, so you have to watch carefully. |
| Stops In | 3. For many problems it's helpful to stop the program at key checkpoints to observe the values of your selected variables. You can do this by selecting Stops In from the Run menu. Then each time the mouse is clicked to the left of a statement a stop sign is inserted. Now when Go is selected from the Run menu the program will execute only to the next stop sign. Program execution is continued only when Go or Go-Go is selected from the Run menu. Go-Go behaves like Step-Step except that it only pauses at stop signs. |
| Go-Go | |

observing
parameters
and scope

The Observe window can also be used to examine the nuances of local and global variables in subprograms. Let's observe this variation on *Confusion* from *Oh! Pascal!*, page 81. (We've removed most of the *writeln* statements because the Observe window serves the same function here.)

```
program Even_More_Confusion;
{Comments? Nope-that would be telling.}
var
  A, B, C, D : integer;
procedure Confuse (C, A : integer;
  var D : integer);
var
  B, E : integer;
begin
  A := 5; B := 6; C := 7; D := 8; {Lines combined to save space}
  E := A + B + C + D;
end; {Confuse}
begin
  A := 1; B := 2; C := 3; D := 4;
  Confuse(B, A, D);
  writeln(A, B, C, D)
end. {Confusion}
```

You'll see in the windows on the following pages that we've set stops at the beginning and end of the procedure, at the procedure call, and at the final *writeln*, and we've typed the variable names into the Observe window. When we select Step-Step, the program plods along until it stops at the first stop sign, waiting for you to give it the go-ahead so it can execute the fingered statement. All of the variables have values displayed except E, which is local to the procedure and therefore undefined at this point.

using stops

Even_More_Confusion		
<pre> A := 1; B := 2; C := 3; D := 4; Confuse(B, A, D); writeln(A, B, C, D) end. {Confusion} </pre>	Observe	
	1	A
	2	B
	3	C
	4	D
		Undefined name E

We can select Step-Step again to watch the changes in the variables as the program continues, or we can select Go to move quickly to the next stop sign. Either way, the finger moves into the procedure and stops at the **begin**. Inside the procedure all five observed variables are defined, including the local variable E. But B and E have not been assigned values by the program, so they show values of 0, because Macintosh Pascal initializes all variables automatically. D is a variable parameter; A and C are value parameters. All three of these variables have values assigned as a result of being passed as parameters. Note also that the values for A, B, and C are local to this procedure and will change when the procedure ends.

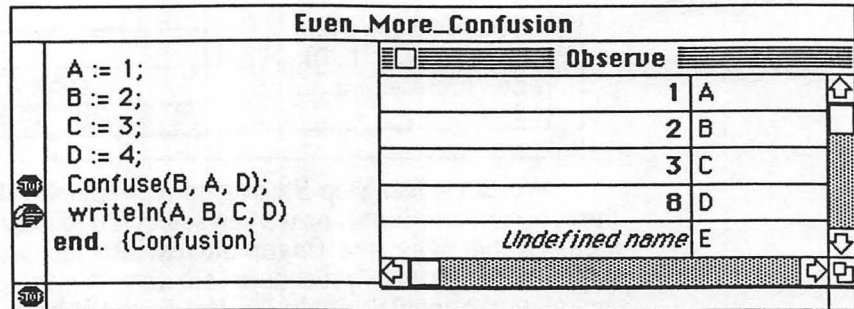
Even_More_Confusion		
<pre> var D : integer; begin A := 5; B := 6; C := 7; </pre>	Observe	
	1	A
	0	B
	2	C
	4	D
	0	E

The finger has reached the end of the procedure and values have been assigned to each variable within the procedure. A, B, C, and E are local variables and will be undefined when the end statement is executed. When execution returns to the main program A, B, and C will refer to the variables local to the main program.

Even_More_Confusion		
<pre> C := 7; D := 8; E := A + B + C + D; end; {Confuse} begin A := 1; B := 2; </pre>	Observe	
	5	A
	6	B
	7	C
	8	D
	26	E

3 Procedures and Functions for Problem Solving

Back in the main program, E is no longer defined. The identifiers for A, B, and C now refer to the versions declared in the main program and contain the values assigned to them in the main program. D was passed as a variable parameter and its contents were affected by what went on in the procedure.



Once you're done observing variables, you'll probably want to remove the stop signs from the program. A single stop sign may be removed with a click on the sign. To remove the whole set, select Stops Out, which has replaced Stops In in the Run menu.

Observing the values of variables during the course of program execution is one of the most effective ways of finding program errors. All programmers do this kind of observation – often by inserting temporary *writeln* statements. The Observe window is one of the most important and powerful tools in your Macintosh Pascal toolbox; we encourage you to use it often.

Still More Exercises

- 3-32 Use the Observe window to follow the values of the variables and parameters in *HardToBelieve* (3-13) as the program runs.
- 3-33 Answer exercise 3-14 for Macintosh Pascal.
- 3-34 Write the program to help Monica Marin (3-23) using QuickDraw graphics.
- 3-35 Write procedures to produce block letters in the Drawing window using QuickDraw graphics. Write a program to display your name with these letters.
- 3-36 Write procedures to produce a character font of your own design.
- 3-37 Write procedures to produce line graphs. These might include. *Draw_Axes*, *Label_Axes*, and *Draw_Curve*. The input is to be from the keyboard and will include titles for labeling the axes, the range of values along the *x* and *y* axes, and the *x* and *y* values to be plotted by the line. Do the input values have to be ordered and, if so, how?
- 3-38 Write a procedure that draws a simple house. The procedure should have parameters for receiving values for the position of the house and the size of the house. Then use the procedure in a program that draws a town with houses of different sizes.
- 3-39 Write a procedure that draws a flower. Use that procedure in a program that draws a flower garden.
- 3-40 Program your solution to Hangman (3-25) using QuickDraw graphics.

Taking Control of Execution: the **for** Statement

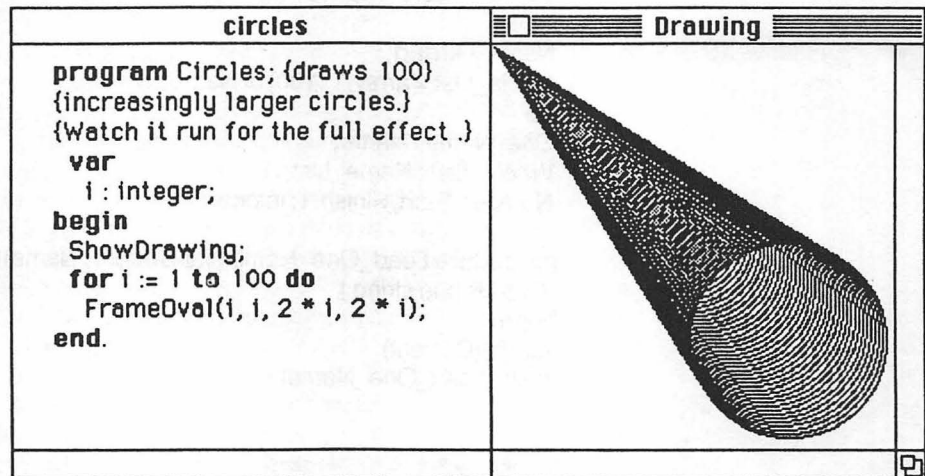
THE **FOR STATEMENT** in Macintosh Pascal is identical to the Standard Pascal **for** statement. In this chapter we'll do a quick graphics **for**-loop trick and then show by example how strings can be combined using arrays in Macintosh Pascal. The section on arrays is optional; read it only if you're reading the corresponding optional section in *Oh! Pascal!*

Don't be confused by the section on strings in this chapter of *Oh! Pascal!*. You've already seen the Mac Pascal **string** type in action. But in the language of Standard Pascal, a *string* is just a packed array of characters. Packed character arrays are available in Mac Pascal, but since they're not as convenient or powerful as **strings**, we won't use them, and you probably won't, either. Consequently, that section may be unnecessary unless you're interested in understanding Standard Pascal, warts and all. In the interest of completeness, we've rewritten the section on arrays of strings (pages 1123-125) using Mac Pascal **string** variables; the other material in that section is old news to you.

The **for** statement allows you to create elaborate graphics in Mac Pascal by repeating basic graphic commands. This example shows how a simple loop can create an impressive visual effect:

for
Statements
and
Program
Actions
(100)

for statements
and graphics



Try creating your own graphics programs using QuickDraw procedures to get a good sense of how this iterative control statement works in Pascal.

4 Taking Control of Execution: the **for** Statement

One- Dimensional Arrays* (112)

As you've seen, **for** statements are ideal for working with arrays, too. Arrays of strings are common in Macintosh Pascal. Here's how you might declare an array variable to contain up to 100 names:

arrays of strings
(123)

```
type
    Name = string;
    Name_List = array [1..100] of Name;
var
    One_Name: Name;
    Whole_List: Name_List;
```

Using arrays of **strings** is surprisingly easy. Let's develop a program that uses the array techniques we've learned so far:

Write a program that reads in up to 100 names. Let the user ask for a printout of a sublist of names (e.g., the twentieth through fifty-ninth).

A pseudocode restatement of the problem gives us:

```
find out how many names there will be;
read the names
find out which names should be output;
for the correct starting through finishing name
print the name;
```

Here's a Mac Pascal solution:

```
program Store_Names; {Maintains an array of strings.}
type
    Name = string;
    Name_List = array[1..100] of Name;
var
    One_Name : Name;
    Whole_List : Name_List;
    Number, Start, Finish, i : integer;

procedure Load_One_Name (var Current : Name);
    {Reads one string.}
begin
    readln(Current)
end; {Load_One_Name}
```

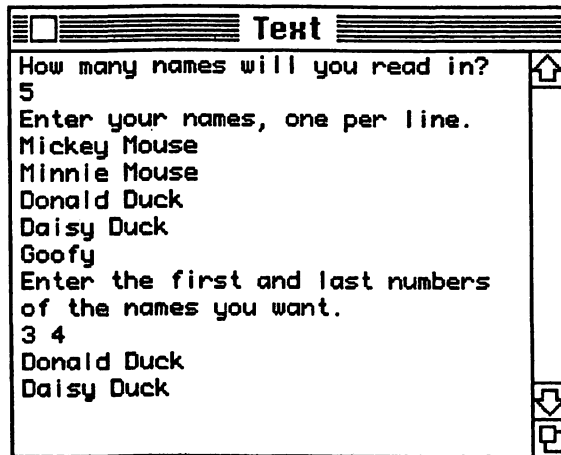
{continued}

* This section is optional.

```

begin
  ShowText;
  writeln('How many names will you read in?');
  readln(Number);
  writeln('Enter your names, one per line. ');
  for i := 1 to Number do
  begin
    Load_One_Name(One_Name);
    Whole_List[i] := One_Name;
  end; {We've read in the list of names.}
  writeln('Enter the first and last numbers of the names you want. ');
  readln(Start, Finish);
  for i := Start to Finish do
    writeln(Whole_List[i]);
  end. {Store_Names}

```



Of course, it's kind of silly to isolate the *readln* in a separate procedure; we just did it to show you how easy it is to pass an individual element of this **string** array as a parameter. As you might expect, an array element whose value is modified (as in procedure *Load_One_Name*) must be passed to a variable parameter.

Still More Exercises

- 4-26 Write the graphics part of 4-20 using QuickDraw graphics.
- 4-27 Write a program to draw a checkerboard with the pieces.
- 4-28 With the **for** loop you can easily draw tick marks on the axes of a graph. Write a procedure that draws the axes for a graph and places major ticks (long) every tenth of the way along each axis and minor ticks (short) each twentieth of the way along the axis.
- 4-29 As you saw in program *Circles*, you can use **for** loops to create very interesting and attractive graphics by repeatedly drawing the same shape at different positions and/or sizes. Write programs to produce such graphics by drawing lines at various angles, drawing ovals at various positions, and drawing circles whose centers form a circle.

Making Choices: the **case** Statement

THE MAC PASCAL **case** structure has just one extension – the **otherwise** clause – which we'll deal with immediately. We'll also say a few words about random number generation, and finish with our first major example of graphics in action: a rewrite of the *Oh! Pascal!* frisbee bar graph program.

The **case**
Statement
(138)

case constants
(139)

The "Golden Rule of case Constant Lists, " (Every potential value of the **case** expression must be specified in the **case** constant list) doesn't really apply in Macintosh Pascal. Mac Pascal, like many other versions of the language, allows **case** statements to include an **otherwise** clause. If the case selector does not match any of the case constants, and the **case** statement includes an **otherwise** clause, the statement following the reserved word **otherwise** is executed, and no error results. If there is no **otherwise** clause *and* no match, an error occurs. The **otherwise** clause is especially useful for catching "none of the above" exceptions:

```
case Score of
  10 :
    writeln('Exceptionally good');
  8, 9 :
    writeln('Good');
  5, 6, 7 :
    writeln('Barely Passing');
  3, 4 :
    writeln('Flunking');
  0, 1, 2 :
    writeln('Exceptionally Flunking');
  15 :
    writeln('Something tells me you cheated');
otherwise
  writeln('Error in Score')
end; {case}
```

(By the way, **case** constants may be any ordinal type except *longint*.)

random number
generation
(145)

You may remember from Chapter 1 that Macintosh Pascal, has a built-in function, *Random*, that generates random *integer* values from -32767 through 32767. *Random* uses a different seed each time a program is run, which is appropriate when it's used in a game or other application where surprise is important. On the other hand, there are some situations when you *want* the same sequence of numbers every time you run the program – when you're debugging your game program, for example. For those situations, you can preset the built-in seed *randSeed* with the statement

```
randSeed:=1;
```

If you want a random number generator that returns values between zero and one, so you can substitute it for the *random* function on page 145 of *Oh! Pascal!*, all it takes is a simple conversion:

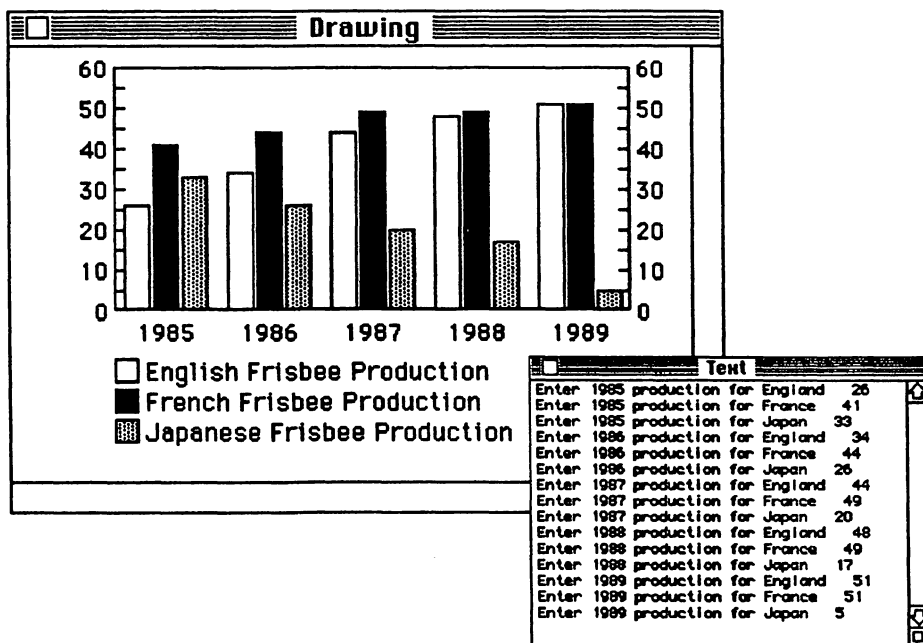
```
function Real_Random : real;
{Returns real random numbers between 0 and 1}
begin
  Real_Random := abs(Random/MAXINT)
end;
```

bar graphing
program
(150)

The *GraphMaker* program in *Oh! Pascal!* produces adequate bar graphs, given that it builds the bars from rows of text characters. But that kind of graph wouldn't impress the board of directors at the Frisbee factory. The QuickDraw subprograms in Mac Pascal allow us to produce graphics more like those we see in magazines and business reports.

Character output in Standard Pascal is printed from left to right one row at a time, so the position of each succeeding character is predetermined. That's why the bars in the go from left to right.

QuickDraw graphics allow us to draw things anywhere in the Drawing window in any order. With freedom comes responsibility: we *have to* tell the computer where to draw each element of our graph. This version of *Graph_Maker* is longer because it includes many lines of code that do nothing more than tell the pen where to draw each element. Is it worth the extra trouble? You decide....



5 Making Choices: the **case** Statement

```
program Graph_Maker; {Draws a bar graph, Macintosh style.}
const
  TOP = 10; {Constants that determine the size of the graph.}
  BOTTOM = 130;
  LEFT = 50;
  RIGHT = 300;
  ENGLAND = 1; {The countries.}
  FRANCE = 2;
  JAPAN = 3;
var
  Bar_Pos, Country, Production, Year : integer;

procedure Draw_Axes; {Draws the graph axes with tick marks.}
var
  Tick_Mark, Height : integer;
begin
  FrameRect(TOP, LEFT, BOTTOM, RIGHT); {Draw the axes.}
  for Tick_Mark := 1 to 11 do {Draw the tick marks.}
    begin
      {Compute the vertical position of the tick mark.}
      Height := BOTTOM - Tick_Mark * 20;
      MoveTo(LEFT, Height); {Draw a tick mark on the left axis.}
      LineTo(LEFT + 5, Height);
      MoveTo(RIGHT - 1, Height); {Draw a tick mark on the right axis.}
      LineTo(RIGHT - 6, Height)
    end {for}
  end; {Draw_Axes}

procedure Label_Vertical_Axes;
var
  Production, String_Length, Height : integer;
  Labels : string;
begin
  for Production := 0 to 6 do {Label the vertical axes.}
    begin
      TextSize(10); {Set text size to 10 point.}
      Labels := StringOf(Production * 10 : 1); {Convert the label to type string.}
      String_Length := StringWidth(Labels); {Determine the size of the label.}
      {Center the label position on the vertical axis.}
      Height := BOTTOM - Production * 20 + 5;
      MoveTo(LEFT - String_Length - 3, Height); {Draw label on the left axis.}
      DrawString(Labels);
      MoveTo(RIGHT + 3, Height); {Draw the label on the right axis.}
      DrawString(Labels)
    end {for}
  end; {Label_Vertical_Axes}
```

```

procedure Label_Horizontal_Axis;
var
  Year, String_Length, Height : integer;
  Labels : string;
begin
  Height := BOTTOM + 15; {Compute vertical position for the date labels.}
  for Year := 0 to 4 do {Draw date labels.}
  begin
    Labels := StringOf(Year + 1985 : 1); {Convert year to a string.}
    String_Length := StringWidth(Labels); {Compute label size.}
    {Compute starting position for label.}
    MoveTo(50 * Year + 25 + LEFT - (String_Length div 2), Height);
    DrawString(Labels) {Draw label on horizontal axis.}
  end {for}
end; {Label_Horizontal_Axes}

procedure Draw_Graph_Legend;
begin
  TextSize(12); {Set text size to 12 point}
  FrameRect(BOTTOM + 24, LEFT, BOTTOM + 36, LEFT + 12); {Draw bar type.}
  MoveTo(LEFT + 15, BOTTOM + 35);
  DrawString('English Frisbee Production'); {Label bar type.}
  PaintRect(BOTTOM + 39, LEFT, BOTTOM + 51, LEFT + 12);
  MoveTo(LEFT + 15, BOTTOM + 50);
  DrawString('French Frisbee Production');
  FrameRect(BOTTOM + 54, LEFT, BOTTOM + 66, LEFT + 12);
  FillRect(BOTTOM + 55, LEFT + 1, BOTTOM + 65, LEFT + 11, ltgray);
  MoveTo(LEFT + 15, BOTTOM + 65);
  DrawString('Japanese Frisbee Production')
end; {Draw_Graph_Legend}

procedure Draw_Bar (Production, Country, Bar_Pos : integer);
begin
  {Fill the bar.}
  case Country of
    ENGLAND :
      FillRect(BOTTOM - Production * 2, Bar_Pos, BOTTOM, Bar_Pos + 13, white);
    FRANCE :
      FillRect(BOTTOM - Production * 2, Bar_Pos, BOTTOM, Bar_Pos + 13, black);
    JAPAN :
      FillRect(BOTTOM - Production * 2, Bar_Pos, BOTTOM, Bar_Pos + 13, ltgray);
  end; {case}
  {Outline the bar}
  FrameRect(BOTTOM - Production * 2, Bar_Pos, BOTTOM, Bar_Pos + 13)
end; {Draw_Bar}

```

5 Making Choices: the **case** Statement

```
begin {Graph_Maker}
  ShowText;
  ShowDrawing;
  Draw_Axes;
  Label_Vertical_Axes;
  Label_Horizontal_Axis;
  Draw_Graph_Legend
  for Year := 1985 to 1989 do
    begin
      {Compute the horizontal position of the bars to be drawn.}
      Bar_Pos := (Year - 1985) * 50 + LEFT + 5;
      for Country := ENGLAND to JAPAN do
        begin
          case Country of
            ENGLAND :
              write('Enter ', year : 4, ' production for England ');
            FRANCE :
              write('Enter ', year : 4, ' production for France ');
            JAPAN :
              write('Enter ', year : 4, ' production for Japan ');
          end; {case}
          readln(Production);
          Draw_Bar(Production, Country, Bar_Pos);
          Bar_Pos := Bar_Pos + 14
        end {for Country}
      end {for Year}
    end.
```

Still More Exercises

5-36 With Macintosh Pascal's built-in graphics procedures, it's no harder (or easier) to draw a vertical graph than it is to draw horizontal one. Write two procedures to draw the graph in 5-29, one procedure for the horizontal form and one for the vertical form.

5-37 Use the random number generator to take a random walk about the Drawing window. Trace your path with a line. Each line should have a random length (not to exceed the bounds of the window) and a random direction.

5-38 To make your random walk more interesting, place a small shape in the window and calculate how far you have to walk before your path crosses the shape. (This is a variation on the classic "drunk and lamppost" question.)

Programming Decisions: the if Statement

AN IF STATEMENT IN MACINTOSH PASCAL is just like an if statement in Standard Pascal, so this chapter supplement is shorter than most. We'll discuss how *boolean* variables are initialized, read, and written in Macintosh Pascal, and then leave you to writing your own programs.

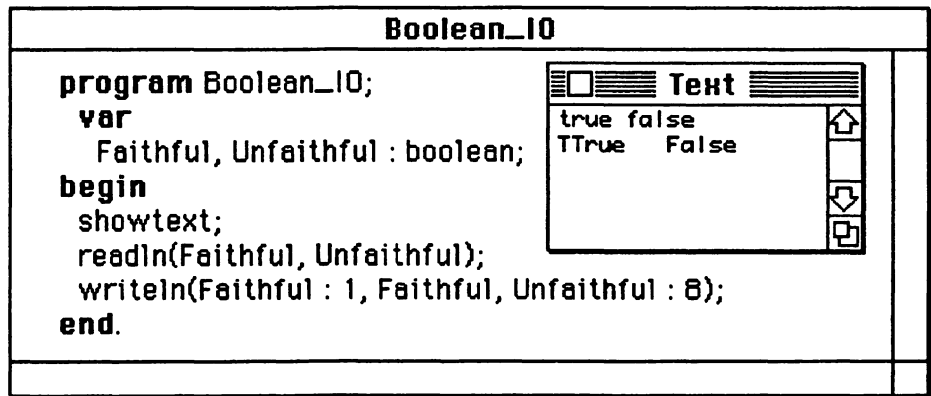
boolean Expressions and if Statements (176)

boolean input
and output

In Macintosh Pascal all *boolean* variables are initialized to a value of *false* when program execution begins, which means that you won't get a ladybug if you forget to assign an initial value to a *boolean* yourself. You may get a more subtle bug instead because your boolean has a value that may or may not be what you intended. Be wise ... initialize!

In Mac Pascal, *boolean* variables may be read using *read* or *readln*. The only acceptable values for input are *true* and *false*. These must be completely spelled out in upper case, lower case, or some combination of the two. Like numbers, *boolean* values must be separated by spaces or **Return**s (end-of-lines) when they're read as input.

Boolean variables may also be written using *write* or *writeln*. The output format of these values may be specified. Here's an example; note how the one-character field width of the first variable is handled:



Making Actions Continue: the Conditional Loops

Macintosh Pascal's conditional loops **repeat** and **while**, are 100% standard, so what you read in Chapter 7 of *Oh! Pascal!* tells you all you need to know about them. We'll show you here how a loop can be used to animate the drawing window and we'll end with a word on getting out of endless loops.

The **repeat** and **while** statements (216)

Movies don't really move; they just mimic the way the world works. Rather than showing seamless change and movement, they rely on an illusion of motion created by rapid repetition of nearly identical frames. Mac Pascal allows you to do something similar; using graphics procedures and conditional loops you can create simple animation. Here's an intriguing example (a modified version of a demo program that's included with each copy of Macintosh Pascal) showing how a **repeat** loop is used to move a bouncing ball across the Drawing window **until** it reaches the edge.

Using Repetition for Animation

```

program Bouncing_Ball; {Watch a ball bounce across the window}
const
  PICTURE_HEIGHT = 100;
  PICTURE_WIDTH = 400;
  BALL_SIZE = 8;      {Changing the constants creates}
  GRAVITY = -0.5;     {different animated effects}
  BOUNCINESS = 0.9;
  COURT_LEVEL = 100;
var
  Horizontal_Position : Integer;
  Vertical_Position, Velocity : Real;

procedure Draw_Ball (Vertical_Position : Integer);
var
  Top, Left, Bottom, Right : Integer;
begin
  Top := COURT_LEVEL - Vertical_Position - BALL_SIZE;
  Left := Horizontal_Position - BALL_SIZE;
  Bottom := COURT_LEVEL - Vertical_Position + BALL_SIZE;
  Right := Horizontal_Position + BALL_SIZE;
  FrameOval(Top, Left, Bottom, Right); {Use PaintOval for a solid ball}
end;

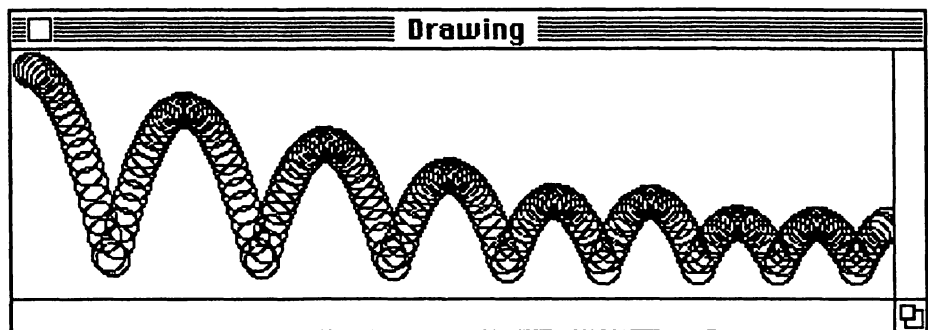
{continued}

```

```

begin {bouncing ball}
  ShowDrawing;
  Horizontal_Position := BALL_SIZE + 1;
  Vertical_Position := PICTURE_HEIGHT - BALL_SIZE - 1;
  Velocity := 0;
  Draw_Ball(round(Vertical_Position));
  repeat
    Horizontal_Position := Horizontal_Position + 2;
    Velocity := Velocity + GRAVITY;
    Vertical_Position := Vertical_Position + Velocity;
    If Vertical_Position <= 0 then
      begin
        Vertical_Position := Abs(Vertical_Position);
        Velocity := -(BOUNCINESS * Velocity);
      end;
    Draw_Ball(round(Vertical_Position));
  until Horizontal_Position >= PICTURE_WIDTH;
end. {bouncing ball}

```



As you can see, the bouncing ball leaves a trail behind it so you can see its path. Since real-world balls don't leave trails, you might want to try making a change in the program so that it erases each old ball before it draws a new one. If so, insert the following statement right before the *FrameOval* statement at the end of procedure *Draw_Ball* and run the program again.

```
EraseOval(Top-10, Left-10, Bottom+10, Right+10)
```

Antibugging and Debugging (246)

boundary
conditions
(246)

The Macintosh doesn't impose a time restriction on the execution of a program. If your program enters an infinite loop, it just keeps rolling until you stop it (or the power fails). The graceful way to stop it is by selecting the Halt option from the Pause menu that appeared when the program started running. The finger will point to the statement which was about to be executed when you halted the program. You can then use your trusty Observe window to peek at the value of the variable or expression that was supposed to terminate your **repeat** or **while** statement. Once you've found the problem, you can restart the program from the beginning by selecting Reset from the Run menu and then selecting Go. (If you don't Reset first, the program will just continue from where it left off.)

Yet Another Exercise

7-36 You've been hired by the First Interplanetary bank of Corvallis to write a front end for the bank transaction program that your tellers are using. The program will eventually call procedures to do five different tasks: record a deposit, record a withdrawal, display the balance, record a loan payment, and print a check. Tellers will select a task by entering a response code to a menu of choices; one code matches each task. In addition to these five choices, the menu displays an option for quitting the program. Your program should display a readable menu in a reasonable format, accept a typed response from the teller, call a dummy procedure corresponding to the response. (A dummy procedure might simply write the message, "printing balance," for example.) If the teller enters a code that doesn't match any of the valid choices, your program should request a new code. This process continues until the code for quit is entered. Make your program as friendly as you can, given that you're limited to working with the keyboard and the text window.

Character-Oriented Computing: Text Processing

TEXT PROCESSING *can* be done in Macintosh Pascal just like it's done in Standard Pascal. But Mac Pascal's **string** data type makes most text processing operations much easier and more intuitive, as you'll see from the examples in this chapter.

Chapter 8 of *Oh! Pascal!* covers many of the nitty-gritty details of text file processing. Because text file processing often confuses beginning programmers, we've provided a program on the *Oh! Mac Pascal!* disk which graphically illustrates the basic concepts.

Text Processing (260)

In Standard Pascal, text processing means working with characters. In Mac Pascal, text processing is working with strings. Strings allow groups of characters to be treated as single variables. Since characters tend to come in groups in the real world, text processing with strings is much more natural and intuitive. Consider *Echo Text*, the basic text processing program on page 265 of *Oh! Pascal!*, rewritten here using **string** variables.

```

program Echo_Text;
{Read and echo a file of text.}
var
  Line : string;
begin
  while not eof do
  begin
    readln(Line);
    writeln(Line)
  end {while}
end. {Echo_One_Line}

```

There's no need to check for end-of-line in this program, because when the **string** is read with the *readln*, the *eoln* character serves as the terminator of the string – and the *readln*..

Using Strings with Relational Operators

Like characters and numbers, strings can be compared with each other and assigned to each other. (Characters are compatible with strings and may be assigned to **string** variables.) All of the usual relational operators (<,>,=,<=,>=) are valid with strings. Dictionary order – not length – determines the relation, so 'Sue' is greater than 'Sarah'.

```

Name := 'Sue';
Other_Name := 'Sarah';
if Name > Other_Name then
  writeln(Name)
else
  writeln(Other_Name); {writes "Sue" to the text window.}

```



String Functions



Strings become even more useful when they're used with the **string** functions built into Mac Pascal. Many of the programs in *Oh! Pascal!* can be simplified by taking advantage of these built-in functions.

Counting characters, for example, involves nothing more than a simple call to the function *length*. *length* returns an *integer* value representing the length in characters of its **string** argument (the number of characters currently stored in the variable).

function *length*

```
String_Variable := 'Harpo Marx';
writeln(length(String_Variable)); {Writes the value 10.}
```

There are several other useful string functions in Macintosh Pascal. Strings may be combined end to end with the function *concat*.

function *concat*

```
concat(String_1,String_2,...,String_n)
```

concat will put together any number of **string** expressions as long as the resulting string is less than 255 characters long. Here's a simple example that combines two **string** variables with a **string** constant:

```
last:= 'Frog';
first:= 'Kermit';
writeln(concat(last, ', ', first)); { Writes "Frog, Kermit".}
```

justifying
string output

Here's a more complex example. When writing a **string** variable using

```
writeln(word:20);
```

the output looks like this (— represents a blank):

```
—————Felix—Cat
```

You can use the function *concat* to fill *Name* with spaces.

```
for counter := 1 to 20-length(Name) do
  Name := concat(Name, ' ');
```

Then `writeln(word:20)` yields

```
Felix—Cat—————
```

The *pos* function searches for the position of a sub-string value (constant, variable, or expression) within a **string** value. An *integer* value specifying the starting position of the sub-string in the string is returned. Zero is returned if the sub-string is not found in the string.

function *pos*

```
writeln (pos('at','Patty')); {Writes the value 2 in the text window}
writeln (pos('ta','Patty')); {Writes the value 0 in the text window}
```

The *copy* function is handy for extracting part of a string. It returns a **string** value with any number of consecutive characters copied from anywhere in a specified **string** expression. The function requires three parameters; a **string** value from which the characters are to be copied, an *integer* value which specifies the position of the first character in the string that's to be copied, and an integer value specifying the number of characters to be copied. If the sum of the last two parameters is greater than the length of the first parameter then all the characters beginning at the one specified by the second parameter are copied.

```
function copy          writeln(copy('Patty',2,2)); {Writes the string "at".}
                      writeln(copy('Patty',3,5)); {Writes the string "tty".}
```

The *omit* function returns a sub-string of a **string** value with any number of consecutive characters from the original string omitted. The original **string** value is unchanged. The function requires three parameters: a **string** value, an *integer* value specifying the position of the first character to be omitted, and an *integer* value specifying the number of characters to be omitted.

```
function omit          writeln(omit('Patty',3,2)); {Writes the string "Pay".}
```

Include is the opposite of *omit*. It returns a super-string of a **string** value with characters added to the original string. The function requires three parameters: a **string** value representing the characters to be included, a **string** value which will receive the characters to be included, and an *integer* value specifying the position where the characters are to be included.

```
function include       writeln(include('tt','Pay',3)); {Writes the string "Patty".}
```

String Procedures



The **string** functions you've seen so far behave like good functions should, in that they don't change the values of their parameters. But there are times when you want to change the value of a **string** variable by adding characters to it or taking some away. There are two **string** procedures with *variable* string parameters: *delete* and *insert*.

The procedure *delete*, as you might expect, removes characters from a **string** variable. The procedure requires three parameters; a **string** variable, an *integer* value specifying the position of the starting character to be deleted, and an *integer* value specifying the number of characters to be deleted.

```
procedure delete      String_Variable := 'Patty';
                      delete(String_Variable,3,2);
                      writeln(String_Variable); {Writes the string "Pay".}
```

The procedure *insert* puts a **string** value into **string** variable. The procedure requires three parameters; a **string** value to be inserted, a **string** variable into which the **string** value is inserted, and an *integer* value specifying the position of the insertion in the **string** variable.

```

procedure
  insert      String_Variable := 'Pay';
              insert('tt',String_Variable,3);
              writeln(String_Variable); {Writes the string "Patty".}

```

Let's put some of these built-in procedures and functions to work by writing a procedure to do a simple search-and-replace operation. You've seen a program with search-and-replace capability already: the Macintosh Pascal editor. The Everywhere option in the Search menu allows you to change, say, "lead" to "gold" wherever it appears in your program. Procedure *Replace_All* will take as input a string representing a Line of text, scan it for occurrences of a particular string Old, and replace each occurrence of that string with string New.

```

procedure Replace_All (var Line : string;
                       Old, New : string);
{Replace all occurrences of the string Old with the string New}
var
  Old_Length, Position : integer;
begin
  Old_Length := length(Old);
  Position := pos(Old, Line);
  while Position > 0 do
    begin
      delete(Line, Position, Old_Length);
      insert(New, Line, Position);
      Position := pos(Old, Line)
    end {while Position}
  end; {Replace_All}

```

Suppose the string 'We all live in a yellow submarine, a yellow submarine!' has been read into the **string** variable Line. We can repaint the submarine with a call to *Replace_All*:

```

Replace_All (Line, 'a yellow', 'an aquamarine');
writeln (Line);
↓      ↓      ↓      ↓      ↓      ↓      ↓      ↓
We all live in an aquamarine submarine, an aquamarine submarine!

```

Self-Check
Question

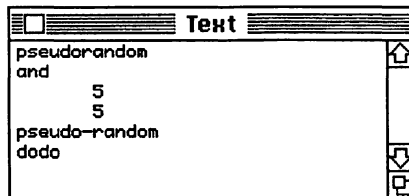
Q. What's the output from this nonsensical program?

```

program stringstuff;
var word, prefix: string;
begin
    word := 'random';
    prefix := 'pseudo';
    writeln(concat(prefix,word));
    writeln(copy(word,2,3));
    writeln(pos('do',prefix));
    writeln(length(omit('dangle',1,1)));
    insert(prefix,word,1);
    writeln(include('-',word,7));
    delete(word,1,4);
    delete(word,3,3);
    delete(word,length(word),1);
    writeln(word);
end.

```

A.



The File Window and External Files*

(280)

the file window
(280)

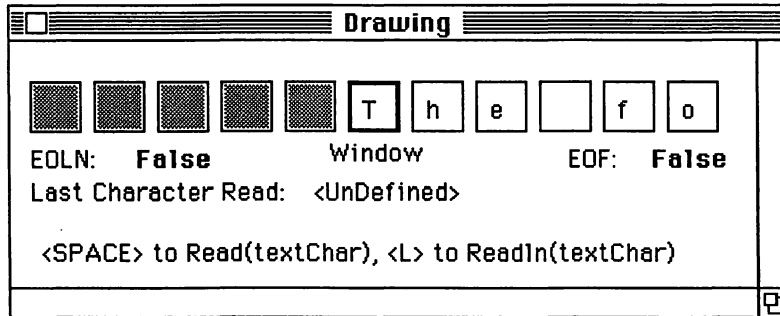
If you read the optional section 8-2 in *Oh! Pascal!*, you learned about Pascal's 'lookahead' mechanism that allows you to peek at the next character in line to be read from a text file. The file window is a simple concept that's not particularly simple to describe. Consequently, programmers learning Pascal from a textbook or lectures often stumble on the idea.

To make things easier for you, we've included a program called "Text File Window" on the *Oh! Mac Pascal!* disk that illustrates the process of reading a text file on screen. The program represents any text file as a train of boxes that chugs leftward each time a character is read. The box in the center – the one with the dark outline – represents the file window through which you can see the character that's about to be read. As each character is read, the program shows the new contents of the file window and the current status of the *eof* and *eoln* functions. When you run the program, it asks you via a dialog box to elect a text file to be processed. Any text file may be used as input; we're using a file containing the verse at the bottom of page 287 of *Oh! Pascal!* (That file is called "poets" on the disk.)**

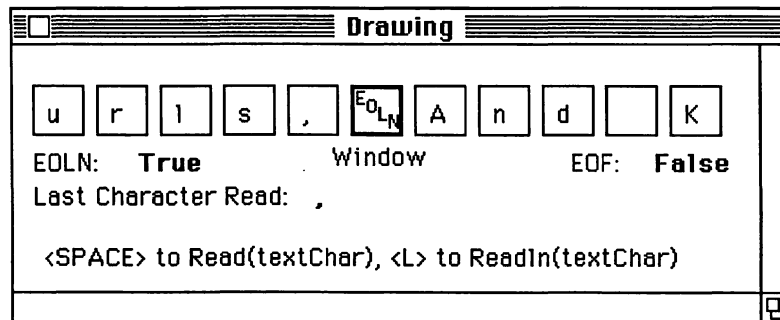
* This section may be read now or as a supplement to Chapter 13.

** The program used for these figures was written by Mark Borgerson, author of *From Basic to Pascal* (Wiley). Mark used several advanced features of Mac Pascal that we haven't covered yet, but you'll still find it worthwhile to spend some time looking at the code to see how it was written.

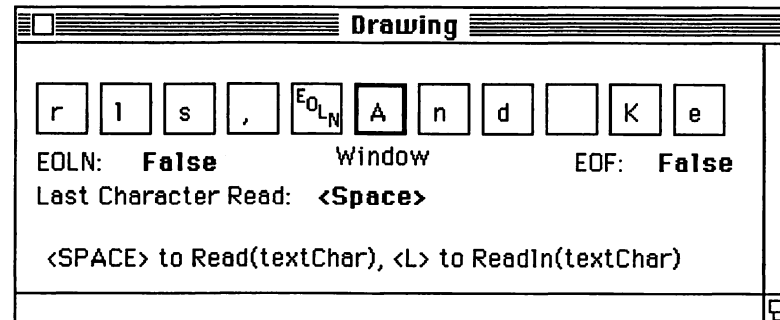
When the file is first opened, the drawing window shows us this:



When the character before the first *eoln* is read, the file window points to the *eoln*, and the *eoln* function returns True:



When the *eoln* is read, *eoln* returns to false and Last Character Read looks like a space:



If you run this program on through the file, you'll discover one of the best kept secrets of Mac Pascal: there is no *eof* character at the end of this, or any other, Macintosh file. Instead of sticking an extra character at the end of each file, the Macintosh operating system keeps track of the *number* of bytes (characters) in each file. By also keeping track of the number of bytes that have been read, the Mac can always tell when it reaches the end of the file. As you'll see when you run the program, the end result is exactly the same as if there were an *eof* character there.

External Files (283)

From a programmer's point of view, the mechanics of processing text files is essentially the same in Macintosh Pascal as in Standard Pascal, except that the **string** data type simplifies many potentially messy coding problems. But whether your programs read and write their text files one character at a time or one string at a time, they'll use a non Standard method for communicating file information to the Macintosh operating system. Standard Pascal uses program parameters in the program header to pass the names of files to the computer's operating system. Macintosh Pascal doesn't generally use program parameters (and Mac Pascal programs don't require them); it offers several alternatives, most of which are easier to understand and use.

Preferences



You've already seen (in your second hands-on session) the easiest way to direct text output to someplace besides the screen: use the Preferences option (in the Windows menu) to route your output to the printer. You can also use Preferences to specify that output is to be sent to a text file by clicking the "Output also to a File" box. The Mac responds with a familiar dialog box asking you to type the name of the file; Drive and Eject buttons allow you to specify the destination disk. If a file with your chosen file name already exists on the disk, Mac asks you if you want to replace the existing file. Be careful; there's no turning back if you write over the top of an important file.

Preferences allows you to send your program's standard text output – anything written by a *writeln* or a *write* without a file parameter – to the printer, a text file, or both. Whichever combination you choose, the output is still displayed in the text window, too. In effect, clicking one of the alternative output boxes in Preferences temporarily creates an additional standard output file.

The main problem with this approach is that it has to be set up in advance each session; your preferences only remain in effect until you override them with a new set of preferences, or until you end your Mac Pascal session. If the person running your program forgets to specify a file preference – or doesn't know how – then standard output goes only to the text window. The alternative is to design the program in such a way that it asks the user where the output is to go each time it's run. That means a little more work on your part when you're coding; you have to

1. explicitly open the file using the standard procedure *rewrite*;
2. include the file name as the first parameter in each *writeln* and *write*;

Step 2 is the same as with Standard Pascal; non Standard output files always have to be explicitly referenced in output statements. Step 1 looks Standard, but the details aren't.

rewrite (284)

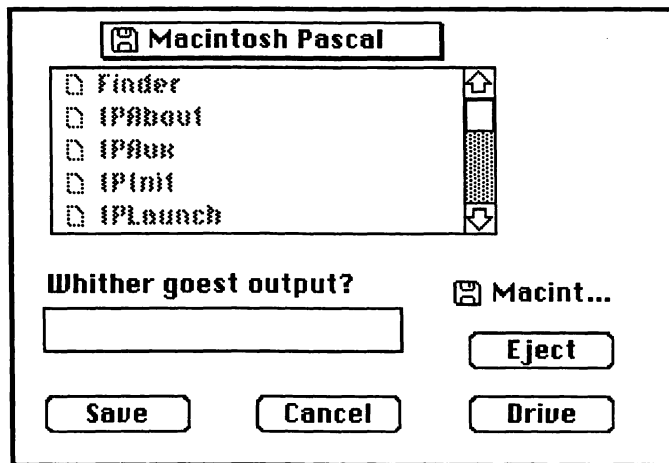
Like Standard Pascal's *rewrite*, the Macintosh *rewrite* must be included before the first *write* or *writeln* to the file. But the Macintosh *rewrite* also does the file-association work that's done in the program heading of a Standard Pascal program. That means *rewrite* must have two parameters:

1. The identifier that's going to represent the file in this program; that is the identifier that appears as the first parameter in *write* and *writeln* statements, just like in Standard Pascal.
2. The name of the actual disk file, represented as a **string**. This parameter *could* be a literal or a previously-assigned **string** variable, provided it specifies the disk and file names appropriately. But this approach doesn't allow much flexibility, and it doesn't protect you from accidentally writing over an important file with the same name. It's safer, easier, and more elegant to allow the user to specify the file name via your very own customized dialog box.

The built-in function *NewFileName* allows you to do just that. It displays a dialog box, complete with whatever prompt you include as a **string** parameter. When the program user responds to the dialog box prompt by selecting a disk (using Drive and Eject buttons) and typing the file name, *NewFileName* checks whether that disk contains a file with that name and issues a last-chance warning if it does. Otherwise, *NewFileName* returns a **string** value which includes the names of the disk and the file in the appropriate format for use in a *rewrite* statement.

function
NewFileName

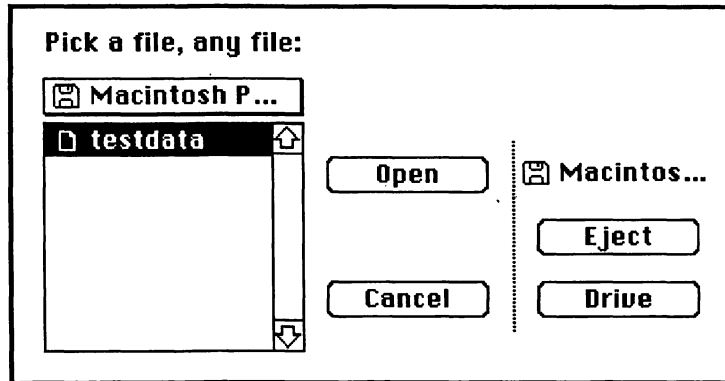
```
rewrite(Output_File, NewFileName("Whither goest output?"));
```



So far we've only talked about non Standard output files. What if you want to read data from a file other than the keyboard? The procedure *reset* and the function *OldFileName* do for input what *Rewrite* and *NewFileName* do for output. *Reset* formally opens a file with an identifier specified by its first parameter; the second parameter is a **string** indicating the actual file name and its disk location. When *OldFileName* is used as the second parameter, it displays a dialog box that lets the user select an input file at run time. The dialog box is the same one that you see when you open a Pascal file; the only difference is the prompt, specified as *OldFileName*'s **string** parameter.

reset
function
OldFileName

reset(File_Identifier), OldFileName('Pick a file, any file:');



To summarize, let's look at the file copy program *Duplicate* from page 283 of *Oh! Pascal!*, which has been rewritten to take advantage of Mac Pascal's string and file-handling extensions. Notice the differences:

1. The Input and Output parameters are absent from the program statement.
2. The *reset* and *rewrite* statements have two parameters, the file identifier and the function that prompts for the file name;
3. The text is read as a series of strings rather than a series of characters. This simplifies the code, eliminating the inner loop and the necessity of checking for *eof*.

file copy program
(283)

```

program Duplicate;
{Demonstrates copying of external files, Mac Pascal style.}
var
  Old, New : text;  {The external files' type.}
  Current : string; {A buffer for characters.}
begin
  reset(Old, OldFileName('Output File Name:'));
  rewrite(New, NewFileName('Input File Name:'));
  while not eof(Old) do
    begin
      readln(Old, Current);
      writeln(New, Current)
    end {while statement}
  end. {Duplicate}

```

Still More Exercises

- 8-23 When procedure *Replace_All* in this chapter is executed, what happens if the new string is longer than the string being searched? What happens if the old string continues to a next line? Refine the procedure to handle those conditions. Is this more easily done by modifying the procedure or by modifying the way in which the procedure is called from the calling program unit? Are there other refinements that should be considered?
- 8-24 In the procedure *Replace_All* the search for each occurrence of the old string always starts at the beginning of the line. Make this procedure more efficient by starting each search at the point of the last replacement.
- 8-25 Run the program *TextFileWindow* until you reach the end of the file and observe the status of the window, *eoln*, and *eof*.
- 8-26 Write a program that opens a text file for input and copies the file to another file. The name of the new file is to be the same as the original file name with " Copy" to the name of the original file.
- 8-27 Write a program that takes a file of names in the format: "Duck, Daffy" and writes the name to the Text window in the form "Daffy Duck".
- 8-28 Files may be processed with three terminating conditions: 1) a particular value in the file (called a sentinel) terminates the read process; 2) the number of values in the file is known and exactly that many are read; and 3) reading from the file terminates when the end of file is encountered. Write three versions of a procedure to list the contents of a text file, one for each of these terminating conditions. Will the parameters required by each procedure be the same?
- 8-29 Rewrite the gerund conversion program using strings.

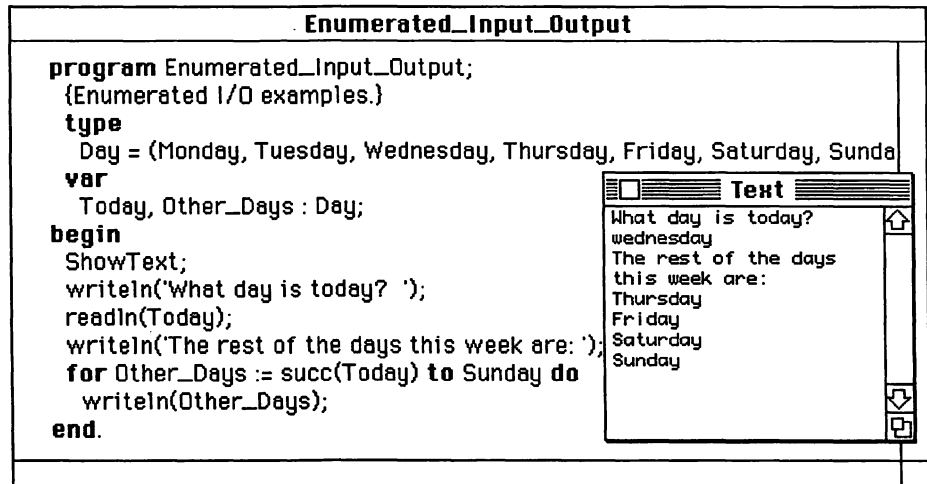
Extending the Ordinal Types

GOOD NEWS! This chapter's short, and it's nothing but good news!

Enumerated Ordinal Types (294)

Pascal's enumerated types allow programmers to write more readable, and therefore more debuggable, programs. This would be even more true if enumerated constants could be read and written in I/O statements. As it turns out, they *can* in Macintosh Pascal.* On input, the only accepted values for a variable of an enumerated type are the identifiers enumerated for that type. Case is irrelevant; *Monday*, *MONDAY*, and *monday* are all the same when read as enumerated values.

enumerated I/O
(297)



Yet Another Exercise

9-12 In Chapter 5's *Graph_Maker* program, we represented countries as integer values. Rewrite that program using an enumerated type for country.

* When compilers translate programs, the source code is not saved. In particular, the identifiers associated with the enumerated type are lost, leaving only numeric constants in their place. In the Mac Pascal interpreter, statements are translated immediately prior to execution, so the identifiers are available for input and output.

Software Engineering

ONE ASPECT OF SOFTWARE ENGINEERING deserves a special mention in any book on Macintosh programming: designing a friendly user interface. The user interface is always part hardware and part software, but no good software engineer today can afford to ignore it.

Writing Programs for People

The earliest computer programs were anything but user friendly. They had to be run by programmers who communicated with the computer by flipping switches and reading patterns of lights on the computer console. When line printers, punch card readers, and teletype keyboards came along, life became easier for programmers, but computing still had to be done by trained professionals. The Macintosh computer is a far cry from those early machines.

Many of the most important ideas behind the Macintosh were born at Xerox PARC Research Lab more than a decade ago. A team of scientists and engineers there spent years studying the man/machine interface, looking for ways to make computers more accessible and functional for people. One of those scientists, Alan Kay, envisioned an inexpensive book-sized machine that would serve as an appointment book, reference library, notebook, journal, calculator, and communication center for tomorrow's professional. Kay's DynaBook would fit easily into a briefcase and be – of course – user friendly.

The technology of the time wasn't capable of producing a DynaBook. Even today, designers can't pack that much machine in such a small package at a reasonable price. But The Xerox PARC team, using large timesharing computers, did manage to produce several important concepts that are still with us today: the bit-mapped screen, the mouse-controlled cursor, windows, icons, pull-down menus – sound familiar?

While these revolutionary software ideas may have originated at Xerox, they were clearly popularized by Apple in the Macintosh. The success of the Mac in the marketplace has changed the face of computing forever. Commodore, Atari, Microsoft, Digital Research, and even IBM have since developed and marketed mouse-and-window operating systems for personal computers. Some analysts predict that the Macintosh-style user interface will become the industry standard, allowing computer users to freely switch between brands of hardware and software without learning new sets of rules.*

What's so special about the Macintosh user interface?

the Macintosh
user interface

- It's visual. You don't need to be a computer scientist to know what that little picture of the disk on the screen means.

* It took a while, but it happened with automobiles.

- It's intuitive. The desktop metaphor turns the confusing world of bits and bytes into an environment that's familiar and comfortable for non-computer people. Neophyte computerists who'd break out in a cold sweat if they had to memorize and type commands like `DEL FRSTRN.PAS` have no trouble dragging a document icon into the trash can.
- It's forgiving. It's almost impossible to do anything potentially destructive without being asked first by the computer if you're sure you want to do it.
- It's consistent. All good Macintosh programs follow the same guidelines, so a user doesn't need to learn a whole new way of doing things every time she switches to a different application.

programming
the user
interface

It's not easy to design software for a Macintosh-style machine; there are all kinds of things that a program has to keep track of constantly (Where is the cursor? Has the mouse been pressed or clicked? Has a key on the keyboard been pressed?) and be able to display and use (icons, pull-down menus, dialog boxes, windows). In short, the programmer has to take care of many things that used to be handled by users.

In some ways, though, Mac programming isn't as tough as it could be. You've probably already seen (in Chapter 8) how a single call to procedure *OldFileName* brings up a dialog box, checks for user input from keyboard and mouse, locates the proper file on the proper disk, and opens the file. It's possible to do all that in a single statement only because of the years of software engineering that are packed into the Macintosh Read-Only Memory, operating system, and programming languages. And *OldFileName* is just one of hundreds of already-been-engineered routines, most of which can be used to make your programs more intuitive and easy to use.

Many Macintosh routines are complex and difficult to use unless you're an experienced programmer who's spent many an hour studying the dense *Inside Macintosh* reference set. Others, like *OldFileName*, are easy to incorporate into even the simplest Mac Pascal programs. We've included a healthy sampling of those special Macintosh routines in the 16 chapters of this book; if you want to delve deeper, we'll tell you how in the last chapter.

All this is not to suggest that the Macintosh is the ultimate personal computer. The microcomputer age has only just begun. We don't even have a DynaBook yet, and Alan Kays all over the world are still dreaming up better ideas. Stay tuned....

Arrays for Random Access

OH! PASCAL!'S CHAPTER 11 IS RICH WITH EXAMPLES illustrating the many applications of arrays. In this chapter we'll add two more that aren't available in Standard Pascal: sound and graphics. Then we'll take another look at the **string** data type, this time as a special kind of array rather than a special kind of simple type. (Once again, you may want to skip the section on strings in *Oh! Pascal!*, since it has so little relevance to Mac Pascal.)

Focus on
Programming:
Arrays
(341)

An Array of
Sounds


If the only thing you've heard your Macintosh say is "beep!" you may be surprised to learn that the Macintosh has three built-in sound synthesizers. Macintosh Pascal has a set of procedures that allow you to use those synthesizers to add synthesized music, speech, and other sounds to your programs.

Pianos, violins, trombones, flutes, drums, freight trains, and human voiceboxes all create sounds by vibrating the air around them. Sound vibrations can be analyzed mathematically as complex waveforms and simulated digitally by electronic synthesizers. This kind of synthesis of complex waves is beyond the scope of this book, but we can show you how to use *Note*, a simple procedure for controlling the simplest of the Mac's three synthesizers. *Note* is no threat to Stradivarius, but it can be easily manipulated into producing simple scales and melodies. *Note* takes three parameters:

1. Frequency - a *longint* value in the range 12 to 783360 which determines the pitch of the sound.
2. Amplitude - an *integer* value in the range 0 to 255 which determines the volume of the sound.
3. Duration - an *integer* value in the range 0 to 255 which determines the length of time the sound is generated in 60ths of a second.

procedure *note*

Here's a program that uses *Note* to play the chromatic (12-tone) scale. Procedure *Initialize* stores all of the note frequencies for one octave in an array, so those notes can be accessed in any desired order. (We only stored one octave's worth because frequencies of notes in higher and lower octaves can be calculated by multiplying or dividing by powers of two. Actually, *all* of the notes can be calculated from one starting frequency.) It's easy to modify this program so that it plays melodies rather than scales; you just need to tell *Note* the name and duration of each note in the melody individually.

```

program Play_The_Scale; {Play a musical scale}
const
  VOLUME = 100; {Amplitude for the square wave synthesizer.}
  DURATION = 20; {Length note is played in 60ths of a second.}
type
  Octave = (A, A_Sharp, B, C, C_Sharp, D, D_Sharp, E, F, F_Sharp, G,
            G_Sharp); {Assign enumerated names to the notes}
var
  Tone : Octave;
  Pitch : array [Octave] of integer; {array to hold the frequencies of Pitch}
procedure Initialize (var Tone : Octave);
begin      {assign the frequency values for each of the Pitch}
  Pitch[A] := 440;
  Pitch[A_Sharp] := 466;
  Pitch[B] := 494;
  Pitch[C] := 523;
  Pitch[C_Sharp] := 554;
  Pitch[D] := 587;
  Pitch[D_Sharp] := 622;
  Pitch[E] := 659;
  Pitch[F] := 698;
  Pitch[F_Sharp] := 740;
  Pitch[G] := 784;
  Pitch[G_Sharp] := 831;
end; {initialize}
begin {play_the_Scale}
  Initialize (tone);
  for Tone := A to G_Sharp do {Up the scale through middle C.}
    Note(Pitch[Tone], VOLUME, DURATION);
  for Tone := G downto A do {Down the scale through middle C.}
    Note(Pitch[Tone], VOLUME, DURATION);
  for Tone := A to G_Sharp do {Play the scale an octave lower.}
    Note(Pitch[Tone] div 2, VOLUME, DURATION);
  for Tone := A to G_Sharp do {Play the scale an octave higher.}
    Note(Pitch[Tone] * 2, VOLUME, DURATION);
end. {Play_The_Scales.}

```

We can't list the output of this program, so you'll have to try it yourself.

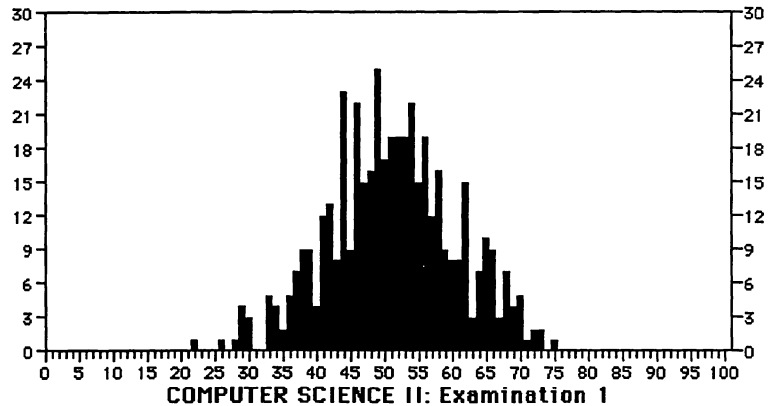
Plotting Arrays of Data



Arrays are incredibly versatile tools for storing large quantities of information, but large quantities of information can be overwhelming if they aren't provided in a form that we can understand. Carefully designed graphic representations can make even the most formidable arrays of data into useful storehouses of information. Whether we're dealing with a chess game simulation or demographic statistics, most of us find it easier to see patterns in the data when they're displayed graphically.

In Chapter 5 we created a simple bar graph. Let's look at a more complex (and more realistic) example.

As teaching assistant for Computer Science II (the hard one), you've been asked to write a program that, among other things, draws a histogram showing the distribution scores on the latest exam:



Our program will perform several operations on student data, so we've decided to read all of the student scores into an array defined like this:

```
const
  MAXIMUM = 500; {Maximum class size.}
  MAX_SCORE = 100;
  MIN_SCORE = 0;
type
  Valid_Values = MIN_SCORE..MAX_SCORE;
  Data_Array = array[1..MAXIMUM] of Valid_Values;
var
  Student_Score : Data_Array;
```

But what we really want for this graph is an array that contains the number of students who earned each score: the number of students who got zeroes, the number who got ones, and so on. That array might be defined like this:

```
type
  Frequency_Array = array[Valid_Values] of integer;
var
  Frequency : Frequency_Array;
```

Think of *Frequency* as an array of 101 turnstiles with counters. After we set all of the counters to zero, we'll need to ask each student to go through the turnstile representing her test score. Since it doesn't matter what order students go through turnstiles (as long as each goes through only once), we might as well treat our array *Student_Score* as a line of students and process them in that order:

```
for i := 1 to Class_Size do {Compute score frequencies.}
  Frequency[Student_Score[i]] := Frequency[Student_Score[i]] + 1;
```

Notice how we're using the array element *Student_Score [i]* as the subscript of another array – *Frequency*. That's logical, legal, and safe, because the base type of the student score array is the same as the index type of the frequency array.

Here's the procedure for calculating and plotting frequencies, less a couple of key procedures for doing the real graphics work:

```

procedure Plot_Frequencies (var Student_Score : Data_Array;
    Class_Size : integer); {Computes and plots test score frequencies.}
var
    Frequency : Frequency_Array;
    Scaling_Factor : real;
    i, Max_Frequency : integer;
    Graph_Title : string;
begin
    for i := MIN_SCORE to MAX_SCORE do {Initialize frequency array.}
        Frequency[i] := 0;
    for i := 1 to Class_Size do {Compute score frequencies.}
        Frequency[Student_Score[i]] := Frequency[Student_Score[i]] + 1;
    Max_Frequency := 0;
    for i := MIN_SCORE to MAX_SCORE do {Find max. freq.}
        if Frequency[i] > Max_Frequency then
            Max_Frequency := Frequency[i];
    Scaling_Factor := Scale (Max_Frequency);
    writeln;
    writeln("Enter graph title (up to 40 characters):");
    readln(Graph_Title);
    Draw_Axes(Graph_Title, Max_Frequency, Scaling_Factor);
    Plot_Bars(Frequency, Scaling_Factor);
end; {Plot_Frequency}

```

plotting
frequencies

After it calculates the frequencies but before it draws calls its graph-drawing routines, procedure *Plot_Frequencies* calculates *Max Frequency*, the number representing the longest bar in the graph. *Max Frequency* is then used in function *Scale* to calculate a scaling factor that adjusts the height of the graph so the bars won't go out of the drawing window or be all scrunched at the bottom:

```

function Scale (Max_Frequency : integer) : real;
var
    Increment : integer;
    Scaling_Factor : Real;
begin
    Scaling_Factor := 10.0; {Find the scaling factor for the vertical axis.}
    while Max_Frequency >= trunc(Scaling_Factor) do
        Scaling_Factor := Scaling_Factor + 10.0;
    Increment := trunc(Scaling_Factor) div 10; {Increment to scale the Axis.}
    Scale := (BOTTOM - TOP) / Scaling_Factor;
end; {Scale}

```

calculating
scaling factor

We'll skip the messy details of drawing and labeling the axes and show you instead how to put the histogram in the scaled graph.

plotting
bars

```

procedure Plot_Bars (var Frequency : Frequency_Array;
    Scaling_Factor : real); {Draws the bars for the histogram.}
const
    BAR_WIDTH = 4;
var
    Score : Valid_Values;
    Vertical, Horizontal : integer;
begin
    for Score := MIN_SCORE to MAX_SCORE do
        begin
            Vertical := BOTTOM - round(Frequency[Score] * Scaling_Factor);
            Horizontal := LEFT + Score * BAR_WIDTH + 2;
            PaintRect(VERTICAL, Horizontal, BOTTOM, Horizontal + BAR_WIDTH)
        end {for}
    end; {Plot_Bars}

```

These procedures are part of the program *Statistics* on the *Oh! Mac Pascal!* disk. If they aren't clear, try adding stops and stepping through the troublesome spots.

Strings as Arrays (351)

We've been using strings since Chapter 1, so they should seem as familiar as the other simple data types. But now you're ready for the truth: the Macintosh Pascal string is more than just another simple data type; it's actually a very special kind of array of characters. Of course, a string is not declared as an array; instead, it's declared using the reserved word **string**:

```
var name : string;
```

You'll remember that a string declared this way has a *maximum* length of 255 characters, although its functional length depends on exactly what's stored in it. Since most **string** variables don't ever come close to the maximum length of 255 characters, Mac Pascal provides a way to make programs more efficient by specifying shorter maximum lengths by providing an integer less than 255 to represent the string's maximum length:

```
var name : string[Maximum_Length];
```

string arrays

This kind of variable is sometimes called a *string array*, but it's no different from any other string except for the explicitly declared maximum length. What does this have to do with the function *length*? Nothing, except that *length* can never be greater than the maximum value that you specify. In general, strings declared this way work just like strings with the default limit, except that you have to be careful not to overstep the length limit you've imposed. Mostly that means not trying to put a value into the string that won't fit. For example, if name has been declared to be type **string** [10], this statement will cause an error:

```
name := 'Rumplestiltskin'
```

It's also incorrect to try to *readln(name)* if the next input line contains 15 characters; you can't put more than 10 characters into a **string**[10] variable.

Individual characters can be accessed in a **string** variable as if the variable were an array with a lower bound of 1. String elements can be used in any way that you might use any *char* variable.

Here's an example that writes out the characters of **string name** in reverse order:

```
for i := length(name) downto 1 do write (name[i]);
```

Notice the use of *length* for the upper limit of the **for** loop. If we had used 10, *MAX_LENGTH*, or any other constant value, we would have risked trying to go beyond the actual current length of the string and write a nonexistent character. In general, it's illegal to reference an element of a string whose subscript is greater than the length of that string as given by the function *length*.

It's handy to be able to access individual string elements with subscripts, but it's seldom necessary when Mac Pascal's built-in string functions and procedures are available. For example, compare this program with its Standard Pascal counterpart on page 352 of *Oh! Pascal!* Both programs are designed to read a line of characters containing two words separated by a blank and print those words out in alphabetical order.

```

string
demonstration
program
(352)
program Order_Words;
  {Demonstrates input, output, and comparison of strings.}
  const
    WORD_LENGTH = 15;
    LINE_LENGTH = 30;
    BLANK = ' ';
  var
    First, Second : string[WORD_LENGTH];
    Line : string[LINE_LENGTH];
  begin
    ShowText;
    writeln('Enter two words. ');
    readln(Line);
    First := Copy(Line, 1, Pos(BLANK, Line) - 1); {Extract first word.}
    Second := Copy(Line, Pos(BLANK, Line) + 1, Length(Line));
    writeln('The words are ', First, ' and ', Second);
    writeln('In alphabetical order the words are...');
    if First < Second then
      writeln(First, BLANK, Second)
    else
      writeln(Second, BLANK, First)
    end.

```

Because it uses **string** input, this version of the program requires an end-of-line (**Return**) to terminate the input; it crashes if the line length or a word length exceed their respective variable definitions. The Standard Pascal version won't crash, but it will give incorrect output. Which is worse? In this case, it doesn't much matter. But in the real world, incorrect output is far more dangerous than no output at all.

Still More Exercises

11-25 You have been given a text file that contains no *eoln* characters. Write a program that reads the text file character by character into a **string** variable whose maximum length is 80 characters. Each time the **string** variable is filled it should be written to a new file using *writeln*.

11-26 Write a program that takes typed values from the computer keyboard and translates them into notes on a musical scale (using the frequencies given in the chapter). Use the QWERTY row to represent the high octave going from A to G-sharp (So the Q-key represents the A above middle-C, the W represents A sharp, etc.), the ASDFG row to represent the middle octave, and the ZXCVB row to represent the low octave. Store the typed values in an array. When a slash (/) is typed, the song is over, and the computer should play the array of notes as typed.

11-27 Make your music composition program friendlier by adding a graphic display of a piano keyboard. Label each piano key with the letter or symbol that represents that note on the computer keyboard.

11-28 Add two more arrays to your music program which allow the composer to control the duration and amplitude as well as frequency of each note. Can you think of a user-friendly interface for the input of duration and amplitude?

11-29 The cumulative frequency of a test score is the frequency of that score plus the total of the frequencies of all lesser scores. Cumulative frequency is usually expressed as a percent of the total number of scores, the percentile. Write a procedure for the program *Statistics* that calculates the percentile for each exam score and plots the percentiles as a line on the graph.

11-30 In question 11-20 you are asked to write procedures and functions that already exist in Macintosh Pascal. Instead, write Mac Pascal statements that perform those string handling tasks.

E Pluribus Unum: Records

YOU'LL FIND NO SIGNIFICANT VARIATIONS from the Standard in this chapter, only some important applications. Many of the Macintosh's hidden tools are defined as record structures. We'll look at a couple.

Predefined Record Types



As you already know, there's a digital clock ticking away somewhere inside every Macintosh. That clock is used by the machine to schedule tasks, but it can also be used by your programs. Suppose you want to print the date and time at the top of the output of your program. The built-in procedure *GetTime* returns the current date and time as calculated by the Macintosh clock. Since the date and time combined include six different values, a six-field record is the logical structure for storing and passing them to *GetTime*. As a convenience, Macintosh Pascal includes a built-in record type that's custom-made for the job:

```
DateTimeRec = record
    Year, Month, Day, Hour, Minute, Second, DayOfWeek : integer
end;
```

About the fields: *Year* is a four digit value that must be greater than 1903 (we don't know why, either); *Hour* is the number of hour since midnight (military time); *DayOfWeek* is valued 1 to 7 (Sunday - Saturday); the rest are obvious. You don't need to include this type declaration in your program, it's automatically there when you need it. The procedure call

checking
the time

```
GetTime(Date_Time);
```

returns all of the date and time information in its variable parameter *Date_Time*, which must be declared to be of type *DateTimeRec*. Here's a procedure to stamp the time and date in your program listing:

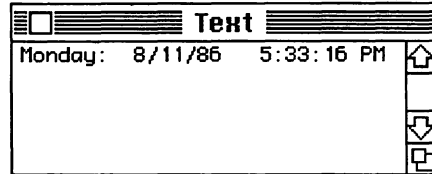
```
procedure Time_Stamp; {Writes the current day, date, and time}
var
    Date_Time : DateTimeRec; {A Mac Pascal predefined record type}
begin
    GetTime(Date_Time); {A built-in procedure that reads the clock}
    with Date_Time do
    begin
        case DayOfWeek of {Convert day from integer code to string}
            1 : write('Sunday');
            2 : write('Monday');
            3 : write('Tuesday');
            4 : write('Wednesday');
```



```

5 : write('Thursday');
6 : write('Friday');
7 : write('Saturday')
end; {case}
write(':', Month : 2, '/', Day : 2, '/', Year mod 100 : 2, ' ');
if Hour < 12 then           {Check for AM vs PM}
  writeln(Hour : 2, ':', Minute, ':', Second, 'AM')
else
  writeln(Hour - 12 : 2, ':', Minute : 2, ':', Second : 2, 'PM')
end {with}
end; {Time_Stamp}

```



Predefined QuickDraw Record Types



There are many predefined record types in QuickDraw – far too many to cover here. (The *Macintosh Pascal Technical Appendix* lists them all.) We'll look at just one here. *Rect* is defined as a special type of variant record which lacks a tag field, but for our purposes we can think of it like this:

```

Rect = record
  Top, Left, Bottom, Right : integer
end;

```

If we declare a variable *Rect_Var* to be of type *Rect*, we can assign values to its fields like this:

```

Rect_Var.Top := 40;
Rect_Var.Left := 30;
Rect_Var.Bottom := 100;
Rect_Var.Right := 150;

```

The work of these four statements can be done with one call to procedure *SetRect*. This procedure has five parameters: a variable parameter of type *Rect* followed by four integer value parameters which represent left, top, right, bottom, in that order. (Note the different order. There are many inconsistencies like this in QuickDraw; it pays to pay attention.)

```
SetRect(Rect_Var, 30, 40, 150, 100);
```

Now that we've defined a record of type *Rect*, what can we do with it? For starters, we can pass it to any of the QuickDraw shape procedures we introduced in Chapter 2. Here's an example:

```
FrameRect(Rect_Var);
```

Macintosh Pascal Functions and Procedures

Screen Management

HideAll 55, 87
SetDrawingRect 55
SetRect 54, 55
SetTextRect 55
ShowDrawing 9
ShowText 9

File Handling

close 57, 58-63
Filepos 60, 64
NewFileName 40, 41, 56-64, 70
OldFileName 40, 41, 45, 56-64
open 59, 60
reset 40, 41, 56
rewrite 39, 56
seek 60

Input and Output

Button 70
DrawChar 14
DrawString 12, 14-15, 26
get 62-63
GetMouse 70
Note 46, 47
page 68
put 59
ReadString 13
reset 40, 41, 56
rewrite 39, 56
SaveDrawing 70
seek 60
WriteDraw 12, 14

String Handling

concat 34
copy 35, 51
delete 35
include 35
insert 35
length 34, 51
omit 35, 37
pos 34
ReadString 13
StringOf 13, 14, 26
StringWidth 13, 14, 26

Miscellaneous

GetTime 53
Random 8, 24, 25, 69

Line Drawing

Line 10, 15
LineTo 10, 15, 26
Move 10, 15
MoveTo 10, 15, 26

Shape Drawing

EraseArc 12
EraseOval 12, 15, 31
EraseRect 12
EraseRoundRect 12
FillArc 12
FillOval 12, 15
FillRect 12, 27
FillRoundRect 12
FrameArc 12, 15
FrameOval 12, 15, 21, 30
FrameRect 11-13, 26, 54
FrameRoundRect 12
InvertArc 12
InvertOval 12
InvertRect 12
InvertRoundRect 12
PaintArc 12
PaintOval 12
PaintRect 12, 50
PaintRoundRect 10, 15

Pen Control

HidePen 10
PenNormal 10
PenPat 10
PenSize 10, 15
ShowPen 10

Graphics Text




CharWidth 14
DrawChar 14
DrawString 12, 14-15, 26
TextFace 14, 15
TextFont 14
TextSize 14, 15, 26
WriteDraw 12, 14

The Mouse

Button 70
GetMouse 70

MACINTOSH DESKTOP COMMAND SUMMARY


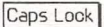

WORKING WITH DISKS

- To start the computer.....Insert Pascal disk, turn on machine
- To initialize (format) a new disk.....1) Insert blank disk (after starting computer)
.....2) Click "One-Sided" or "Two-Sided" or "Initialize"
.....3) Type new disk name
- To eject a disk.....Select disk, select EJECT from FILE menu *or*
.....Select disk, press -Shift-E *or*
.....Press -Shift-1 (internal drive) *or*
.....Press -Shift-2 (external drive) *or*
- To remove a disk from desktop (*not* the startup).....Drag disk icon to trash (also ejects the disk)
- To make a copy of a disk.....1) Insert the disk in one drive
.....2) Insert a blank (or erasable) disk in the other drive
.....3) Drag source disk icon to destination disk icon
- To lock (write protect) a disk.....Slide plastic tab to open hole (close to unlock)
- To open a disk window or folder window.....Double-click icon *or*
.....Select icon, then select OPEN from FILE

WORKING WITH FILES

- To start Macintosh Pascal for creating a program.....Double-click the Pascal icon *or*
.....Select the icon, then select OPEN from FILE
- To open an existing Pascal program.....Double-click the program icon *or*
.....Select the icon, then select OPEN from FILE
- To select several files at once.....Drag a rectangle around the icons *or*
.....Click one icon and shift-click the others
- To copy file(s) between disks.....Drag file icon(s) into window of destination disk
- To remove file from disk.....1) Drag file icon into Trash icon
.....2) Select EMPTY TRASH from SPECIAL
- To lock or unlock a file.....1) Select icon
.....2) Select GET INFO from FILE
.....3) Click LOCKED box

PRINTING

- To print the active window.....Press -Shift-4
- To print the entire screen.....Press --Shift-4
- To print a program from desktop.....1) Select the program icon
.....2) Select PRINT from FILE
- To print a catalog of files.....1) Select the disk or folder to be cataloged
.....2) Select PRINT CATALOG from FILE

Macintosh Pascal Functions and Procedures

Screen Management

HideAll 55, 87
SetDrawingRect 55
SetRect 54, 55
SetTextRect 55
ShowDrawing 9
ShowText 9

File Handling

close 57, 58-63
Filepos 60, 64
NewFileName 40, 41, 56-64, 70
OldFileName 40, 41, 45, 56-64
open 59, 60
reset 40, 41, 56
rewrite 39, 56
seek 60

Input and Output

Button 70
DrawChar 14
DrawString 12, 14-15, 26
get 62-63
GetMouse 70
Note 46, 47
page 68
put 59
ReadString 13
reset 40, 41, 56
rewrite 39, 56
SaveDrawing 70
seek 60
WriteDraw 12, 14

String Handling

concat 34
copy 35, 51
delete 35
include 35
insert 35
length 34, 51
omit 35, 37
pos 34
ReadString 13
StringOf 13, 14, 26
StringWidth 13, 14, 26

Miscellaneous

GetTime 53
Random 8, 24, 25, 69

Line Drawing

Line 10, 15
LineTo 10, 15, 26
Move 10, 15
MoveTo 10, 15, 26

Shape Drawing

EraseArc 12
EraseOval 12, 15, 31
EraseRect 12
EraseRoundRect 12
FillArc 12
FillOval 12, 15
FillRect 12, 27
FillRoundRect 12
FrameArc 12, 15
FrameOval 12, 15, 21, 30
FrameRect 11-13, 26, 54
FrameRoundRect 12
InvertArc 12
InvertOval 12
InvertRect 12
InvertRoundRect 12
PaintArc 12
PaintOval 12
PaintRect 12, 50
PaintRoundRect 10, 15

Pen Control

HidePen 10
PenNormal 10
PenPat 10
PenSize 10, 15
ShowPen 10

Graphics Text




CharWidth 14
DrawChar 14
DrawString 12, 14-15, 26
TextFace 14, 15
TextFont 14
TextSize 14, 15, 26
WriteDraw 12, 14

The Mouse

Button 70
GetMouse 70

MACINTOSH DESKTOP COMMAND SUMMARY


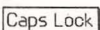

WORKING WITH DISKS

- To start the computer.....Insert Pascal disk, turn on machine
- To initialize (format) a new disk.....1) Insert blank disk (after starting computer)
.....2) Click "One-Sided" or "Two-Sided" or "Initialize"
.....3) Type new disk name
- To eject a disk.....Select disk, select EJECT from FILE menu *or*
.....Select disk, press -Shift-E *or*
.....Press -Shift-1 (internal drive) *or*
.....Press -Shift-2 (external drive) *or*
- To remove a disk from desktop (*not* the startup).....Drag disk icon to trash (also ejects the disk)
- To make a copy of a disk.....1) Insert the disk in one drive
.....2) Insert a blank (or erasable) disk in the other drive
.....3) Drag source disk icon to destination disk icon
- To lock (write protect) a disk.....Slide plastic tab to open hole (close to unlock)
- To open a disk window or folder window.....Double-click icon *or*
.....Select icon, then select OPEN from FILE

WORKING WITH FILES

- To start Macintosh Pascal for creating a program.....Double-click the Pascal icon *or*
.....Select the icon, then select OPEN from FILE
- To open an existing Pascal program.....Double-click the program icon *or*
.....Select the icon, then select OPEN from FILE
- To select several files at once.....Drag a rectangle around the icons *or*
.....Click one icon and shift-click the others
- To copy file(s) between disks.....Drag file icon(s) into window of destination disk
- To remove file from disk.....1) Drag file icon into Trash icon
.....2) Select EMPTY TRASH from SPECIAL
- To lock or unlock a file.....1) Select icon
.....2) Select GET INFO from FILE
.....3) Click LOCKED box

PRINTING

- To print the active window.....Press -Shift-4
- To print the entire screen.....Press --Shift-4
- To print a program from desktop.....1) Select the program icon
.....2) Select PRINT from FILE
- To print a catalog of files.....1) Select the disk or folder to be cataloged
.....2) Select PRINT CATALOG from FILE

Interesting, but not particularly useful, since we already have a perfectly good way of passing values to *FrameRect*.^{*} What we really need is a way to control the size and shape of windows automatically so we can see the entire output from our Frisbee production graph program (Chapter 5) without manually dragging window frames around. As you might have guessed, variables of type *Rect* are also used as parameters for window-management routines. Here's a procedure we can add to the Frisbee program to size the windows automatically:

```
procedure Window_Setup; {Sizes and displays output windows}
var
  Text_Window, Drawing_Window : Rect;
begin
  HideAll;                                {Clear the screen.}
  SetRect(Text_Window, 280, 40, 508, 330); {Build text rectangle record.}
  SetRect(Drawing_Window, 2, 40, 279, 330); {and drawing rect. record.}
  SetTextRect(Text_Window);               {Size and place the text window}
  SetDrawingRect(Drawing_Window);         {and the drawing window}
  ShowText;                               {Display the text window}
  ShowDrawing                             {Display the drawing window}
end; {procedure}
```

In this procedure, *SetRect* is used to load values into *Text_Window* and *Drawing_Window*, both of which are variables of type *Rect*. These two record variables are used in turn as parameters for *SetTextRect* and *SetDrawingRect*, which set their respective windows using the values assigned to their record parameters. This example puts the drawing window on the left of the screen and the text window on the right, so that the two together fill the screen. These procedures and predefined record types may be used to customize the output from all of your Mac Pascal programs.

Still More Exercises

12-18 Define a variant record *Date Time* that stores the time either as military time (0-23 hours) or as civilian time (AM, PM) and the date as American (month, day year) or European (day, month, year).

12-19 In exercise 11-28 you used three arrays to store values of frequency, amplitude, and duration for use by the *note* procedure. Repeat that exercise using an array of records instead of three arrays.

12-20 Represent a deck of cards as an array of records with suit and value. Write a program that shuffles the deck and deals four 13-card bridge hands. Draw the cards using QuickDraw graphics.

12-21 Write a procedure that sizes and shows the Drawing and Text windows. The procedure receives two values, the percent of the screen to be used for the text window (0 to 100) and a code which specifies whether the screen is divided vertically or horizontally.

^{*} Sharp-eyed readers will note an inconsistency here: *FrameRect* had four parameters (top, left, bottom, right) when we introduced it in Chapter 2, but has only one record-type parameter when used in this way. Think of this, not as a contradiction, but as a convenience. Because type *Rect* is a variant record, you can treat it as four simple variables or one structured variable, depending on your needs and your preference.)

Files and Text Processing

WHEN NIKLAUS WIRTH designed Pascal, sequential file processing was the order of the day. But the world moves faster now, and most of today's data storage and retrieval applications are better served by *direct access files*. This chapter will show you how to use this important file type that's not even mentioned in the Pascal Standard.

But first we'll look again at external sequential files, filling in some of the details we left out in Chapter 8. So if you haven't yet read that part of that chapter, now's the time.

Making and Using Textfiles (420)

You may remember from Chapter 8 that Macintosh Pascal handles external files by associating each file with a file identifier in a *reset* or *rewrite* statement, like this:

```
reset(Input_File, String_Var);
```

In that chapter, we introduced *OldFileName* and *NewFileName*, powerful procedures that turn dialog box input into strings that can be used as **string** parameters of *reset* and *rewrite*, respectively. From the program user's point of view, these file name procedures allow input in a friendly and familiar way. From the programmer's point of view, they're handy because they provide what's needed by *reset* and *rewrite* in exactly the right format.

But there are times when you don't want to bother the user for input, even if it's only a click of the mouse. Many programs are designed to work with specific files every time they're run; there's no reason to ask for a file name in those programs when a built-in **string** constant will fill the bill. To use *reset* and *rewrite* without *OldFileName* and *NewFileName*, you'll need to know more about their parameter lists.

In both *reset* and *rewrite*, the file's identifier (the one used inside the program) is the first parameter while a string indicating its name and disk location is the second. The second parameter can be a string constant, variable, or expression. It can be coded as part of the program, entered as data, or computed. The string generally contains two pieces of information: the name of the disk that contains the file and the name of the file. A colon (:) is attached to the end of the disk name so it doesn't run together with the file name. (The disk name may be omitted if the file is on the current disk – the disk on which Mac Pascal resides when it is first opened or the one currently selected with the file dialog box – but it's a good practice to always include the disk name.) Confused? Some examples should help:

reset and
rewrite
(421,422)

```
rewrite(Output_File,'mydisk:myfile');
rewrite(Output_File,String_Variable);
rewrite(Output_File,concat(Disk_String_Var,':',File_String_Var));
```


Reset and *rewrite* open files so they can be used by your programs; the non Standard procedure *close* does the opposite. *Close* is used to close a file and to disassociate the file identifier from the the actual file. *Close* has one parameter, the file identifier:

procedure *close*



```
close(Output_File);
```

While it's not technically necessary to formally close a file if it's the only one you're using, it's a good habit that's especially important when you're writing programs with lots of data files. Here's an example (with no redeeming social value) that shows Mac Pascal files in action:

```
program External_Files; {Create 11 files and write their names to them}
var
  Output_File : text;      {Declare the file identifier.}
  File_Num : integer;
  Disk_Name, File_Name : string;
begin
  ShowText;
  write('Please enter file name ');
  readln(File_Name);      {Read file name as data.}
  write('Please enter disk name ');
  readln(Disk_Name);
  File_Name := concat(Disk_Name, '.', File_Name); {Compute file name.}
  rewrite(Output_File, File_Name); {Open the file to write.}
  writeln(Output_File, 'This is file 0'); {Write file name in the file.}
  close(Output_File); {Close the file so the identifier can be used again.}
for File_Num := 1 to 9 do {Create 9 files, computing their names.}
begin
  rewrite(Output_File, concat(Disk_Name, 'File', chr(ord('0') + File_Num)));
  writeln(Output_File, 'This is file ', File_Num : 1);
  close(Output_File)
end; {for}
  rewrite(Output_File, 'File 10'); {This file name is a string constant.}
  writeln(Output_File, 'This is file 10');
  close(Output_File)
end. {External_File}
```

printer output



Way back in the second hands-on session you learned how to use the Preferences window to send your "Output also to the Printer." You'll remember, though, that this option sends everything to the printer, including prompts and input that's typed during the execution of the program. Consequently, our *Dear_Mom* printout necessarily included typed input above the actual letter.

You can avoid this side effect of the Preferences window by explicitly assigning a file to the printer in your program:

```
rewrite(Print_File, 'Printer:');
```


This statement attaches the file identifier *Print_File* to the printer in the same way the *rewrites* in our last program associated file identifiers to files on specific disks in the disk drives. But unlike a disk drive, the printer can't contain individual files, so there's no file name after the colon in the string. Here, then, is how we could rewrite *Dear_Mom* to produce a printout unblemished by input prompt and response:

```

program Dear_Mom; {This program prints a letter.}
var
    How_much : integer;
    Print_File : text;
begin
    HideAll;
    ShowText;
    writeln('Enter the amount:') {Display prompt in text window}
    readln(How_much);
    rewrite(Print_File, 'Printer:'); {Associate identifier with printer}
    writeln(Print_File, 'Dear Mom,'); {Write letter directly to printer}
    writeln(Print_File);
    writeln(Print_File, 'Please send me $', How_much : 3, '.');
    writeln(Print_File);
    writeln(Print_File, 'Love,');
    writeln(Print_File, 'Skip');
end.

```

other devices



It's possible to associate other devices, including *Keyboard* and *TextWindow*, to file identifiers. This may seem unnecessary, since both of these devices are Standard I/O files in Mac Pascal, but it's occasionally convenient to be able to redirect standard input or output to other devices and back again. *TextWindow* can only be used with the file identifier *output*, and *Keyboard* with *input*. But within those restrictions, you can do this kind of thing:

```

program out(output);
{The program parameter 'output' is necessary to define the file identifier!}
begin
    close(output);
    rewrite(output, 'Printer:');
    writeln('Goes to the printer even though it's a standard writeln.');
```

```

    close(output);
    rewrite(output, 'TextWindow:');
    writeln('Goes to the text window, just like the good old days.');
```

```

end.

```

Direct
Access Files



In *Oh! Pascal!*, as in Standard Pascal, a file is a sequential file – period. As you know by now, the elements of a sequential file must be read in order, starting with the first one. If you need information from the hundredth component in the file, you have to read through the first 99 before you can get what you're after.

sequential files If you decide to change something in that hundredth element, you can't just switch to write mode and make your corrections. Because a file can't be read and written in the same pass, you have to create a new file, copy the first 99 elements from the original, write the new version of the hundredth, and then copy the remaining elements from the original.

In cases where many or most of the elements are changed each time a file is used (such as when the current balances for all checking accounts are updated at the end of the day) this is an efficient way to process files. But there are clearly many applications where this kind of processing just won't do. If airline reservations were stored on sequential files, the delays in ticket lines would be so long that most customers would probably find it faster to take the train – or walk.

direct access files Airline clerks clearly need real-time systems that allow for rapid updates of individual pieces of information. *Direct access files* make such systems possible. With direct access files, a single component can be quickly accessed, modified, and written back without the necessity of reading or copying the other elements. (Direct access files are often called random access files because any randomly chosen element can be quickly accessed.) Procedures for allowing direct (non-sequential) access to files are implemented in most modern versions of Pascal; in Macintosh Pascal these procedures are *open* and *seek*. In addition, many of the standard procedures for working with sequential files can be used with direct access files.

A call to procedure *open* opens a file for both input and output and establishes direct access. The form of the call is the same as for *reset* and *rewrite*:

procedure *open* `open(File_Identifier,String_Var);`

This procedure opens the file indicated in *String_Var* (which can come from *OldFileName*, *NewFileName*, or the program).

When *open* is used with a new file name in the string identifier (as from *NewFileName*), a new file is created with that name. Since the file is empty, *eof* is true and the file window is undefined. If a non-empty file exists with the specified name, *eof* is false and the file window frames the first file component. Either way, you can use either *put* or *write* to add components to the file.

procedure *put* When used with sequential files, *put* always places a component at the end of the file. With a direct access file, *put* places the file component at the current file position – wherever that might be – and then advances the file position one component. Typically direct access files are created initially by adding a batch of new records in a sequence using either *put* or *write*.

Once the file has some components, your program can read or change any of those components directly without scanning the rest of the file. But in order to do that, the program has to tell the computer which component it's looking for. It does that with *seek*, a procedure that effectively says, "locate the fifth (or first, or 5678th) component of the file." Each component has an associated number, starting with the first component that you entered when the file was open. The first component added to a direct access file is numbered zero (for reasons that have to do with binary code),

the second is component number 1, and so on. The *seek* procedure allows us to specify the identifier of the file and the number of the desired component as parameters:

```

procedure seek
    seek(File_Identifier, Longint_Value);

```

Seek moves the file window to the file component specified by the second parameter (which may be any *longint* or *integer* value). Here's how *seek* can be used to move the file window to the end-of-file to add more components:

```

seek (File_Identifier, MAXLONGINT);

```

When cruising around a direct access file in the file window, your program may need to figure out where it is. The function *FilePos* returns a longint value representing the current file position. It requires one parameter, the file identifier. Here's how *FilePos* can be used to move the window ahead 20 frames:

```

function FilePos
    seek (File_Identifier, FilePos (File_Identifier) + 20);

```

As this short program shows, direct access files are easy to read sequentially:

```

program List_Files; {Lists the contents of a direct access file.}
var
    Data_File : file of integer;
    Data : integer;
    File_Position : integer;
begin
    open(Data_File, 'filename');    {Open the file for direct access.}
    while not eof(Data_File) do
        begin
            read(Data_File, Data);
            writeln(Data)
        end {while}
    end. {List_Files}

```

reading
direct access
files sequentially

It's just as easy to treat a direct access file as a true random access file:

```

for File_Position := 1 to 5 do
    begin
        seek(Data_File, random mod FILE_LENGTH);
        read(Data_File, Data);
        writeln(Data)
    end {for}

```

random access

Order Entry
System:
a Realistic
Example

Let's consider a non-random application for direct access files. Al Gorithm's Software Distribution Company stocks hundreds of products which are sold to computer stores all across the country, mostly through telephone orders. Al is trying to computerize the system so that the crucial details of each phone transaction – customer number, product number, and quantity ordered – can be entered into the computer. He'd like the computer to use that information to adjust the product inventory records and to adjust the billing information for the store that placed the order. The system has to update the appropriate records immediately so Al's salespeople know with certainty that the inventory records shown on the screen are accurate. Since there are far too many products and customers to allow the records to be kept in the computer's internal memory, the best alternative is to keep the information in two direct access files: one for all of the customer information and one for product information.

A direct access file is no better than a sequential file, though, unless the program has an efficient way to retrieve an individual record without searching through the others. Each customer has a unique account number; does that help? Let's suppose the account numbers – and only the numbers – were stored in an array in the computer's memory. Since the array contains only account numbers, it doesn't take up much space in memory. And since array searches are much faster than file searches, a complete scan of the array can be done without significantly slowing the program's response time. What happens when the number is matched in the array? If the customer numbers are stored in the array in exactly the same order that they're stored in the file, then the array subscript of the matching number is the address of the corresponding record in the file.

key values In the vernacular, the customer number is the *key* to the customer file, and we're using an array of *key values*. The same scheme can be used to find records in the parts file. Of course, since arrays are kept in the computer's volatile memory, the information in each array will have to be copied to a file when the program isn't being used. If that file is a direct access file, it can be easily updated right along with the corresponding master file.

Let's focus on the algorithm for the order-entry part of the system:

```

Read customer key value file into an array
Read product key value file into an array
Read customer number
while customer number is valid do
  Find the customer number position in customer key value array
  Read customer record from customer file at the same position
  Read product number
  Find the product number position in the product key value array
  Read product record from the product file at the same position
  Read order
  Update the inventory field of the product record
  Update the amount owed field of the customer record
  Write updated records to respective files
  Read customer number
end

```

In Macintosh Pascal:

order entry
program

```

program Order_Entry; {Simple order entry system}
const
  ARRAY_SIZE = 1000; {Maximum number of Products and customers}
type
  Customer_Record = record
    Number : longint;
    Name, Address, City, State_and_Zip : string;
    Balance_Owed, Credit_Limit : real
  end;
  Product_Record = record
    Number : longint;
    Product_Description : string;
    Unit_Price : real;
    In_Stock : longint
  end;
  Customer_File = file of Customer_Record;
  Product_File = file of Product_Record;
  Key_Array = array[0..ARRAY_SIZE] of longint;
var
  {Global file identifier variables}
  Customers : Customer_File;
  Products : Product_File;
  {Local variables}
  Product_Array, Cust_Array : Key_Array;
  Cust_Number, Product_Number : longint;
  Number_Ordered, Total_Products, Total_Cust, Rec_Number : integer;
  Price : real;

procedure Read_Key_File (var Data_Array : Key_Array;
  {Reads a file of key values into an array.}
  var Number_Read : integer;
  Prompt : string); {Prompt for dialog box.}
var
  Key_File : file of longint;
begin
  reset(Key_File, OldFileName(Prompt));
  Number_Read := 0;
  while not eof(Key_File) do
    begin
      read(Key_File, Data_Array[Number_Read]);
      Number_Read := Number_Read + 1
    end; {while}
  close(Key_File)
end; {Read_Key_File}

```

```

procedure Find_Rec_Number (Key_Value : longint;
  var Data_Array : Key_Array;
  var Rec_Number : integer;
  Size : integer);
{Uses a linear search to find the position of a key value in the index array.}
var
  Found : boolean;
begin
  Rec_Number := 0;
  Found := false;
  while (Rec_Number < Size) and not Found do
    if Key_Value = Data_Array[Rec_Number] then
      Found := true
    else
      Rec_Number := Rec_Number + 1
    end; {Find_Rec_Number}

begin {Order_Entry}
  Read_Key_File(Cust_Array, Total_Cust, 'Customer Key File');
  Read_Key_File(Product_Array, Total_Products, 'Products Key File');
  open(Customers, OldFileName('Customer File'));
  open(Products, OldFileName('Products File'));
  write('Enter customer number ');
  readln(Cust_Number);
  while Cust_Number > 0 do
    begin
      Find_Rec_Number(Cust_Number, Cust_Array, Rec_Number, Total_Cust);
      seek(Customers, Rec_Number);
      write('Enter Product number ');
      readln(Product_Number);
      Find_Rec_Number(Product_Number, Product_Array, Rec_Number,
        Total_Products);
      seek(Products, Rec_Number);
      write('Enter size of order ');
      readln(Number_Ordered);
      Price := Number_Ordered * Products^.Unit_Price;
      Customers^.Balance_Owed := Customers^.Balance_Owed + Price;
      Products^.In_Stock := Products^.In_Stock - Number_Ordered;
      put(Products);
      put(Customers);
      write('Enter customer number ');
      readln(Cust_Number);
    end {while}
  end. {Order_Entry}

```

The exercises at the end of this chapter suggest several additions you can make to this program to turn it into a complete, working system.

Antibugging and Debugging (445)

A common error with *put* and *get* in direct access files is the off-by-one error. To make sure you always end up looking at the record you're after, remember three things:

1. Zero is the file position of the first component in a direct access file.
2. Put moves the file position forward by one *after* it writes to the file.
3. Get moves the file position forward by one *before* it reads from the file.

Self-Check Question

Q. What comes out when this program is run?

```

program File_Position;
  {Demonstrates how file procedures affect the file position.}
var
  Play_File : file of integer;
begin
  open(Play_File, OldFileName('Input File'));
  write(Filepos(Play_File) : 2);
  Play_File^ := 0;
  put(Play_File);
  write(Filepos(Play_File) : 2);
  get(Play_File);
  write(Filepos(Play_File) : 2);
  seek(Play_File, 0);
  write(Filepos(Play_File) : 2);
  Play_File^ := 0;
  put(Play_File);
  writeln(Filepos(Play_File) : 2);
end.

```

A. 0 1 2 0 1

Still More Exercises

13-33 We can improve the performance of the *Order_Entry* program by sorting the key values in their respective arrays so the program can use a binary search instead of a linear search to find key values. If we change the order of the records, though, we won't be able to use the subscript of either array to point to the file record position. Instead, we'll have to include the file position information in the arrays. Each array will become an array of two-field records; the first field of each record will be the key value, and the second will be the file position number of the corresponding record. Rewrite the order entry program to order the key values as they are read from their files and to search the key value arrays using a binary search.

13-34 Write procedures that add customers and products for both the original order entry program and for the version modified in 13-33. The advantage of having the key values files direct access (even though they are read sequentially) should be obvious.

- 13-35 Write procedures that delete customers and products from the order entry system. Is it necessary to delete records from the customer and product files?
- 13-36 Write a program that uses the order entry files to print an inventory report listing the quantity on hand for each product.
- 13-37 Write a program that uses the order entry files to print bills for each customer whose amount owed is greater than zero.
- 13-38 Write a procedure that updates the amount owed for a customer when that customer makes a payment.
- 13-39 Write a procedure that updates the quantity on hand to allow new shipments to be added to the inventory.
- 13-40 Work with a team of students to develop a full-featured order entry system following the software engineering principles given in Chapter 10 of *Oh! Pascal!*

Collections of Values: The **set** Type

THERE'S LOTS OF LEEWAY in the Pascal Standard's definition of the **set** data type. The designers of Macintosh Pascal were generous with sets. They made the size limits large enough to handle most any programmer's needs, and only placed one small additional restriction on the way sets can be used.

Defining and Programming Set Types

(458)

base types

(458)

set cardinality

(458)

size limit

(458)

the empty set

Sets may be of the base type *enumerated*, *boolean*, *char*, and *integer*. All of the following are allowed in Macintosh Pascal.

type

```
Reserved_Words = (program, procedure, function, while, repeat, string);
Reserved_Type = set of Reserved_Words;
Character_Set = set of char;
Boolean_Set = set of boolean;
Integer_Set = set of integer;
```

Amazingly enough, the maximum *cardinality* of set size in Macintosh Pascal is 65535, the full range of type *integer*. There are, of course, some practical limitations. For example, sets of *enumerated* types can't be that large because of memory constraints. And alas, *Longint* values outside of the range of *integer* may not be members of sets. Still, the size limit on sets is so large that you're not likely to find an application where it's a factor.

And finally, a minor restriction. The null constant "[]" may not be used with the set operators, so you'll occasionally have to add an extra statement. For example, this

var

```
Alpha : set of char;
if Alpha = [ ] then Do_Something;
```

would have to be changed to this

```
var Alpha, Null : set of char;
Null := [ ];
if Alpha = Null then Do_Something;
```

in Macintosh Pascal.

Abstract Data Structures Via Pointers

EVERYTHING YOU LEARNED ABOUT POINTERS in *Oh! Pascal!* applies to pointers in Macintosh Pascal. Actually, that's not quite true; Mac Pascal isn't quite as restrictive in the way pointers are assigned values.

Advanced Pointer Tools



From a beginner's point of view, Standard Pascal's pointer restrictions aren't a problem; in fact, they provide important protection against costly and frustrating errors. But for advanced programmers, Mac Pascal provides two additional tools for assigning values to pointers: the pointer operator (@) and the function *pointer*.

The pointer function allows you to create pointers of different types that point to the same location. That makes it possible to treat the contents of a given location as both character and numeric data within the same program, for example. The pointer operator makes it possible to point to specific locations in the system's stack, allowing your program to do fancy memory referencing tricks.

Though these two tools are important and powerful additions to the experienced programmers toolbox, they're not particularly useful for most programming tasks, since they're tricky and potentially dangerous in the hands of the untrained. We thought you might want to know about these tools for future reference, but we don't recommend that you use them until you've mastered the basic pointer operations.

If you want to learn more about these advanced tools, consult the *Macintosh Pascal Reference Manuals*, experiment with small test programs, and keep backups of all your important files tucked away in a safe place.

Advanced Topics: Macintosh Pascal and Beyond

WE DON'T PRETEND to tell you everything you want to know about Macintosh Pascal in this introductory guide. In this final chapter we'll pass on a few miscellaneous tricks to add to your Mac bag, including a note on the *page* procedure mentioned in the "Everything You Wanted To Know..." Appendix of *Oh! Pascal!* We'll close with a few words on where to go from here when you're ready to explore further the world of Macintosh programming

Mac Pascal Extensions



As mentioned in *Oh! Pascal!*, the Standard procedure *page* is highly implementation-dependent. In Macintosh Pascal, *page* clears the text window when it's called without an argument. If the file identifier for the printer is included, *page* advances the printer paper to the top of the next page without changing the Text window. An undocumented feature* of Mac Pascal causes the printer to advance whenever *page* is called with or without argument. So if you're using *page* to clear the Text window several times in a program, help save trees by turning off your printer.

procedure *page*
(560)

```
page(output); {or} page; {clears Text window}
rewrite(printer,'printer');
page(printer); {advances printer to the top of the next page}
```

declaration
order

As long as we're in the spirit of full disclosure, we want to mention an extension of Macintosh Pascal that has all kinds of potential for abuse. In spite of what you read in *Oh! Pascal!*, **const**, **type**, **var**, **procedure**, and **function** declarations in Mac Pascal may appear in a block in any order, subject to just two restrictions: an identifier must still be defined before it is referenced and forward referenced pointers must be defined in the same **type** clause. Consider the following confused program:

```
program Non_Standard_Order;
  procedure Print;
    const FIRST = 10;
    begin
      writeln(FIRST, Second)
    end;
  var
    Second : integer;
  begin
    Second := 20;
    Print
  end.
```

* Computer Science terminology for a bug.

In obvious violation of the Pascal Standard, the **var** clause follows the procedure declaration. Macintosh Pascal normally allows variables to be declared after procedures, but there's another problem here that will keep this program from running: *Second* is referenced in the procedure before it's defined in the main program. We could easily avoid the run-time error that will result by moving *Second's* declaration before the procedure, but then we'd be accessing *Second* as a global variable, opening up the possibility of a side effect. So our non Standard positioning of the *var* clause has a hidden benefit: it points out (with an error message) a global reference to the variable.

We'll leave it up to you (and your instructor) to decide whether this advantage offsets the possible lack of portability and readability that can result from taking advantage of Mac Pascal's relaxation of the order of declarations.

Mac Pascal
libraries

Macintosh Pascal has access to three major libraries of procedures, functions, predefined constants, and predefined types. We've used one of these – the QuickDraw1 library – extensively in this book. The two we haven't used are QuickDraw2, the advanced graphics library, and SANE, the Standard Apple Numerical Environment library. Unlike QuickDraw1, these libraries can't be accessed from a Macintosh Pascal program unless they're explicitly requested by the program with a **uses** clause. This clause comes immediately after the *program* statement.

```
program Show_Uses;  
  uses Sane, QuickDraw2;
```

(Note that **uses** is a reserved word in Macintosh Pascal.)

The SANE library is loaded with numerical functions. One useful function is *random*. This function has the same name as the Mac Pascal built-in function *random*. The SANE *random* function returns pseudo-random numbers in the range 1 to $2^{31}-1$. This is a much greater range than the built-in *random*. If SANE is declared in a **uses** clause then any call to *random* calls the SANE *random*. instead of the built-in *random*.

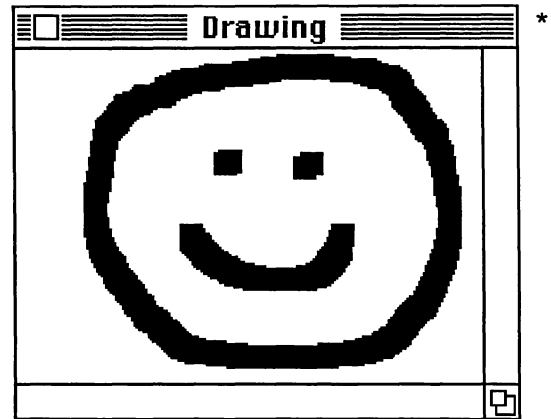
programming
the mouse

You've been using the mouse as an input device throughout this book. If you've wondered how to write programs that can read the mouse, you'll be happy to know that it's not difficult. The function *Button* and the procedure *GetMouse* provide the basic information your programs need to accept simple mouse inputs. *Button* is a boolean function that returns *true* when the mouse button is pressed and *false* when it is not. *Button* has no arguments. *GetMouse* returns the vertical and horizontal positions of the mouse in the Drawing window. The *Getmouse* procedure has two integer variable parameters which represent the horizontal and vertical positions. Here's a simple program that shows how *Button* and *GetMouse* can be used to create a sort of poor man's *MacPaint*.*

```

program Magic_Pen;
{A program for freehand picture drawing}
var
  x, y : integer;
begin
  PenSize(10, 10);
  while true do
    begin
      repeat
        until Button;
      GetMouse(x, y);
      MoveTo(x, y);
      while Button do
        begin
          GetMouse(x, y);
          LineTo(x, y)
        end
      end
    end
  end.

```



Once you've created a mousterpiece with *Magic Pen*, how can you save it to disk? You learned in the appendix how to do it with the keyboard, but wouldn't it be nice if the program had an option for saving built right in? The procedure *SaveDrawing*, when included in a program, saves the contents of the drawing window as a picture file that can be accessed by *MacPaint* and other applications. It has one **string** value parameter, the file name.

SaveDrawing(NewFileName); {Saves the drawing window as a file}

Unfortunately, there's no place to put *SaveDrawing* in this program, because it's built around an infinite loop. The only way to stop it with the Halt option in the Pause menu, and then there's no way to call the procedure. So it's back to the keyboard if we want to save our pictures.

Obviously, this program in its present form isn't going to inspire many budding Picassos. To be truly useful as an application, it would have to allow for pen size changes, special effects, and saving pictures, among other things. And these additional features should be accessible via pull-down menus in the Macintosh tradition.

Advanced Programming Resources



Unfortunately, we don't have the space here to show you how to develop this *Magic Pen* into a full-blown Macintosh application. It's a big jump from *Magic Pen* to *MacPaint*, and it's not a particularly easy jump in Macintosh Pascal. If you're interested in taking the leap, there's plenty of reading matter available to help you jump the right direction. Start by thoroughly studying the material on data structures in *Oh! Pascal!*; a

* This picture was drawn by Rosemary Johnson using *Magic_Pen*. All inquiries should be directed to the artist.

thorough understanding of data structures is essential for Macintosh programming. Then read the documentation that comes with Macintosh Pascal: *The Macintosh Pascal Reference Manual*, *The Macintosh Pascal Technical Appendix*, and the *InLine* documentation that comes on disk. *InLine* is a challenging but powerful procedure that allows you to access routines in the Macintosh Toolbox to create your own menus and such.

If that documentation isn't clear, consult one of the many commercial books on Macintosh programming. *Mastering the Macintosh Toolbox*, by David B. Peatroy and Datatech Publications, published by Osborne McGraw-Hill, is a particularly accessible book aimed specifically at Macintosh Pascal programmers.

If you're really serious about Macintosh programming, you'll probably want a copy of *Inside Macintosh*, the challenging but definitive reference on the subject.

Beyond Macintosh Pascal



When you're ready to graduate to writing full-scale applications programs, you may want to consider switching from the Macintosh Pascal interpreter to a compiler version of Pascal. The Mac Pascal interpreter is unbeatable for learning Pascal, and its friendly user interface makes it nearly ideal for designing and debugging Pascal programs. (We know one professional programmer who develops his programs using Mac Pascal and then converts them to whatever language the job requires.) But Macintosh Pascal suffers from a lack of speed and an awkward toolbox interface when compared with professional development tools. Fortunately, almost everything you've learned about Pascal in this book is directly applicable to most compiled Pascals.

Several excellent Pascal compilers are available for advanced applications programming on the Macintosh. One in particular stands out as a logical next step from Mac Pascal. Lightspeed Pascal, from Think Technologies (the same company that developed Macintosh Pascal) has most of the friendly features of the Macintosh Pascal user interface and is nearly 100% compatible with Macintosh Pascal.

Whichever language and implementation you choose for your work, you'll find that the tools and techniques you learned in *Oh! Pascal!* and this book will make your programming efforts more productive and enjoyable. Good programming practices work in *every* language.

Happy programming!

One Last Exercise

16-1. Rewrite *Magic_Pen* so that it automatically (1) displays the Drawing window at the maximum possible size on the screen; (2) draws two square "buttons" in the bottom of the Drawing window, one labeled "Save" and the other labeled "Quit"; and (3) recognizes mouse clicks inside those square buttons and reacts accordingly. Add buttons to change the pen size and pattern or to make other changes to the pen or the picture. Use your imagination. Finally, if you're interested in a real challenge, redo *Magic_Pen* using the Macintosh Toolbox routines to create pull-down menus, dialog boxes, and other official Macintosh enhancements.

I hear and I forget,

I see and I remember,

I do and I understand.

—Ancient Chinese Proverb

Appendix: A Hands-On Introduction

LEARNING TO PROGRAM is like learning to swim: reading about it just isn't enough. This appendix is designed to give you some hands-on experience so you can start writing and running programs as quickly as possible. It's divided into two sessions, each of which can be completed in an hour or so. The first session, designed to accompany Chapter 1, covers the basics of working with a Macintosh and the Mac Pascal environment. Session 2 introduces several tools and tricks you'll need for writing longer programs; it works best when used with Chapter 2. After you've finished these two sessions you'll be ready to strike out on your own.

Session 1: Meet Mac Pascal

If you haven't already, it's time to sit down in front of a Macintosh and make things happen. You'll need a Macintosh Pascal program disk, a blank 3.5" diskette, this book, and about an hour. If you're afraid of computers or worried about your ability to use them, let go of your anxieties and have fun!

(If you're using a Macintosh that's connected to a hard disk or other computers in a network, some of the details of operation may vary slightly from what's presented here. For example, many machines with hard disks will automatically boot – computerese for start up – without having a diskette inserted, provided the hard disk is turned on before the computer is. And you probably won't need a Pascal program disk at all if you're running off of a hard disk or a network. Your instructor or lab assistant should be able to provide you with details.)

Ready to go? Reach around the left side of the box, and you'll find the on/off switch about half-way down the back. Flip it on, and your Mac will wake up with a beep and a question mark. (If the screen is too bright or too dark for your taste, you can adjust the brightness with the knob located directly below the rainbow-colored Apple logo.)

That little screen picture (icon, in the vernacular) is saying, "Where's the disk?" Your Mac needs a disk to tell it what to do. And not just any disk will do; it needs a disk with system files on it. The Macintosh has lots of very intelligent circuitry built into its memory, but not enough to do anything useful. It needs to have the rest of its smarts loaded into memory from a disk each time it's turned on; that's what the system files are for.

So grab your Mac Pascal program disk. Before you insert it, check the little square hole in the corner next to the label. If you can see through the hole, the disk is locked (write-protected) so the computer can't add to or change any of the information stored on it. Since Macintosh Pascal occasionally needs to write to the disk in the course of doing its business, you'll need to slide the little plastic cover over the hole if it's not already there.

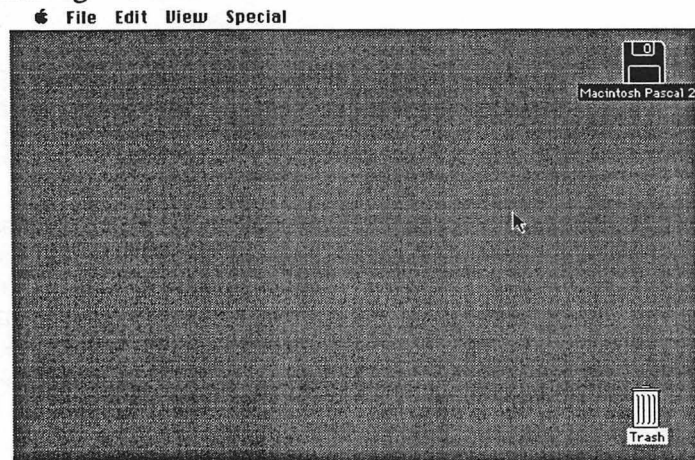
Now insert the disk (metal end first, label side up) in the slot under the screen. (Remember: the little metal door on the disk protects it from dangerous dust scratches and fingerprints, so don't open it yourself; the machine will do it when the time comes.) Gently push it all the way in, and the disk drive motor will start whirring or clinking as the disk spins. Eventually you'll see a little Mac on the screen smiling as if to say, "Thanks." (If you accidentally insert a disk that doesn't have the system



locked disks

files on it, the little Mac will stick its tongue out, the screen will turn black, and the disk drive will spit out the disk. No thanks. If that happens, or if the disk is rejected and an "X" appears in the tiny disk icon on the screen, turn the machine off for a second and try again, making sure you're using your Pascal disk, and not your blank one. If it happens again, have your disk checked by your lab assistant.)

After the system has finished booting, you should see a scene that looks something like this:



(If someone else has been using your diskette, the screen may not look exactly like this. Carry on anyway.)

the finder

You're staring at your electronic desktop (brought to you by an important system program called the Finder). This desktop is your work space when you're dealing with Macintosh, and you can treat it the way you treat any desktop. You can keep it neat and free of clutter, like it is now, or lay out your most important work in an organized way, or pile it high with notes, folders, disks, and miscellaneous garbage. But before you can do anything, you have to learn how to move things around. That's what the mouse is for.



To use the mouse, you'll need a clean space about a foot square on your desktop (your *actual* desktop, not the one on the screen). You'll be rolling your mouse around in that area whenever you use your Mac, so locate it in the position that's most comfortable for you. Orient the mouse so the cable points away from you and roll it around a little. Watch what happens to that little pointer (☞) on the screen. As the mouse goes, so goes the pointer. As you move the mouse around, remember to keep it facing the same direction. If it bumps into something or reaches the edge of the desk, pick it up and move it to a more roomy location; lifting the mouse doesn't move the pointer on the screen.

clicking

Once you get the hang of moving the pointer, move it so that it points to the little trash can. Now push and release the mouse button once with your finger (this is called clicking), and watch the trash icon change from

white to black. Try it again with the Pascal disk icon: point to the disk (the picture itself, not the "Pascal" label) and click the button once. The can is now back to normal, and the disk is now black. This inverted coloration (Apple calls it *highlighted*) indicates that you have selected the disk icon. When you use the Macintosh, you generally select something before you do something.

Point to the trash can and press the mouse button, but this time don't let go. Move the mouse to the left with the button held down, and watch an outline of the can move with the pointer. When you let go of the button, you're letting go of the can, too, and it suddenly relocates where you left the outline. In the Macintosh vernacular, this is called dragging an icon. Now drag the can back where you found it and let go. You can drag any icon around the screen this way.

dragging

desk
accessories

Your desk comes equipped with more than a trash can; there are several desk accessories hidden inside. Which desk accessories you have depends on what's been included in your system file, but you probably have a calculator, an alarm clock, a note pad, and a few others. These tools are available for your use almost anytime you're using the Mac -- even while you're writing a program. To find them, move the pointer to the Apple (🍏) up in the corner of the screen. Then press the button and watch what happens: the apple is highlighted, and a hidden menu suddenly appears under the apple.

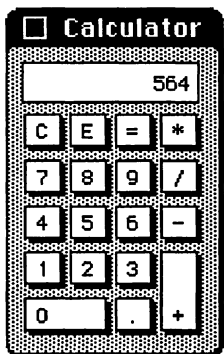
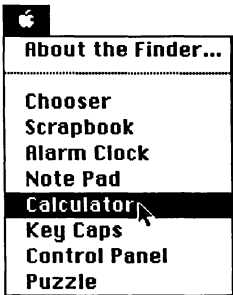
This is called a pull-down menu, because you'll pull (or drag) the pointer down to the item you want to select. Try it: without releasing the mouse button, move the arrow slowly down the list, watching each item become highlighted as you roll by. When you reach Calculator, release the button. (If you overshoot the calculator, just move the pointer back up to Calculator, or roll off the edge of the menu with the button down and try again. If you don't have a calculator in your list, pick something else.) The menu disappears, the disk whirs, and your personal calculator appears!

The calculator works just like any good \$5 calculator does, except that you can use the mouse instead of your fingers to punch the buttons. Try it: move the pointer to the 9 and click; move the pointer to the plus (+) key and click; then click 5 three times and click the equal sign (=). There's your answer.

There are four buttons that might not be obvious to you on the calculator: the asterisk (*), the slash (/), the E, and the C. Computerists use the asterisk to represent multiplication and the slash for division. The E stands for exponent, and is used to represent numbers in scientific notation (explained on page 11 of *Oh! Pascal!*). And C clears the calculator's screen and memory.

If you think punching those little buttons with the mouse is awkward, you're right. Fortunately, you have another choice: you can just type in the numbers and symbols from your Macintosh keyboard or numeric keypad. For example, if you type "C6*99=" on the keyboard while the calculator is selected, you'll clear the calculator and multiply 6 times 99.

You can drag the calculator around to different parts of the desk if it's ever blocking your view of something else. Just point anywhere on the



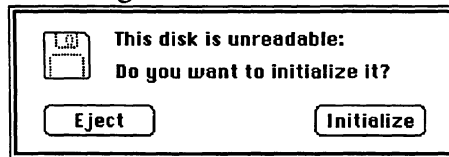
black bar at the top (except inside that little white square), press the mouse button, drag to a new location, and release. When you're through, click inside that little square in the upper-left corner, called a close box, to make the calculator disappear from the desktop.

ejecting
a diskette

Unless you've got a network, a hard disk, or some other unusual hardware configuration, you'll probably be working with two disks: the Pascal program disk that contains the interpreter and the system files, and a document disk for storing the programs that you write. You can use your blank disk for that purpose, but you'll have to get the Mac to recognize it first. If you've only got one disk drive, select (with a click) the Pascal disk icon and choose Eject (with a drag) from the File menu. The disk should pop out like a piece of toast, allowing you to insert your blank disk in the drive. If you have two drives, there's no need to eject the program disk; simply insert the blank disk in the extra drive.

If your disk has been used before in a Macintosh, it should show up on the screen right under the Pascal disk. When it does, select Erase Disk from the Special menu to turn it into an empty disk.

If you're using a brand new disk the computer will respond with a dialog box that looks something like this:

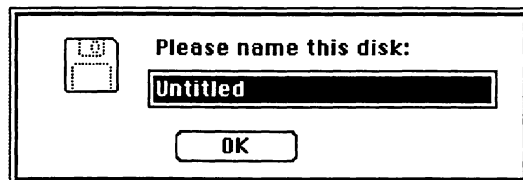


initializing
a diskette

Click inside the button that says "Initialize". Why? A new diskette contains no information at all -- just a magnetic surface waiting to be used. But your disk drive won't let you store any files on it until you initialize (or *format*) the disk. If you think of the diskette like a freshly-paved circular parking lot for programs, initializing the disk is a little like painting stripes and stall numbers on the pavement so the parking lot attendant will be able to easily park and unpark the files.

If the dialog box has buttons labeled "One-Sided" and "Two-Sided" instead of an "Initialize" you're working with a double-sided disk drives. If your disk is certified as double-sided, you can choose either of these buttons. The advantage of choosing double-sided is that, as you might suspect, you can store twice as much information on the disk. The disadvantage is that a double-sided disk can't be used at all in a machine that doesn't have a double-sided drive. If you choose single-sided, your disk will work in either kind of drive. If your lab has both kinds of machines, you might be safer going single.

Either way, after about a minute, Mac will say



Type a name for the disk – *George's Disk*, or whatever – and click OK. After a few seconds, a George's Disk icon will appear on your desktop right below the Pascal disk. If you're working with a single disk drive, Pascal is now shaded gray, indicating that it's not physically in the drive. Your new disk should be black, indicating that it's now the selected icon.

opening
a window

In general, a window is a tool for looking into something, and most of the icons on your desk have windows associated with them. Since your new disk is already selected, you can open its window by choosing (with a drag) Open from the File menu. When you select Open and release the button, the disk icon turns gray (or hollow) as a window zooms out.

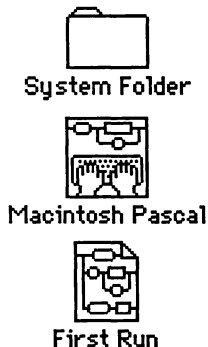
This window, labeled *George's Disk*, shows us what's stored on the disk: nothing. To close the disk window, either select Close from the File menu or click the close box in the upper left corner of the window. The window shrinks back into the disk icon, which is now back to black, still selected and waiting to be used. (If there are any other windows open on your desktop, close them now, too.)

double-clicking

When you're ready, move the pointer to the Pascal disk icon. You're going to open it now, but not with the pull-down File menu you used before. Instead, with the pointer pointing to the disk icon, double-click the mouse button. That's the shorthand way to open something; in this case, a window looking into the selected disk. If nothing changed on the screen, try again, making sure that the two clicks are close together. When you get it right, you should see the disk icon turn hollow as a window opens up, showing the contents of the disk.

The disk window should contain at least two icons:

- The System Folder, which contains the all-important system files. (You can open it and see, if you like.)
- The Macintosh Pascal Interpreter, which is the tool you'll be using throughout this book to create, modify, and run Pascal programs.



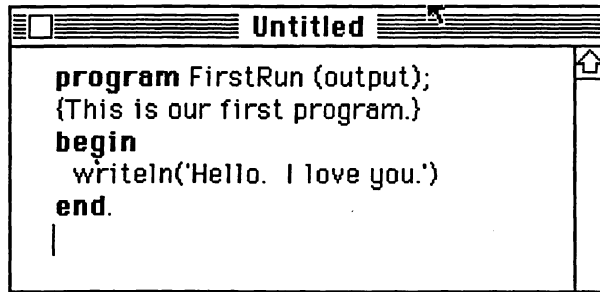
Your disk may also contain some Macintosh Pascal programs created with the interpreter. In order to create programs like these, you need to open the Mac Pascal interpreter. So do it: select the Macintosh Pascal icon, and open it using either the Open menu option or a double-click. (If you're working with a single disk drive, your blank disk will pop out and Mac will ask you to insert your Pascal disk. This kind of swapping can get tiresome, but it's necessary when you're using two disks in a single-drive system.) Eventually, you'll see the three windows of the Macintosh Pascal screen: the text window, the drawing window, and the program window.

The text and drawing windows are only useful if you've got a program to run, so let's look first at the program window (labeled "Untitled" because your program-to-be doesn't have a name yet.) It already contains the skeleton of a Pascal program highlighted in black. You're going to replace that program skeleton with FirstRun, the first program in Chapter 1 of *Oh! Pascal!* There are two ways of approaching this task:

1. Erase the skeleton and start typing FirstRun from scratch.
2. Replace those words and symbols that aren't in the FirstRun program (like the word "Untitled" after "program"), salvaging those parts that are (like the word "program").

Both methods yield the same results, but for the sake of simplicity, let's start with the first approach. The easiest way to get rid of text in the program window is to select it and then hit the `[Backspace]` key. (As usual, first select, then do.) But the prototype program is already highlighted, just as if you'd selected it, so you can remove it by simply hitting `[Backspace]`. Try it.

Your slate should now be blank, except for the flashing vertical bar in the corner. This insertion bar shows you where your typing will be inserted in the window. (The insertion bar is sometimes called a *cursor*, for CURrent poSition indicatOR.) Typing text on a computer is pretty much like using a typewriter, with one major difference: the computer is more forgiving. If you make a mistake, or if you change your mind, you can correct your program without retyping it from the beginning. Soon you'll learn about a number of powerful tools that can make short work of major editing jobs. But for this first program you'll only need the one you've already used: the `[Backspace]` key.



```

program FirstRun (output);
{This is our first program.}
begin
  writeln('Hello. I love you.')
end.

```

typing
the program

Try typing this program *exactly* as it's shown here, character for character, ending each line by pressing the `[Return]` key. Pascal is picky; it won't understand what you're telling it to do if you don't use language it can recognize, right down to the last semicolon. Special characters can be particularly troublesome; make sure to use apostrophes (') rather than double-quotes or backward accents to surround the greeting, and curly brackets {like these} to enclose the second line.


But don't take this to mean that you can't make mistakes in your typing; the worst that can happen is that you'll have to change something, and that's easy. If you hit a wrong key or three, just `[Backspace]` until the offending characters are gone and retype from that point. `[Backspace]` can even take you back to the previous line to correct errors.




Be sure to put spaces where they're shown between words and characters, but don't worry about indentation and typestyle details in the program listing. Each time you press `[Return]`, the Mac Pascal editor will scan the line you just typed, displaying all Pascal reserved words in **boldface**, indenting the line when appropriate, cleaning up messy details, and **outlining** obvious syntactical problems.

You might notice that the program listing doesn't exactly match the one in the textbook, even after the editor has prettied it up. From Pascal's

point of view, the stylistic differences are irrelevant. Line indentation and typestyle variations are included to make it easier for humans like us to read programs, and different humans have different ideas about what display styles make a program most readable. You don't need to make any decisions on these matters, because the designers of Mac Pascal have already decided for you by creating an editor that does the program formatting.

Program *FirstRun* is *Standard Pascal*, straight from Chapter 1 of *Oh! Pascal!* It should run just fine using the Macintosh Pascal interpreter, but it can be improved by taking advantage of some Mac Pascal extensions. For example, you can change your program name from *FirstRun* to *First Run*.

editing In order to edit your program name so it looks that way, you'll need to move the insertion bar. Move the pointer over the program window, if it's not already there. The pointer will turn into an *I-beam* () , allowing you to position it precisely between "First" and "Run". *Don't confuse the I-beam with the insertion bar.* The only thing the I-beam is good for is pointing to the place where you want the insertion bar. When you've found the right spot with the I-beam pointer, clicking the mouse instantly moves the insertion bar there. But until you click, the insertion point is still back where you left it. So click once and move the pointer out of the way so you can see the insertion bar in its new location. Now when you type an underscore character (the shifted hyphen on the top row of the keyboard), it goes after First, and Run shifts to the right to make room for it.

Next, take the unnecessary program parameter "output" out of the first line. You *could* click the pointer after the "t" in "output" and  each letter away, but there's an easier way. Move the pointer so it's positioned anywhere in the word "output", and double-click to select the entire word. Now whatever you type will *replace* that word. You want to replace it with nothing, and you learned how to do that when you wiped out the highlighted program skeleton earlier:  once. Notice how the left parenthesis and the semicolon slide to the left to fill the void, and the insertion bar is the only thing left where "output" used to be. A single  vaporizes everything that's highlighted, whether a single character or an entire program. Use it with care....

It's time to check your program for errors. Move the pointer to the Run menu and select Check. If the program is correct, nothing will change on the screen after some audible disk spinning. But if you've followed instructions, you're probably looking at a box like this one:

checking and
running



An invalid PROGRAM parameter list has been found.

The bug is telling you that your program has a bug — one of those errors that the editor couldn't find. It's not your fault; we led you here intentionally to show you how bugs are handled in Mac Pascal. The message in the box is supposed to help you figure out what the problem is. But as this example indicates, error messages aren't always as clear as

debugging
the program

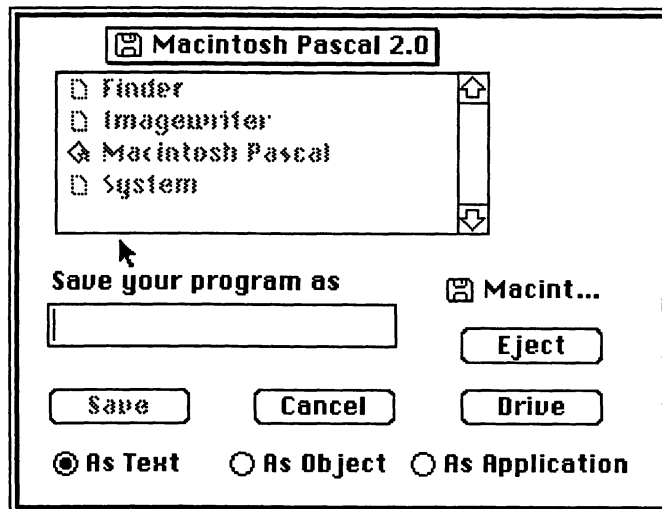
you'd like them to be. Click on the bug (or anywhere in the box) to acknowledge the message, and it'll go away, allowing you to look for more clues in the program listing. The thumbs-down in the margin tells you which line contains the error. The *program parameter list* referred to in the error message is the part inside the parentheses. There's nothing in there, so the problem must be that Pascal doesn't allow empty program parameter lists. You could solve the problem by putting back the "output" parameter, in conformance with standard Pascal. But a better alternative in Mac Pascal is to get rid of the parentheses altogether.

So move your pointer to the space right after the "n" in First_Run, and press the mouse button to position the pointer there. Now hold the button down while you *drag* the I-beam across both parens. Let go when you've reached the spot between the right parenthesis and the semicolon. (If you go too far, just back up or repeat the drag from the beginning.) The offending characters should be highlighted in black. Backspace them away, check your program again, and (hopefully) discover that there are no more errors. If the bug stays away, select Go from the Run menu to actually make your program execute. If all goes well, you'll see the love note in the text window.

saving
the program

Once you get your program running, try saving it to disk. To save this program, move the pointer to the File menu and select "Save As ...". (Notice that Save appears in the menu in a hard-to-read, light gray color. Don't try to adjust your set. That's just Mac's way of telling you that command isn't operative in this particular situation. Save can only be used after you've assigned a name to your program in a previous "Save As ..." operation. Dimmed commands in a menu *always* mean "no can do.")

After you've selected "Save As ..." from the menu, you'll see a dialog box asking you what name you'd like to assign to your saved program. The box will look more or less like this, depending on which version of the system you're using:



At the top of the box is a labeled icon representing your Pascal disk. If you've got two disk drives, click the Drive button in the dialog box to switch to the other disk. If you've got just one disk drive, you'll need to click the Eject button to pop the Pascal disk out so you can insert your blank disk in its place. Either way, the name of the disk at the top of the box should now be the name you gave your blank disk.

The flashing vertical bar tells you Mac is waiting for you to type a name for your soon-to-be-saved program. Most any name will do, but it makes the most sense to choose the name that appears in your program heading: First Run. (Macintosh file names can have embedded blanks, so you don't need to include an underscore character.) When you're done typing, you can give Mac the go ahead by pressing **Return** or by clicking the button that says "Save." If you're having second thoughts, you can click "Cancel" instead.

When the saving process is completed, the only change on the screen is your title on the program window. You saved a *copy* of the program onto a disk; the original is still in the computer's main memory, ready for more editing. So let's try adding a little bit of personalized dialog to the program. Move the pointer to the end of the *writeln* statement (after the final parenthesis) and click to position the insertion bar there. Now type a semicolon (;), press **Return** once, and type these lines, following each with a **Return**:

```
writeln("What's your name?");
readln(your_name);
writeln("Bye, ", your_name, "!")
```

Notice how each **Return** moves the last line of the program down. **Return** is like an invisible character that adds a new line to your program. The empty line before the **end** line is just the **Return** you typed. It's not necessary, so **Backspace** it away.

Now move the pointer so it's positioned after the closing brace at the end of the second line. Click to position the I-beam there, press **Return** to open a new line, and type

```
var your_name : string;
```

The Pascal editor will turn this into two lines for you whether you like it or not. But it's still the same old declaration, no matter how it's formatted.

Making changes to existing lines of your program is just about as easy as adding new ones. Move the pointer somewhere over the word "love" and double-click to select it. Now when you type "don't like", "love" vanishes. Now drag the I-beam through "don't lik" and type "lov" to make it look the way it did before.

As you can see, there are lots of ways to select text to change or delete:

selecting text

- You can drag the pointer from the beginning to the end of the text to be selected. If the text crosses several lines, you can beeline down to the end of the selection without crossing all of the characters on each line.
- You can drag the pointer backwards through text.
- You can double-click on a word to select only that word.
- You can double-click on a word and *then* drag the pointer to some other word; both of those words will be selected, as well as every word and character in between.
- You can triple-click anywhere on a line to select the entire line.
- You can triple-click on a line and drag to another line to select all lines between (and including) those two.

Try selecting various pieces of this short program using each of these techniques. Don't touch the keyboard, though, or you'll erase the selected text.

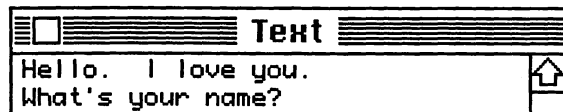
By now, the program looks quite a bit different from the one we originally typed and saved. In the interest of truth in advertising, you should change the name on the first line from `First_Run` to `Second_Run` and change the comment so it says "This is our second program." Use any of the editing techniques you've learned so far to make these changes.

When you're done making changes, it should look like this:

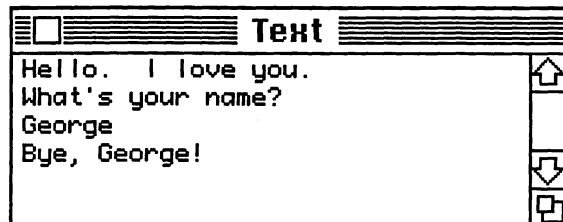
```
program Second_Run;
{This is our second program}
var
  your_name : string;
begin
  writeln('Hello. I love you. ');
  writeln('What's your name? ');
  readln('your_name ');
  writeln('Bye,', your_name, '!')
end.
```

Check it for syntax errors with **Check**, then run it with **Go**. The text window will say

running an
interactive program



and then wait for you to type your name into the **string** variable `your_name` via the `readln` statement. Press **Return** after you type your name to tell the program where the **string** stops. Your typing and the final program `writeln` should add two more lines to the text window, so it looks like this:



resaving the
program

When you're convinced it works, you probably should save it again. Since it's already been saved once, you *could* select Save from the File menu instead of "Save As ...". That tells Mac to save this new program right on top of the earlier saved version. The old one is wiped out because you can't have two things on a disk with the same name.

What if you want to keep the earlier version *and* the new one? All you need to do is choose a new name for the second copy (or put it on a different disk). The "Save As ..." option in the File menu brings up the dialog box again, allowing you to choose a new name and/or disk to save the picture, so that you don't wipe out the old one. (When this dialog box appears, your original name is highlighted in the name box, but anything you type will replace that name. Or you can position the vertical bar and edit the title. Or you can leave the name alone and use the buttons to select a different disk.) For this exercise, save the program as Second Run, so you'll have two saved programs with different names.

disaster
insurance

(When you're building larger programs, it's important to resave your work like this often. Since the computer's internal memory depends on a steady flow of power, the program you're editing can vanish anytime the machine is turned off or the power fails. But if you've saved a copy recently, you can always bring that copy in from the disk and start editing from that point. Frequent saves also provide you with a way of going back to an earlier version in case you accidentally delete or hopelessly mess up a large chunk of your program during an editing session. A good strategy for protecting yourself against program loss is to select Save every fifteen minutes or so when you're originally entering the program, and Save again right before you run it. Whenever you make major changes to an existing program, "Save As ..." a different name so you can fall back to the old version if something goes wrong. When you quit for the day, make a backup copy of the program using the techniques described later in this session.)

printing
the program

To make a hard (paper) copy of your program, select Print from the File menu. If your Macintosh is directly connected to an Imagewriter printer, you'll see a dialog box something like this:

The dialog box is titled "ImageWriter" with "v2.3" in the top right corner. It contains the following controls:

- Quality:** Three radio buttons: ☐ Best, ☒ Faster, ☐ Draft.
- Page Range:** ☒ All, ☐ From: [] To: [].
- Copies:** A text box containing the number "1".
- Paper Feed:** Two radio buttons: ☒ Automatic, ☐ Hand Feed.
- Buttons: "OK" and "Cancel" in the top right corner.

You can probably ignore most of the options in this box; you most likely want one copy of the whole program on automatic (pin-feed) paper. But you should know about the three quality choices:

- Best, which produces clean, easy-to-read copy *very* slowly.
- Faster, which prints text that looks almost as good as Best, but at about twice the speed.
- Draft, which prints readable, but ugly, text *much* faster than either of the other choices.

There's generally no reason to select Best unless you're trying to impress somebody. Faster is a good choice for printing easy-to-read copies of your program fairly quickly. But both Best and Faster make a temporary copy of the printout on your disk before routing it to the printer, so *they won't print anything at all if your Pascal disk is full, almost full or locked* (see the first page of this tutorial). Unfortunately, no error message appears under these circumstances; in fact, the computer may display a dialog box indicating that the program is being printed! If your program fails to print in Faster mode, and you know that the printer is turned on, it's connected to the computer, and the printer's Select light is on, try draft mode and remember to delete some old files from the disk later.

(If you're using some other kind of printer, or if you're sharing a printer with other Mac users via network, you'll probably need some additional instructions from your lab assistant to print your program.)

That's enough Pascal for a while. Select Quit from the File menu to return to the Macintosh desktop. If you're using two disk drives, you should see two disk icons on the screen. (If you're using just one, you'll need to Eject (File menu) your program disk and insert your document disk to make both appear on the screen.) Open both disk windows (with double-click or the Open menu option) if they aren't already.

moving windows
If you only see one window, one's probably hidden behind the other. Position the pointer so that it's pointing to the title of the visible window (or anywhere along the title bar) and drag. The whole window drags with you. Drag it so that it partially reveals the other window.

Notice the difference between the title bars and borders of the two windows: One window is highlighted with more details around the edge because it's the active window -- the last window selected (or the window containing the selected object). When two windows overlap like this, the active window is always "on top." If you want to do anything with a window, you must first make it the active window, if it's not already. To do that, simply click anywhere in the inactive window. Try it.

switching windows
Now make your document disk window active, and notice that it contains two icons representing the programs you saved. If these were important programs that took hours or days to create, you'd probably want to make backup copies immediately, so you wouldn't have to start over if something corrupted or destroyed the originals.

making backups
To see how to make backups, drag the pointer from a point above and to the left of the leftmost icon to a point below and to the right of the other. A dotted rectangle will form between the two points, showing you the area of the window you're selecting. When you release the mouse button, the two icons inside that rectangle turn black, indicating that you've selected both icons at once. This multiple-selection capability is handy when you want to do the same thing to several items. (The rectangle doesn't need to surround the icons completely to select them. If you need to select several icons that can't easily be surrounded or touched by a rectangle without including some unwanted icons, you can click each of them separately while holding down Shift.)

To duplicate these two programs, select Duplicate from the File menu. Since both icons are selected, you'll soon see two additional icons in the window. These two copies are identical to the original programs except for their names.

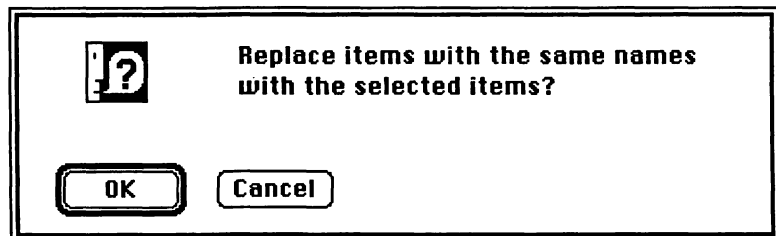
This is the quickest and easiest way to make backup copies of your programs, but it's not the safest. If you lose your disk, run it through the washer in your shirt pocket, or park it next to your magnetic flashlight in your backpack, you may lose everything on it, including backup copies.

copying
between disks

It's much safer (if more expensive) to put your backup copies on a second disk and keep that disk in a safe place. To copy a program from one disk to another, simply drag its icon from the source disk window to the window of the destination disk. Try it: select the First Run and Second Run icon(s) by dragging a rectangle around them in and drag them onto the Pascal disk.

When the operation is complete, you'll see a First Run and Second Run in both disk windows. Your originals are still on your document disk; you just made copies on the other disk. (Whenever you move an icon *from one disk to another*, the original remains undisturbed.) There are two situations where this technique won't work:

1. Mac won't copy the files if it can't find enough space on the destination disk. (Solution: delete or move some files from the disk, or use another disk. You may run into this problem when copying these files onto your Pascal disk. If so, don't try to delete the Pascal or System files; they're important to keep. If you don't have another disk, just carry on.)
2. If a file already exists on the destination disk with the same name as one you're copying, Mac won't proceed with the copy without a warning query and a chance to cancel your request:



(Solution: Since two files on a disk can't have the same name, you'll have to change the name of one or the other before making the transfer, or else say goodbye to one. To change a file name, just click the icon and type a new name.)

You could, in fact, move everything on the disk that way to make a backup copy of the entire disk, but there's an easier way: just drag the icon representing the disk to be copied onto the icon representing the soon-to-be backup disk.

(If you're working with your own hard-earned Pascal disk, you may want to copy that disk to protect your investment against accidental whatever. Earlier versions of the Macintosh Pascal interpreter were *copy-*

protected so they couldn't be duplicated in this way. Copy-protection has been removed from the software to make it easier for you to make extra copies of the interpreter on different disks in case something happens to your original. Mac Pascal is still *copyrighted*, though, so those extra copies should *not* be shared with friends. Programming is hard work, and complex software like Macintosh Pascal is terribly difficult and expensive to develop. When you give away copies of that, or any other, copyrighted software, you're *stealing* from the people who wrote it – *and* violating federal law.)

The Golden Rule of Sharing Software

Bootleg unto others as you would have them bootleg unto you.

trashing
programs

There's no point in saving both of these tiny programs on both disks, so drag Copy of First Run and Copy of Second Run out of the document disk window and into the trash. When your pointer reaches the can, the can will turn black, indicating that the selected items are now inside it. Let go of the button, and the copies disappear from the window. They should be in the trash. Open the can to check, using either of the two can-opener techniques you've learned. If you did everything correctly, you should see both icons in the Trash window.



Trash

If you're having second thoughts about throwing them away, you can drag them out of the trash window and back to their original disk windows. (The programs are still on the disks, even though the icons disappeared from the window. This feature can come in very handy – even the best of us occasionally have to dig through our own garbage.) Or, if you're ready to *really* dispose of them, you can move the pointer up to the Special menu and drag it down to "Empty Trash." When you do, the program disappears from the trash window, and you're given back the space it occupied on the disk. (Normally, it's not necessary to empty your own trash like this unless you're in a hurry to get your disk space back. The machine will take care of it for you eventually, the next time you open Pascal or when you remove your disk and quit.)

Shut Down

If you just turn off your machine when you're through, you won't be able to take your disks with you. What's worse, you won't give the machine a chance to take care of last minute diskkeeping details. So each time you're ready to quit, pull down the Special menu and select Shut Down. (If you're using a version of the system that doesn't include the Shut Down command, just select and eject both disks instead, and ask your lab assistant about updating your system file.) Mac will eject your disk, forget everything you did this session, beep, and ask if you want to start over by bringing back the "Where's the disk?" icon.



If nobody's waiting to use the machine, turn it off and call it a day. Next time you'll learn some advanced editing and debugging techniques to make your programming easier and you'll see how to write programs that draw pictures in the graphics window.

Session 2: Tools and Tricks

For this session you'll need your Mac Pascal disk and the document disk you created in your first hands-on session. This time we'll cover several new topics: manipulating windows, editing longer programs, and organ-izing your files and disks. As usual, this tutorial will work best if you *do it* rather than read it. If some of the more advanced material seems over-whelming, let it go and remember that it's here when you need it.

To get started, turn on your Mac and boot it with your familiar Pascal program disk. Then insert your document disk in the other drive (if you have one, or in the main drive, after ejecting Pascal, if you don't). If you left your document disk window open the last time you ejected the disk, it should automatically open when you start up this time. If it doesn't, open it (with a double-click or the Open command in File). The Second Run icon should be there, where you left it.

Open that icon just like you opened the disk; Mac will recognize it as a Pascal program, open Pascal, and bring Second Run into the program window, ready to edit or run. You're going to use Second Run as the raw material for building a longer, more interesting program. But first, a word from the Instant window.

instant window

Select Instant from the Windows menu. The Instant window will appear, waiting for you to type a Pascal command. Type

```
writeln ('Hello again.')
```

That phrase should appear in the text window as soon as you click "Do it." No check, no run; just instant results. Backspace that *writeln* statement away and try another. Type

HideAll

```
HideAll
```

Now push "Do it." Poof! *HideAll* is a powerful Macintosh Pascal procedure for automatically closing all the windows on the screen. You could, of course, close these windows manually with their close boxes, but *HideAll* works anywhere inside a program, allowing you to clear the screen automatically.

Select Instant from the Windows menu again, and replace *HideAll* with

ShowText and
ShowDrawing

```
ShowText;  
ShowDrawing
```

Now you can resurrect the Text and Drawing windows by pressing "Do it." Bring back the program window manually by selecting Second Run from the Windows menu. Everything should look normal now.

Now let's change Second Run from a love note to a letter to Mom. Start by changing the program name; double-click on "Second_Run" to select it and type:

```
Dear_Mom
```

Appendix: A Hands-On Introduction

Next, change the comment; triple-click on the comment line and type

```
{This program writes a letter.}
```

Now select the line where *Your_name* is declared and replace it with this declaration:

```
How_much: integer;
```

On to the program body. Insert the cursor after **begin**, press **Return** once, and type these procedure calls:

```
HideAll;  
ShowText;
```

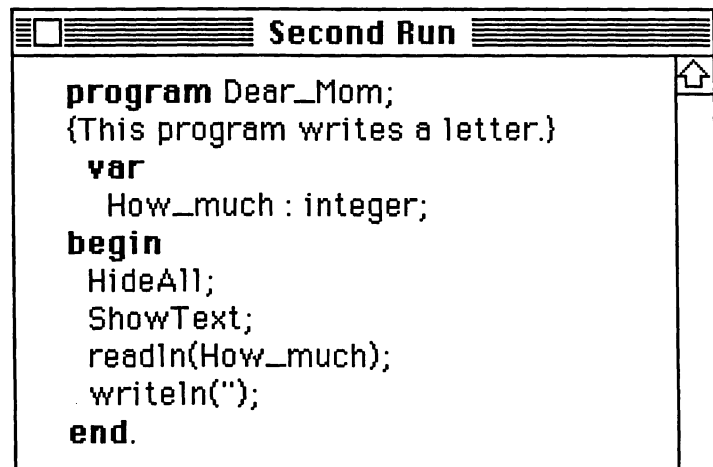
Now select the first two *writeln* statements and **Backspace** them away; you won't need them in this program. Change the variable name in the *readln* statement so it looks like this:

```
readln (How_much);
```

Now take the remaining *writeln* statement and erase (select/**Backspace**) everything between the first and last quote marks, so all that remains is a generic *writeln*:

```
writeln (");
```

Checkpoint – your program-in-transition should look like this. (The window is still called "Second Run" because you haven't resaved the program with a new name yet.)

A screenshot of a Pascal program window titled "Second Run". The window has a standard Mac OS-style title bar with a close button (red square) on the left and a scroll bar on the right. The program code is displayed in a monospaced font:



```
program Dear_Mom;  
{This program writes a letter.}  
var  
    How_much : integer;  
begin  
    HideAll;  
    ShowText;  
    readln(How_much);  
    writeln(");  
end.
```

using the
clipboard

You'll need one generic *writeln* for each line of your letter, with the text of that line inserted between the quotes. There's a tool for mass-producing these *writeln*s hidden up in the Edit menu. Select the entire line to be replicated with a triple-click; then select Copy from the Edit menu. You just made a copy of that line, but you can't see it, because it's hidden in the invisible clipboard. You can paste that copy anywhere you want in your program by putting the insertion bar at the desired spot and selecting Paste from the Edit menu. Put the insertion bar at the beginning of *end* and select Paste. Then select Paste again. Each time you paste, you'll see a new *writeln* added to your program.


keyboard
shortcuts

Edit	
Cut	⌘H
Copy	⌘C
Paste	⌘V
Clear	
Select All	⌘A

You don't need to go to the menu each time you paste, though. Try holding down  – called the *command* key – and pressing the "v" key. That keyboard shortcut has the same result as selecting Paste from the menu. As it turns out, many of the commands in the pull-down menus can be activated by using the command key with some other key. The keyboard commands are usually shown in the menus to the right of their mouse equivalents. Use -v to paste ten more *writeln*s into your program. Then, line by line, insert text between the quote marks so your letter reads like this:

```
writeln ('Dear Mom,');
writeln;
writeln ('Due to circumstances');
writeln ('beyond my control,');
writeln ('I'm broke!');
writeln ('My rent is due. ');
writeln ('My car payment is due. ');
writeln ('Please send me');
writeln ('$ ', How_much:3, '. ');
writeln ('I hope you're fine. ');
writeln;
writeln ('Love, ');
writeln ('Skip');
```

(Notice as you're editing that three of these lines don't quite follow the pattern of the others; two are empty *writeln*s for producing blank lines, and one includes mention of the variable *How_much* between quoted punctuations.)

The copy command can be used to copy anything you can select, from a single character to a whole program. Try selecting three lines starting with the 'I'm broke!' line (triple-click on that line and drag down), then press -c to copy those four lines into the clipboard. (This action wipes out the previous contents of the clipboard, which can only handle one chunk of text at a time.) Now paste those three copied lines into a spot right before the 'Please send...' line, and change the second 'I'm broke' to 'I repeat:', so that section of the program now reads

Appendix: A Hands-On Introduction


```
writeln ('I'm broke!');  
writeln ('My rent is due.');
```

```
writeln ('My car payment is due.');
```

```
writeln ('I repeat:');
```

```
writeln ('My rent is due.');
```

```
writeln ('My car payment is due.');
```

To make the letter a bit less heavy-handed, you might prefer to have it start, rather than end, with 'I hope you're fine.' It's easy to move that line up; just select it (triple-click), choose Cut from Edit (or use -x), move the insertion bar right before the 'Due to circumstances' line, and paste. Cut, like Copy, puts the selected text in the clipboard; the only difference is that it deletes the selection at the same time. (Cut, Copy, and Paste can also be used to transfer text between different programs, and even between different Macintosh applications. The clipboard always remembers what you've put there until you clip something new there or shut the computer down.)

It's taken so long to write the letter that your payments have lapsed, so let's change all of those "My ... payments are due" to sentences to read "My ... payments are overdue." We could edit those lines using the techniques we've already covered, but there's a way to do all of those changes at once. Select "What to Find..." from the Search menu, and you'll see this dialog box:

searching
and replacing

Search for	<input type="text"/>	
Replace with	<input type="text"/>	
<input checked="" type="radio"/> Separate Words	<input checked="" type="radio"/> Case Is Irrelevant	<input type="button" value="OK"/>
<input type="radio"/> All Occurrences	<input type="radio"/> Cases Must Match	<input type="button" value="Cancel"/>

Type "due" in the "Search for" box, press Tab to move to the "Replace with" box, type "overdue", and click OK. Toward the bottom of the window, notice that "Separate Words" and "Case is Irrelevant" are bulleted, indicating that those are the operative rules, unless you specify otherwise. "Separate Words" means that *due* must be in a word by itself to be changed to *overdue*; *dues* or *residue* wouldn't be recognized or changed. "Case is Irrelevant" says that the capitalization is not a factor in the search. Don't change anything; just click OK. When the box goes away, select Everywhere from the same menu. Another dialog box appears, giving you a last chance to change your mind:

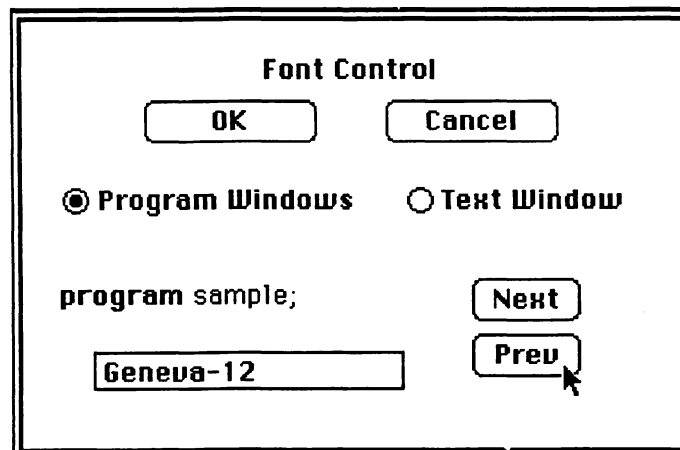
Change "due" to "overdue" everywhere in the active window?	
<input checked="" type="button" value="Yes"/>	<input type="button" value="No"/>

Click the circled Yes button, or just press **Return** to indicate that the circled option is your choice. When you've got a 500-line program full of misspelled identifiers, this search-and-replace tool can save lots of time. But it can also become search-and-destroy if you don't think through all of the implications of your request before you press OK. For example, you may not have wanted to change 'Due to circumstances' to 'overdue to circumstances' in the first line, but you just did. (We led you astray again; case *was* relevant.) It's easy to correct that oversight – use any of the editing tricks you've learned. But imagine having to correct it 75 times! The Replace option in the same menu is a safer tool for cautious changes; it searches out the *next* occurrence of the word in question and replaces it with the substitute word. Find is like Replace, except that it doesn't make any substitutions. It's useful for finding needles in your larger haystacks.

By now you can't see the whole program at once, because it overflows the window. Most of the programs you'll be writing will be much longer than this one, so it's important for you to learn how to scroll the window up and down through the program using the *scroll bar* along the window's right edge. Move your pointer to the arrow that points up (⬆), press the button, and hold it down. Your program will start scrolling through the window, with the white scroll box moving up the scroll bar like a tiny elevator to show you the position of your window relative to the overall listing. Now scroll back down with the down-arrow (⬇).

It's also possible to drag the scroll box to a different position on the scroll bar without using the arrows, or move it one window's worth with a click on the gray part of the bar between the scroll box and the arrow that's pointing the direction you want to go. Experiment. Most Macintosh windows have scroll bars that work just like this one.

Scrolling is handy, but it's often inconvenient if you're trying to examine many lines of code at the same time. Since you can't put more lines in by making the window taller, the only alternative is to make the characters shorter, using the Font Control option from the Windows menu.



When you select this option, a dialog box appears, allowing you to indicate your font (typestyle) preference for characters that appear in the program and text windows. The window initially indicates that your program window displays characters in the 12-point Geneva font. By clicking the Prev button, you can select the next-smallest available size of the Geneva font – probably 9-point. Browse through other fonts available in the system file by clicking the Next and Prev buttons. Some fonts, like Times, Helvetica, and Courier, are designed to look best when printed on Apple's LaserWriter printer. Others, like Chicago and Geneva, look best on the Mac screen or the Imagewriter. If you select a mono-spaced font like Monaco or Courier, all characters in the window will take up the same horizontal width, as they do on typewriters and most computer screens.

Since many Pascal programs are designed to produce columns of facts or figures, and columns are difficult to align using proportionately-spaced fonts like Geneva, Mac Pascal's text window is set up to display output in Monaco-9. But you can change this, too, by clicking the Text Window button and choosing a different font for that window.* When you're happy with your choices, click OK and take another look at your program. If all is well, you should be looking at something like this:

```
program Dear_Mom;  
{This program writes a letter.}  
var  
    How_much : integer;  
begin  
    HideAll;  
    ShowText;  
    readln(How_much);  
    writeln('Dear Mom,');  
    writeln;  
    writeln('I hope you're fine.');
```

```
    writeln('Due to circumstances');  
    writeln('beyond my control,');  
    writeln('I'm broke!');  
    writeln('My rent is overdue!');  
    writeln('My car payment is overdue!');  
    writeln('I repeat:');  
    writeln('My rent is overdue!');  
    writeln('My car payment is overdue!');  
    writeln('Please send me');  
    writeln('$', How_much : 3, '.');
```

```
    writeln;  
    writeln('Love,');  
    writeln('Skip');  
end.
```

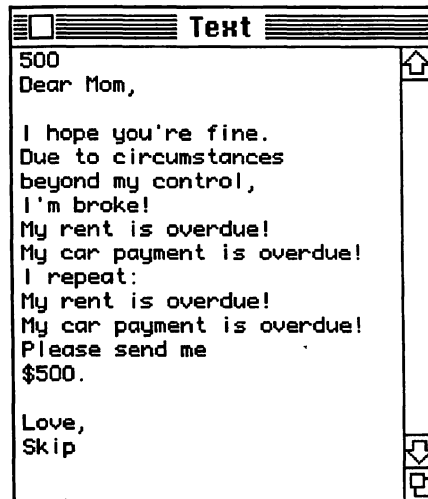
* Hundreds of fonts are available, but you'll only see the ones that have been included in your system file. If you want one that's not there, you'll need to install it using the Font/DA Mover utility, the same program that's used to add or remove desk accessories.

If your program looks OK, check it with the Check option in the Run menu. When it passes, select Go to see if it displays the letter correctly in the text window.

It won't, of course, until you give it the input it's waiting for: the number called *How much*. Type in any whole number you like, press Return, and watch the letter appear.

resizing
a window

Oops! Even if your program works perfectly, it's going to be hard to read the output in that tiny text window. You could scroll backward and sideways using the scroll bars to see the rest of it, but there's a better way. Grab the little *size box* in the lower right corner of the window and drag it straight down. The window stretches toward the bottom of the screen, making room for the missing lines of your letter. You can also use the size box to make the window wider, but you'll have to move the window to the left first by dragging the title bar. Play with the size box until you get a nice fit.



printing
output

It's not practical to mail the computer screen to Mom, so you're going to need to get a paper copy of your output. Select Preferences from the Windows menu, click the box that says "Output also to the Printer", and click OK. Now when you run your program and type in an amount, the letter should appear in the text window and on paper (if the printer is ready).

When you print your letter, you'll notice that it's got an unwanted line at the top: the *How much* figure you typed in. Later in this book you'll learn how to assign the printer to a different file identifier so you can keep your printed output separate from interactive input. Until then, you and Mom will just have to be patient.

Save this program as "Dear Mom" on your document disk. There'll be a "Dear Mom" icon waiting for you when you return to the desktop, but you're done with this program for now. So click anywhere in the program window to activate it and select Close from the File menu to clear it from the screen and memory.

A Graphic Diversion

Before we exit to the finder, let's take a scenic side trip with a graphics program. There's really nothing different about running graphics programs, except that they're often more difficult to understand because they contain so many procedures with numeric arguments. Macintosh Pascal includes two useful tools for slowing the action down in a program so you can watch the play-by-play at your own pace.

If you don't have the *Oh! Mac Pascal!* disk, select New from the File menu and type in one of the ready-to-run graphics programs from this chapter. If you have access to the disk, select Open, rather than New, and use the Drive and Eject buttons in the dialog box to make room for it in one of your drives. If folders rather than programs appear in the dialog box file list, double-click on the folder labeled Chapter 2 to find this chapter's programs. (Folders are explained at the end of this session.)

stepping through
a program

When you've got a program on the screen, select Step-Step instead of Go from the Run menu. This option slows down the program's execution enough so that a tiny hand can point to each statement as it executes. The action is still fast, but statements don't all run together this way.

An even better option for observing the results of individual statements is Step. Each time you select Step (or type `⌘-S`), exactly one statement is executed. The pointing hand shows you where you are in the program, so you can compare output with program every step of the way while *you* control the pace. Both Step and Step-Step work equally well with text and graphics programs; they're powerful tools for understanding and debugging your programs.

(You've probably noticed by now that a Pause menu appears whenever you run a program. It hides a Halt option that can be used to bring things to a stop for any reason. Depending on what you're doing, you may be able to restart things where they left off later. Experiment.)

screen
snapshots

Graphics programs are fun to watch on the screen, but there comes a time when you need to make a paper copy of your output. The easiest way to do that on the Macintosh is to take a snapshot of the Drawing window. Click on that window, turn on your printer, and press the `⌘`, `Shift`, and 4 keys all at once. An exact copy of the window should come out of your printer. This trick works with any active window. If you want a snapshot of the whole screen – a *screen dump* –, press the `Caps Lock` key before you press the other three. (You can also capture any screen as a picture file that can be read by many Macintosh graphics applications programs, if there's room on your Pascal disk, by pressing `⌘-Shift-3`).

Back to
the Finder

When you're ready, select Quit from File to return to the desktop. Your "Dear Mom" icon should be waiting there for you in the document window. In the File menu you'll find a way to Get Info about that, or any other, selected item. Select "Dear Mom" with a click; then select Get Info from the File menu. A new window should pop out of Dear Mom that tells you more about that item than you probably care to know. Most of this window is filled with information from Mac to you, but not all of it.

get info

At the bottom is an empty comment box waiting to be filled with up to three lines of text describing First Run. Type whatever you want, but

remember that the purpose of this text is to remind you (or somebody else) what's in Dear Mom. (You'll be amazed how quickly you'll forget.) As you're typing, you can use the editing techniques that you learned with Pascal.

Above the comment box is a much smaller box labeled Locked. There's no room for typing in this box, but you can "x" it with a click. An "x" in the box indicates that Dear Mom is locked, and can't be thrown away until the "x" is removed with another click in the box. (Leave it there for now.) This doesn't protect a document from being replaced by another document with the same name, or from being destroyed by a teething spaniel, but it does help insure that you won't absent-mindedly erase an important document. Like the comment box, the lock is only helpful if you choose to use it.

view by....

You don't need to open the Info window to find out how big a file is or when it was created; all you need to do is change the view through the disk window by selecting an alternative from the View menu. When you pull this menu down, you'll see a check mark in front of "by Icon", indicating that that's the way you're currently seeing the active window. Select "by Name", and the icons in the window are replaced by mini-icons followed by textual descriptions of the corresponding items on the disk, arranged alphabetically by name. There's some useful information here that you couldn't see in the icons: the amount of space each item occupies on the disk and the date the item was last changed. There's even a padlock in front of Dear Mom to remind you that it's still locked. You can select and open items from this list the same way you did icons. In fact, you can do just about anything with them that you can do with big icons, except scatter them randomly around in the window. View by Date, by Size, and by Kind give you the same list arranged in different orders.

As you might expect, you can also open folders to see what's inside. Try double-clicking the System Folder. You'll find several icons, including System, the brains behind the operation, and Finder, the desktop organizer. The Imagewriter file allows your Mac to talk to the printer. These system files need to be available for your Macintosh to work, but you don't need to look at them. The system folder hides the system files from you so they don't clutter up your disk window. The only time you ever have to open that folder is when you need to tinker with some part of the system (for example, when you want to replace one of these system files with an upgraded version of the same file.)

What the system does, you can do, too. Instead of having a window cluttered with icons representing several month's work, you can neatly stuff your work into personally labeled folders.

folders

Create a folder by selecting "New Folder" from the File menu. "Empty Folder" isn't a very useful name for a folder, so the next step is to rename that folder. The folder is already selected, so all you need to do is type a new name – "Silly Programs" – to replace the old one. If you make an error, correct it with the usual editing techniques. (You can change the names of program and disk icons the same way.)

Once you've created and named your folder, drag it anywhere you want in the window. Then drag Dear Mom over the folder until it turns black and release. Dear Mom is now in the new folder. Repeat the process with the other programs you've created. If you open the folder's window, you'll find those icons.

hierarchical file system You can put anything you want (except disks and the trash) in folders – even other folders. The folder is a powerful organizational aid when you're working with the Finder. But you'll do some file operations – saving and opening – via dialog boxes from inside Pascal. Folders may or may not make a difference there, depending on which version of the system you're using.

When you select Open or "Save as..." from inside Pascal, the original Macintosh File System (MFS) will completely ignore folders and display in a dialog box a list of every file on the selected disk. The newer Hierarchical File System (HFS) includes folders in the list along with files, but it won't show you anything that's hidden inside a folder – unless you open that folder. When you open a folder, that folder's icon replaces the disk at the top of the box, and only those objects stored in that folder are visible in the dialog box list. You can always click the disk icon to return to the disk-level of your hierarchy of folders and files if you need to. It's possible to lose files in a large hierarchy of folders if they aren't filed logically, so exercise care and common sense when you're naming and organizing of folders.

startup disk Before you Shut Down for the day, try one last trick: drag your document disk icon into the trash can. You're not trashing the disk, you're just telling Mac to forget all about it for now. That's a handy trick for keeping your desktop clean, but it's also helpful if you're ever greeted by an "insufficient memory" message. The finder uses memory to keep track of all of the folders and files stored on each disk, whether those disks are currently in the drives or not, unless you give it permission to forget. (There's one situation where this trick won't work: you can't discard the *startup disk*. The startup disk is the one that contains the system files that Mac is currently using as its master instructions. It's usually the one you used to boot your Mac when you turned it on.)

That's enough for now. There's plenty more to try, but you're ready now to do your own exploring. When you read about something new in Mac Pascal – whether it's a language feature like the **for** loop or a programmer's tool like the Observe window – try it out. The Macintosh Pascal environment is designed so you can learn by experimenting. Do it!

Index

- advanced programming tools 45
- animation 30
- array** 22, 46
- bar graphing 29
- bomb 6
- boolean 3, 29
- boolean input/output 29
- built-in functions 8
- Button function 70
- calculating scaling factors 49
- calculator 75
- Cartesian coordinates 9
- case** 24
- char 3
- CharWidth 14
- Check 5, 79
- clicking 74
- clipboard 87
- close 57, 58-63
- comments 2
- compiler 2
- concat 34
- Copy menu option 17, 89
- copy function 35, 51
- copying files 85
- Cut 90
- DateTimeRec 53
- declaration order 68
- delete 35
- desk accessories 75
- dialog box 40, 41, 83
- direct access files 58
- disk
 - disk ejection 76
 - disk initialization 76
 - locked disk 73
 - startup disk 96
- double type 3
- double clicking 77
- dragging 75
- DrawChar 14
- drawing shapes 11
- Drawing window 1, 14
- DrawString 12, 14-15, 26
- enumerated I/O 43
- enumerated types 43
- EraseArc 12
- EraseOval 12, 15, 31
- EraseRect 12
- EraseRoundRect 12
- extended type 3
- external files 37
- File menu 6
- files
 - copying files 85
 - direct access files 58
 - external files 37
 - hierarchical file system 96
 - sequential files 59
 - file window 37, 38
- Filepos 60, 64
- FillArc 12
- FillOval 12, 15
- FillRect 12, 27
- FillRoundRect 12
- finder 74
- folders 95
- font control 91
- for** 21
- FrameArc 12, 15
- FrameOval 12, 15, 21,30
- FrameRect 11-13, 26, 54
- FrameRoundRect 12
- get 62-63
- Get Info 94
- GetMouse 70
- GetTime 53
- Go 18
- Go-Go 18
- HideAll 55, 87
- HidePen 10
- hierarchical file system 96
- histogram 48
- icon 73
- identifiers 2
- include 35
- INF 8
- initializing variables 7
- insert 35
- insertion bar 78, 79
- Instant window 8, 87
- integer 3
- interpreter 2
- InvertArc 12
- InvertOval 12
- InvertRect 12

Index

InvertRoundRect 12
justifying string output 34
key values 61
labeling graphics 12
language extensions 1
length 34, 51
Lightspeed Pascal 71
Line 10, 15
LineTo 10, 15, 26
locked disk 73
longint 3
Mac Pascal libraries
MAXINT 7
MAXLONGINT 7
menus
 Edit menu 89
 File menu 6
 Run menu 5, 18, 79
Move 10, 15
MoveTo 10, 15, 26
moving windows 84
NewFileName 40, 41, 56-64, 70
Note 46, 47
Observe window 17
OldFileName 40, 41, 45, 56-64
omit 35, 37
open 59, 60
Note 46, 47
opening a window 77
ordinal types 3
otherwise 24
page procedure 68
PaintArc 12
PaintOval 12
PaintRect 12, 50
PaintRoundRect 10, 15
parameters 18
Paste 17, 89
PenNormal 10
PenPat 10
PenSize 10, 15
PI 7
pixel 9
plotting bars 50
plotting frequencies 49
pointer operator 67
pointer function 67
pos 34
predefined record types 53
Preferences 39, 57
Print 6
printing 83, 93
program heading 2
program window 1
pseudocode 22, 61
put 59
QuickDraw Graphics 8
QuickDraw1 69
QuickDraw2 69
Random 8, 24, 25, 69
random number generation 24
read 3
readln 3
ReadString 13
real 3
Rect 54
relational operators 33
repeat 30
reset 40, 41, 56
resizing a window 93
rewrite 39, 56
Run menu 5, 18, 79
run-time bugs 5
SANE 69
Save 83
Save As 80, 83
SaveDrawing 70
saving a program 80
scope 18
screen snapshots 94
scrolling 91
search and replace 36, 90
seek 60
selecting text 82
sequential files 59
set cardinality 66
SetDrawingRect 55
SetRect 54, 55
SetTextRect 55
ShowDrawing 9, 15, 21, 28, 31, 55, 87
ShowPen 10
ShowText 9, 23, 28, 43, 51, 87
Shut Down 86
spurious bugs 5
standard types 3
startup disk 96

- Step 18
- Step-Step 18
- Stops In 18
- Stops Out 20
- string 3, 33-36, 56
- string arrays 50
- string functions 17
- string subscripts 50
- StringOf 13,14, 26
- StringWidth 13,14, 26
- switching windows 84
- synthesized music 46
- system bugs 6
- System folder 77
- text processing 33
- Text window 1
- TextFace 14, 15
- TextFont 14
- TextSize 14, 15, 26
- translation bugs 4
- Trash 86
- until 30
- user interface 1, 40
- uses 69
- view by 94, 95
- while 30
- windows
 - Drawing window 1, 14
 - file window 37, 38
 - Instant window 8, 87
 - moving windows 84
 - Observe window 17
 - opening a window 77
 - program window 1
 - resizing a window 93
 - switching windows 84
 - Text window 1
- write 7
- WriteDraw 12, 14
- writeln 3-5
- write protected 73



Common error messages and suggested solutions

This doesn't make sense.

Something is out of place or an inappropriate character is on the indicated line. Examine that line carefully.

A semicolon (;) is required on this line or above but one has not been found.

If you don't find the missing semicolon in the immediate vicinity, then search backward from this point looking for a missing semicolon or end.

Either a semicolon (;) or an END is expected following the previous statement, but neither has been found.

Find the missing part; it should be close to the indicated line.

This does not make sense as a statement.

Often a missing parenthesis (unbalanced parentheses) or an operator used in the wrong place.

This formal parameter type or result type should be a named type or "STRING", but is not.

String[n] is not an acceptable type for a formal parameter. In general, identifiers whose types have not been declared in a type statement may not be used as formal parameters.

An incompatibility between types has been found.

You have tried to assign a value, variable, or expression of one type to a variable of another type. This error also may occur when the types of formal and actual parameters do not match.

Too few/many parameters have been used in a call to a procedure or function.

Check to see that the number of parameters in the function or procedure call match the number of formal parameters. You may have to refer to the *Macintosh Pascal Reference Manual* or *Technical Appendix* to determine the correct number and type of parameters for the built-in procedures and functions.

A STRING value is too long for its intended use.

You have tried to read or assign more characters to a string variable than it was declared to hold. Most frequently this error occurs when reading from a text file into a string variable and the number of characters before encountering an end-of-line is greater than the declared length of that string variable.

Floating point arithmetic exception: Underflow occurred.

A value has been computed that is less than the smallest number the computer can store. Check for multipliers whose values approach zero.

Floating point arithmetic exception: Divide by zero attempted.

A value, expression, or variable used as a divisor is zero or approaches zero.

Floating point arithmetic exception: Overflow occurred.

You have computed a number too large to store. Declare real variables to be of type extended or work with smaller numbers.

An error has occurred while opening the device.

The file directory is full.

The disk is full.

Try using another disk.

An attempt has been made to access a file on a disk or volume which is not known to the system.

An attempt has been made to RESET a file which does not exist.

An attempt has been made to open a file that cannot be found.

Insert the proper disk. The use of the *OldFileName* function avoids this type of problem.

An attempt has been made to write to a locked file.

Quit Mac Pascal. Select the file. Select the Get Info option in the File menu and click the lock box to unlock the file.

An attempt has been made to write to a locked volume.

Eject the disk and move the write protect tab to cover the opening.

An attempt has been made to create a file with the same name as a file that already exists.

Pick a new file name. The use of the *NewFileName* function avoids this type of problem.



The Bomb

When the bomb appears a serious error has occurred. Typically your program is too large for the available memory. Often the program itself isn't too large, but it calls too many built-in Mac Pascal procedures and functions from inside a single module, so these procedures and functions combine to overflow memory. To avoid the problem, keep your program and its modules as small as possible. Comments longer than twenty lines can cause bombs, too. If the bomb appears each time you open the program file you may need to use the editor utility or a word processor to get rid of long comments (or write your own text processing program that deletes long comments). If all else fails, try running your program with a fresh Macintosh Pascal disk.

My errors and solutions:

Macintosh Pascal Reserved Words

and	array	begin	case	const	div
do	downto	else	end	file	for
forward	function	goto	if	in	label
mod	nil	not	of	or	otherwise*
packed	procedure	program	record	repeat	set
string*	then	to	type	until	uses*
var	while	with			

Macintosh Pascal Defined Types

real double* extended* integer longint* string* char boolean

* non Standard

Mac Pascal Implementation-Defined Values

32767	MAXINT
-32767	least integer
2,147,483,647	MAXLONGINT
3.4×10^{38}	greatest real
1.2×10^{-38}	real closest to 0 ('least' real)

7	real	
15	double	significant digits
19	extended	

10	real	default field widths
8	integer	

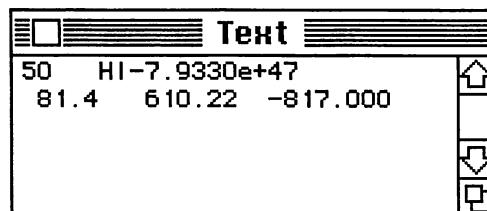
65535	maximum set cardinality
ASCII	character set

97 ord('a') 65 ord('A') 48 ord('0')

Output Format

```
writeln(50:2,'H':5,-7.933e+47:13);
writeln(81.4:5:1,610.22:9:2,-817.0:10:3);
```

↓ ↓ ↓ ↓ ↓



Local Implementation Notes

Non Standard Extensions and Limits

(* comment *) for {comment}
 _ (underscore) is allowed in an identifier
 upper- and lower-case characters are identical
 set of character is OK
 blanks truncated at end of line
 255 character maximum identifier length

Table of Contents

Preface	
A Word to the Student	ix
1 Getting Acquainted with Macintosh Pascal	1
2 Programming Calculations and Graphics	7
3 Procedures and Functions for Problem Solving	17
4 Taking Control of Execution: the for Statement	21
5 Making Choices: the case Statement	24
6 Programming Decisions: the if Statement	29
7 Making Actions Continue: The Conditional Loops	30
8 Character-Oriented Computing: Text Processing	33
9 Extending the Ordinal Types	43
10 Software Engineering	44
11 Arrays for Random Access	46
12 E Pluribus Unum: Records	53
13 Files and Text Processing	56
14 Collections of Values: The set Type	66
15 Abstract Data Structures Via Pointers	67
16 Advanced Topics	68
Appendix: A Hands-On Introduction	73
Index	97
Error Messages and Explanations	Inside back cover
Macintosh Desktop Command Summary	Reference card
Macintosh Pascal Functions and Procedures	Reference card



Norton

W • W • NORTON & COMPANY NEW YORK • LONDON

ISBN 0-393-95598-2