HAYDEN *Macintosh Library*

Apple Press

Introduction to Macintosh™ Pascal

Jonathan D. Simonoff

Introduction to Macintosh™ PASCAL

Introduction to Macintosh™ PASCAL

Jonathan D. Simonoff

Hayden Book Company
A DIVISION OF HAYDEN PUBLISHING COMPANY, INC.
HASBROUCK HEIGHTS, NEW JERSEY / BERKELEY, CALIFORNIA

Composition and production: TECHART, SAN FRANCISCO
Printed and bound by: COMMAND WEB OFFSET, INC.
Cover design: JIM BERNARD
Cover photo: LOU ODOR

Library of Congress Cataloging-in-Publication Data

Simonoff, Jonathan D.
Introduction to Macintosh Pascal.

Includes index.
1. Macintosh (Computer)—Programming. 2. PASCAL
(Computer program language) I. Title.
QA76.8.M3S56 1985 005.265 85-24835
ISBN 0-8104-6562-0

Printed in the United States of America

1	2	3	4	5	6	7	8	9	Printing
85	86	87	88	89	90	91	92	93	Year

Preface

Introduction to Macintosh Pascal was written to provide an easy-to-follow introduction to programming, to programming in Pascal, and to programming the Macintosh.

It is based on the idea that programming is learned best by examining and creating real programs that solve real problems.

Programming is often surrounded with an unnecessary and misleading sense of mystery. Learning to program is really a skill on the order of learning how to write a term paper—with the additional advantage that the computer never writes snide comments in the margin, and never delivers a grade at the end of the semester.

Although a book may have a single name on its cover, all books are the sum of the efforts of many people. I would especially like to thank Susan Keohan, without whose encouragement nothing would have been written, and without whose help nothing would have been finished.

Thanks also to Andrew Singer of Think Technologies; Chris Espinosa of Apple Computer; and Marjorie Lefkowitz and Eben Sprinsock for their great help in testing and verifying the book and its programs.

Thanks for any errors, of course, lie only with myself.

Contents

Introduction

The Educated Macintosh

The subjects covered in this introduction are:

- What Macintosh Pascal is
- What you need to use this book
- How to set up your Macintosh Pascal disk for this book
- How to use this book

What Is Macintosh Pascal?

A computer is a machine for following instructions. When you give instructions to a computer, the instructions have to be in terms the computer can understand. When you write out some instructions for the computer, those instructions are called a **program**.

Sets of allowable computer instructions are called **computer languages**.

Pascal is one of the most popular computer languages. Pascal was designed to be easy to learn and to help the programmer to program well.

The Macintosh Pascal language is a version of Pascal that includes everything in "standard Pascal," and also lets you use all those things that make the Macintosh such a special computer. In particular, you can use the fast, high quality graphics and the sound generator.

To understand what makes Macintosh Pascal really special, you have to know what it is like to program in most versions of Pascal. Usually, you write a program using an **editor**. When you are done, you run a program, called a **compiler**, to translate your program for the computer. If your program has any mistakes, the compiler gives you a list of them, and then you run the editor again, and fix the mistakes. When you compile the program without mistakes, you run another program, called a **linker**, which creates a version the computer can use.

Macintosh Pascal, though, is an **interpreter**. You write your program, and then choose a menu command to run it. If you have a mistake, a little hand points at the line with the mistake. You can then fix the mistake, and run your program again. If you want to see the effect of some change, you just make the change, and you see the effect right away. It is that "instant" quality of Macintosh Pascal that makes the Macintosh Pascal language an ideal learning language.

What You Need To Use This Book

To use this book, you must have:

- A disk copy of Macintosh Pascal (This must be an authentic, authorized copy. Because of the protection scheme used, "bootleg" copies will fail after a few minutes.)
- An Apple Macintosh or Macintosh XL (or Lisa) with MacWorks XL
- A blank disk

This book works for any Macintosh. The programs will all work with 128K of memory (the smallest Macintosh available), but more memory will make it easier for you to write your own programs. The book assumes you have a single (internal) disk drive. A second (external) disk drive also makes things easier, but it is not necessary.

If you use this book with a Macintosh XL, everything except for the programs that use the Mac's sound generator (Chapter 6) will work.

Setting Up For This Book

This section assumes that you are familiar with the basic operation of your Macintosh. You should know:

- How to start your Macintosh
- How to use menus and choose menu commands
- How to use the mouse and mouse button
- How to select, open, and drag icons
- How to select text
- How to use folders

If you don't know how to do these, read the Macintosh manual, or use the guided tour disk that came with your Macintosh.

Preparing For This Book

The Macintosh Pascal disk comes with many useful and interesting programs and much useful information on it. If you do not have an external disk drive, you need to clear some of that information off the Macintosh Pascal disk to make room for the programs created in this book. (If you have an external disk drive, you can skip this section; just store this book's programs on another disk.)

If you have only one (internal) disk drive, follow these directions to make room on the Macintosh Pascal disk for the programs created in this book:

1. If your Mac is on, eject any disk that is in the drive and turn the Mac off.
2. Take one of the disks that comes in your Macintosh Pascal package and put that in the disk drive. (There should be two identical disks in the package. The second is there in case something happens to the first.)
3. Put the disk in the drive, and turn your Mac on. After a few minutes, you should see the Mac's desktop, as shown in Figure Intro-1.

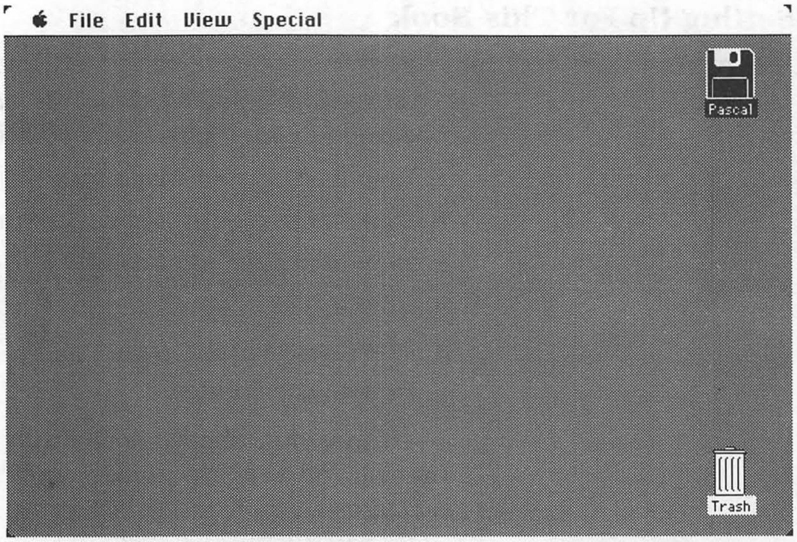


Figure Intro-1 Initial Desktop

4. Select the Pascal icon, and choose Open from the File menu. The icon should open, so that the screen now looks like Figure Intro-2.

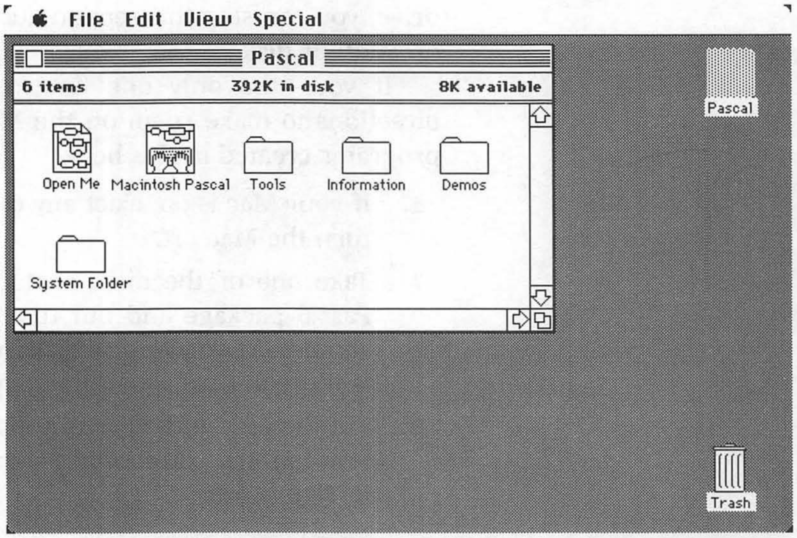


Figure Intro-2 Open Pascal Disk

5. Choose the **New Folder** command from the File menu to create a new folder. A new folder saying "Empty Folder" under it appears.

Notes

If you are using a finder earlier than Finder 4.1, there is no New Folder command. Instead, you create a new empty folder by duplicating the single Empty Folder found on each disk.

6. Change the name of the folder from "Empty Folder" to "Pascal Info Folder". It should now appear as shown in Figure Intro-3.

Figure Intro-3 New Folder

7. Drag the following icons into that folder:
   Open Me
   Tools
   Information
   Demos
8. Click the mouse on the Pascal disk icon image. Use the mouse to pull down the File menu and choose Eject. The Pascal disk pops out of the drive.
9. Insert your blank disk. If you've never used the disk before, a box appears on the screen that asks you if you want to eject the disk or initialize it. Click **Initialize**. If it isn't a new disk, throw everything that is on it into the Trash.

10. After a minute or two, you are asked for a name for the new disk. Type:

    Pascal Info

    If this isn't a new disk, change its name to "Pascal Info" by selecting the old name and typing. You may also want to throw everything on the disk into the Trash or select the new disk (*not the Macintosh Pascal disk*) and choose the Erase command. Make sure there isn't anything important on the disk before you do that.

    Click the mouse in the OK box.

    After a few seconds, the desktop shows again, with your blank disk shown underneath the Pascal disk. The blank disk is labeled "Pascal Info."

11. Drag the Pascal Info Folder icon to the Pascal Info disk icon, and let go. It should disappear.

    The Macintosh displays a box showing the number of files that need to be copied, and it asks you to exchange the disks several times. When it is done, you've copied everything in the Pascal Info Folder to the Pascal Info disk.

12. If the Pascal disk is not in the Macintosh, eject the Release Info disk, and insert the Pascal disk.

13. Drag the Pascal Info Folder into the Trash. You no longer need it, since you have a copy on your Pascal Info disk.

    Your Pascal disk should now look like Figure Intro-4.

    Keep the Pascal Info disk. The information on it is very useful, and the demo programs are fun and well done. There are also three utility programs in the Tools folder that you may use later.

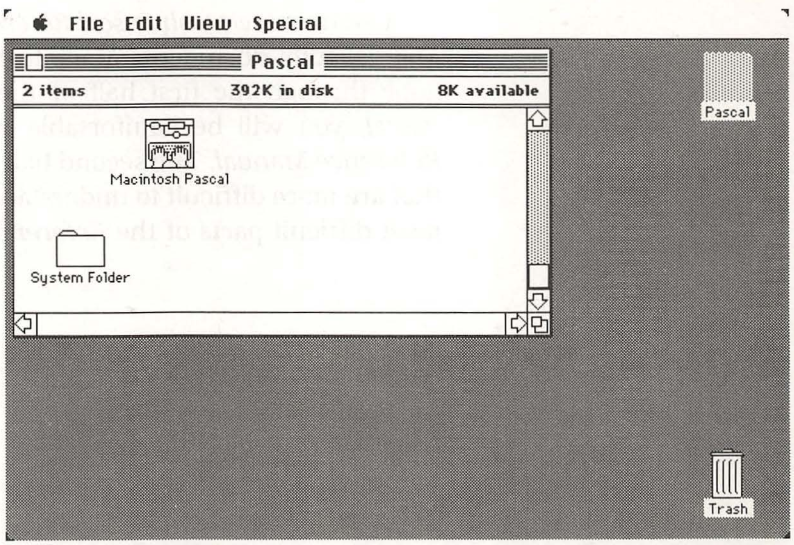


Figure Intro-4 Pascal Disk Prepared for Use

How To Use This Book

This book is designed to be used with Macintosh Pascal and your Macintosh. It isn't hard to learn how to program, but programming is something you can only learn by doing. You can't learn how to program without actually programming a computer any more than you can learn to bicycle without riding a bicycle.

As you read this book, type the programs from the chapters. The instructions are laid out so that you can type them with a minimum of errors. Be careful while you type, though. Computers are very picky, and every character must be right. (Nothing terrible will happen if you make a mistake, you'll just have to find it and fix it.)

Save the programs—they do useful things, and are designed so that you can use them in programs you write later. Also, some of the larger programs are built up chapter to chapter.

Feel free to experiment. Modify copies of the programs. The best way to learn to program is to play and explore. The programs in this book provide working frameworks for using your imagination.

Use the *Macintosh Pascal Reference Manual*. The *Reference Manual* fully documents Macintosh Pascal. By the time you've gone through the first half of the *Introduction to Macintosh Pascal*, you will be comfortable with reading and using the *Reference Manual*. The second half of this book deal with topics that are more difficult to understand, and acts as an aide for the most difficult parts of the *Reference Manual*.

CHAPTER

1

Can I Be an Artist if I Can't Draw a Line?

Starting Macintosh Pascal

Welcome to programming your Macintosh. This chapter shows you how to use your Macintosh to draw a line. It may seem silly to use a computer to do something so simple, but, once you understand how to do that, you will be ready to build programs that do more useful things and use your Mac's abilities.

Before you read this chapter, you should have followed the instructions in the Introduction for setting up your Pascal disk.

If your Mac is already on, eject any disk that is in the disk drive, and turn your Mac off.

Insert the Macintosh Pascal disk in the slot in the front of the Mac, and turn on the Mac. A small Mac with a smiling face should appear on the screen. (If it doesn't, try looking in Appendix B, titled "Help, Please, My Mac Is Burning," in the back of this book.) After a few seconds, the desktop appears.

Make certain that the Pascal disk icon is selected (indicated by darkening the icon). You may have to click once on the disk icon to select it.

Pull down the File menu and choose Open, as shown in Figure 1-1. The Pascal icon opens up to show the contents of the disk.

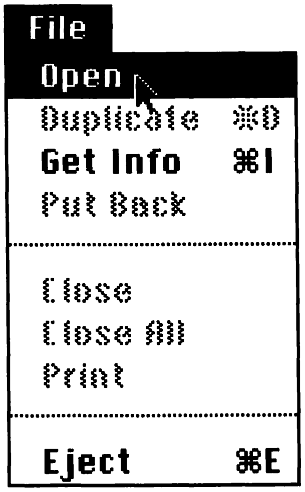


Figure 1-1 Open Command

The icon that says Macintosh Pascal under it, the one that looks like Figure 1-2, represents the Macintosh Pascal application. You use the Macintosh Pascal icon to create and run programs.

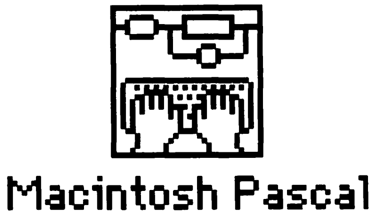
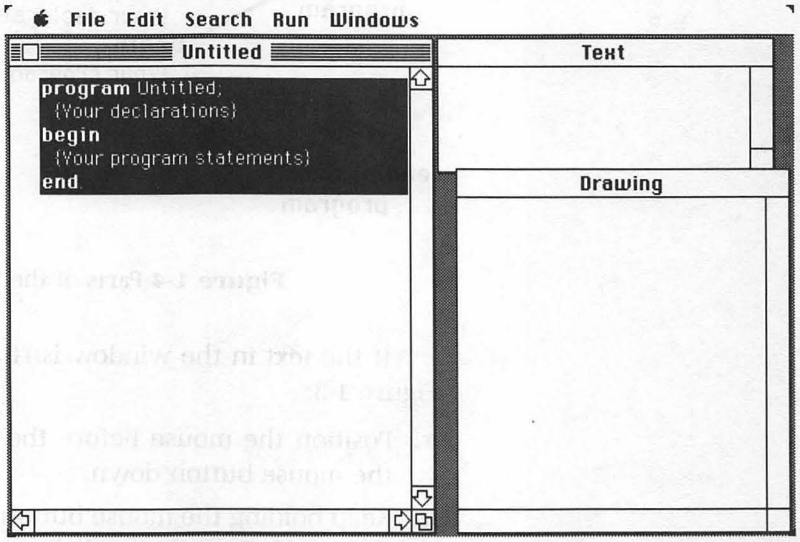


Figure 1-2 Macintosh Pascal Icon

Use the mouse to click once on the Macintosh Pascal icon, so that it is selected. Pull down the File menu and choose Open.

After a few seconds, the screen is divided into three windows, as shown in Figure 1-3. These windows are much like other windows you have seen on the Macintosh. You can move them around, change their size, and make them disappear and reappear. Don't do anything yet, though.



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

Figure 1-3 Initial Pascal Windows

The leftmost window, the one that says "Untitled" in its title bar is the **programming window**. Whatever you type in that window is an instruction that becomes part of a Pascal program.

The text that is already in the programming window is the basic skeleton of a program. Notice that all the writing there is already selected, the way it appears in Figure 1-3.

The text in the window contains the elements that every Pascal program must have. Figure 1-4 points out the different elements. Every program begins with the word "**program**", and contains the words "**begin**" and "**end**". The parts within curly brackets ({}) are **comments**, which are ignored by Pascal. Comments are put in programs for the convenience of people, to make the programs more clear. Ignoring the comments, this is the smallest Pascal program you can write. It doesn't, however, do anything at all. Don't worry about the details of the parts of this program. They'll all be explained as you proceed through this chapter and write a short program that actually does something.

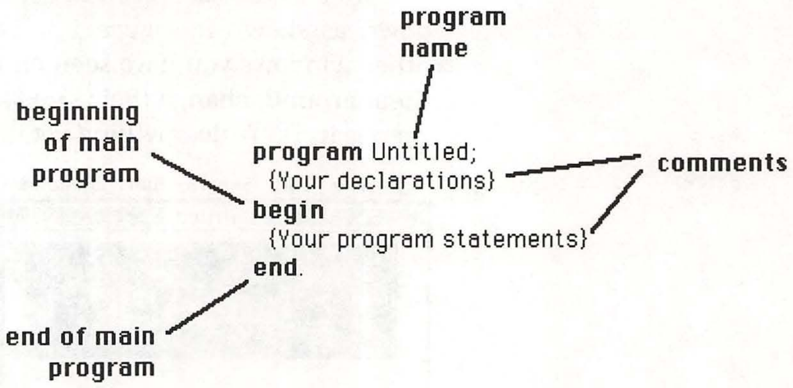


Figure 1-4 Parts of the Smallest Program

If the text in the window isn't still highlighted as shown in Figure 1-3:

1. Position the mouse before the word "**program**" and hold the mouse button down.
2. Keep holding the mouse button down and sweep the mouse down to below the "**end**".
3. Let go of the button.

The text should appear the way it does in Figure 1-3. If not, try these steps again, and be careful to hold the mouse button down the entire time.

Writing a Program

The first thing you need to do is clear the window. The program skeleton that Macintosh Pascal supplies for you is useful when you write your own programs, but now it is in the way.

To clear that text out of the way, as long as all the text is still selected, pull down the Edit menu and choose **Cut**, as shown in Figure 1-5.

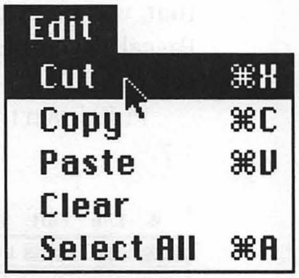
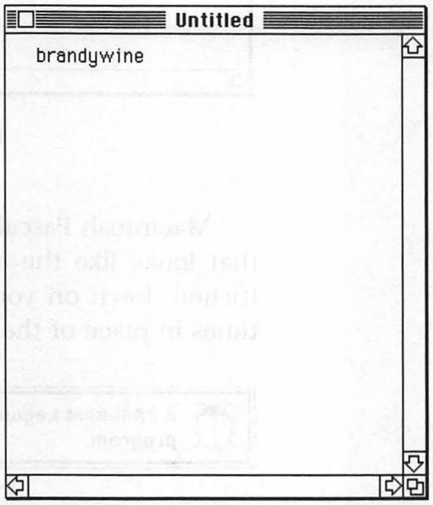


Figure 1-5 Choosing Cut Command

It is time to stretch your fingers out and begin entering commands at your keyboard. Start out by confusing Macintosh Pascal a bit, just to see what it feels like. Type:

brandywine

Then press the Return key on the right side of the keyboard. The programming window should now look like Figure 1-6.



```
brandywine
```

Figure 1-6 Brandywine

Whenever something is to be typed from this book, it is printed like that. For the rest of this chapter, <RETURN> is given at the end of lines to indicate you should press the return key.

Now, tell Macintosh Pascal to act on what you've typed. To do that, you need to give the **Go** command, which says to Macintosh Pascal, "I've finished writing my program. Now see what you can do with it."

Pull down the **Run** menu, and choose Go, as shown in Figure 1-7.

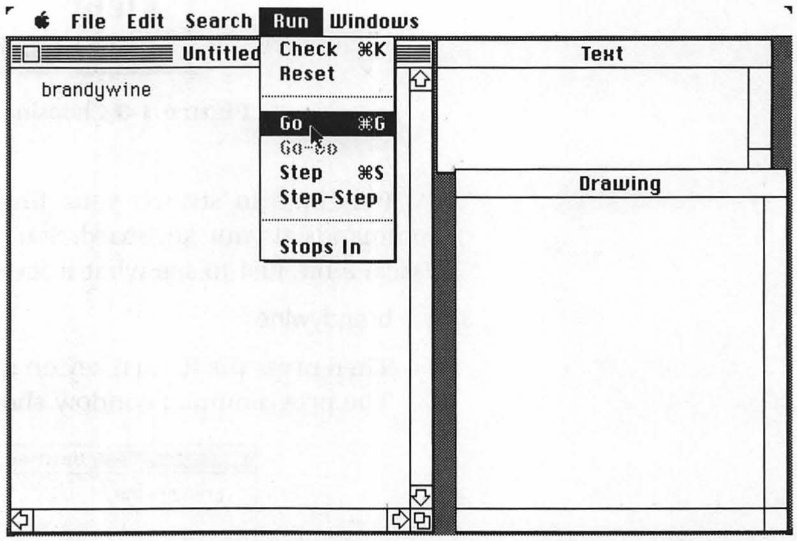
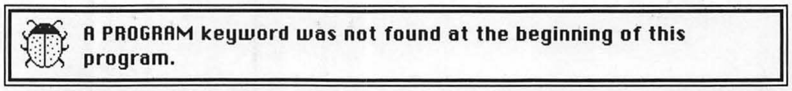


Figure 1-7 Choosing Go

Macintosh Pascal beeps, and gives you an **error message** that looks like the one in Figure 1-8. (If you have the sound turned down on your Mac, the menu bar flashes a couple of times in place of the beep.)



A PROGRAM keyword was not found at the beginning of this program.

Figure 1-8 Error Message

The long, narrow box with a picture of a bug on the left side is an error message. The message is Pascal's way of saying, "Huh? I didn't understand that." The text of an error message tells you something about the error. This message refers to the fact that programs must begin with the word "program". To get rid of the error message, click anywhere in the long narrow box.

You need to remove the line you just typed in. First, select the word "brandywine", and then choose the Cut command from the Edit menu.

Notes

As you get more familiar with how things are done on the Macintosh, you will learn there are quicker ways to do things like removing text. If you already know other methods, feel free to use them.

One thing to remember about using Macintosh Pascal: whenever you make a mistake, just go back and fix it, by using the the Cut command or the Backspace key, or any method you wish. There are several ways you can change what you typed in. While you use this book, whenever you get an error message that you weren't expecting, or something doesn't work right, carefully check what you typed. You may have made a typing error or misread something. Go back and fix it. Macintosh Pascal immediately forgets your error. The end of this chapter has a section on how to change things you've typed.

The word "brandywine" was an error because Pascal programs are supposed to begin with something that identifies them as programs. As the error message indicates, programs begin with the word "program".

Notes

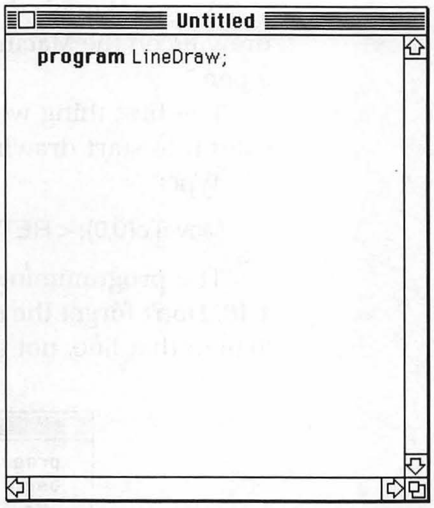
Macintosh Pascal often changes the appearance of text, sometimes to make the program easier for you to read and sometimes to indicate an error. It displays text that seems to be wrong in outlined letters. In general this is helpful. Occasionally, what you type will not appear in normal type when it should. Unless there is an error in what you've typed, you can clear up any problems by pressing the Enter key, which is the key to the right of the space bar. Enter makes Macintosh Pascal read what you've typed.

Notice the way the word "**program**" is printed in bold type. This book prints **program** that way to indicate the word is one of a special class of words: it is one of Pascal's **reserved words**. There are 36 Pascal reserved words. Reserved words have special meaning in Pascal, and you can't change their meaning. Reserved words can be used in programs only in specific ways. **program** can only be used as the very first word in a program. It means, "The following is a Pascal program called— ".

Every program has a name. The name must not contain any spaces, or characters other than numbers, letters, or underscores. It can be up to 255 characters long. Call this program "LineDraw" (with no space between the two words). Type:

```
program LineDraw;<RETURN>
```

The programming window should now look like Figure 1-9.



```
program LineDraw;
```

Figure 1-9 First Line of a Program

Make sure you typed the semicolon (;). Semicolons are very important in Pascal, so you have to be careful you put them in the right places. In general, semicolons separate any two statements or program parts.

The line containing the word "**program**" is the first program part: the program's heading. It must be separated by a semicolon from whatever follows. (Appendix A describes the use of semicolons completely. Don't worry about them for now—the programs in this book always have them where they are needed.)

When you type the semicolon, the word "**program**" changes so that it appears in bold, just as it does in this book. When Macintosh Pascal recognizes a reserved word, it changes it to bold, lower case characters.

Now type:

begin<RETURN>

begin is another reserved word. There is never a semicolon after **begin**. You can think of semicolons in Pascal as something like periods in English. You put a period at the end of a sentence in English, and a sentence expresses a complete thought. Similarly, you put semicolons between statements in Pascal, which are complete instructions. **begin** says in Pascal, "The beginning of this is—", so you don't follow it with a semicolon.

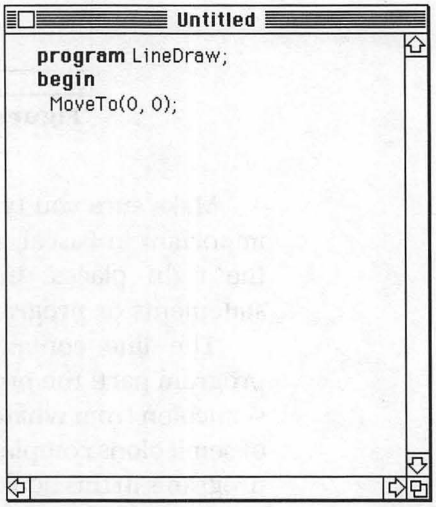
As you know, this program is going to draw a line. All drawing on the Macintosh is done by an electronic equivalent of a pen.

The first thing we have to do is move the pen to where we want it to start drawing.

Type:

```
MoveTo(0,0);<RETURN>
```

The programming window should look as shown in Figure 1-10. Don't forget the semicolon. Make sure you typed two zeros (0,0) in that line, not two letter O's.



```
program LineDraw;
begin
  MoveTo(0, 0);
```

Figure 1-10 Adding MoveTo

MoveTo is an order to the Macintosh, but notice that nothing has happened yet. Macintosh Pascal doesn't do anything with the orders you type in the programming window until you tell it to run the program. When you tell Macintosh Pascal to run a program, it reads the instructions in the program, checks if they are understandable, and, if they are, carries them out. Until you give a Run command, the programming window is just a specialized kind of word processor, like MacWrite, but with some special features that make it easier to write programs.

MoveTo(0,0) tells the Macintosh to move the pen to position (0,0). (0,0) are the **coordinates** that identify the upper left corner of the **Drawing window**. (The Drawing window is the one in the lower right part of the screen, the one that says "Drawing" in its title bar.) When you use Macintosh Pascal, coordinates always give a position in the Drawing window.

Coordinate numbers get larger as you go down and right. Imagine that the Drawing window is divided by vertical and horizontal lines, as shown in Figure 1-11. When you want to identify a point in the window, you identify the two lines that cross at that point. For example, Figure 1-11 highlights the point at (10,10). The first number is the distance from the left side of the window, while the second number is the distance down from the top of the window.

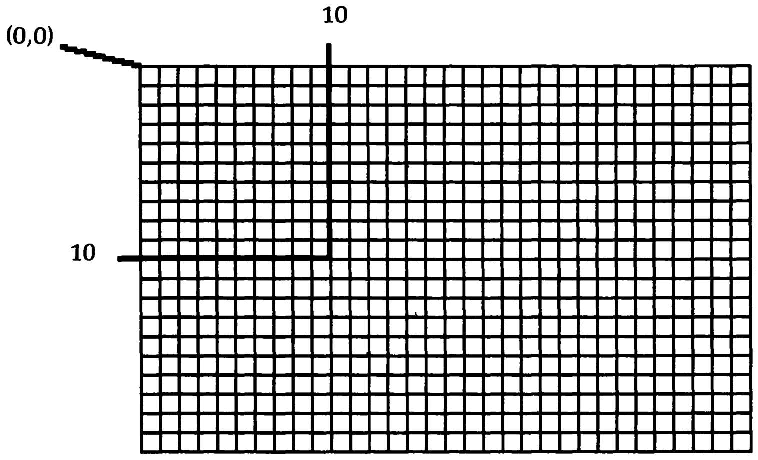


Figure 1-11 The Coordinate System

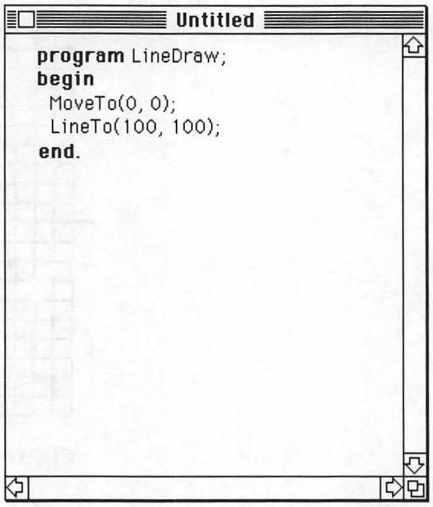
Look closely at your Macintosh screen. Can you see the little dots? (They are easiest to see in a gray area.) The dots are called **pixels**, short for picture elements. Each pixel is equivalent to one coordinate number. There are 72 pixels per inch on the Macintosh. When you give a pair of coordinates, you are identifying a particular pixel.

Now type:
LineTo(100,100);<RETURN>

LineTo, like MoveTo, is an order to the Macintosh to move its pen. With LineTo, though, the pen draws a line as it moves. The effect of these lines is to draw a line from the upper left corner, diagonally across the box to position (100,100).

To finish the program, type:
end.<RETURN>

Notice in figure 1-12 that Macintosh Pascal turned the word into bold, indicating **end** is a reserved word. The word "**end**" tells Pascal that you have finished some sequence. There is an **end** for every **begin**. Don't forget the period after this **end**. The period means you've finished your whole program. Unlike a human being, Macintosh Pascal won't assume you are done telling it what to do until you tell it you are finished.



```
program LineDraw;
begin
  MoveTo(0, 0);
  LineTo(100, 100);
end.
```

Figure 1-12 Finished LineDraw

The period can be understood in terms of English grammar, like the semicolon. You can have a sentence in English that consists of a series of statements, each separated from the next statement by a semicolon. When the entire sequence of statements is complete, you close the sentence with a period.

In a similar way, a program is a series of statements. At the end of the series of statements, the end of the program, you give a period.

Now run the program. Pull down the Run menu, and choose Go. (If you made a mistake in typing the program, Macintosh Pascal may display some of the program's text in outlined type or give you an error message. Don't panic—just compare your program to what appears in Figure 1-12 and fix what is wrong.)

After a few seconds a line appears in the Drawing window, as shown in Figure 1-13.

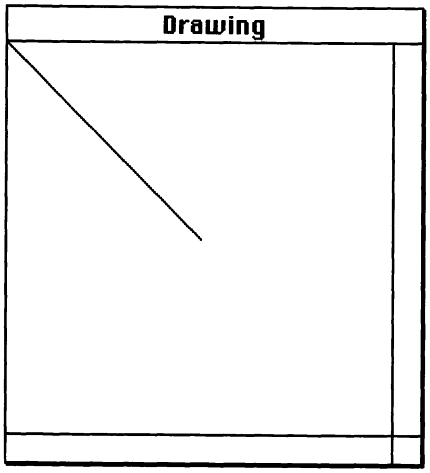


Figure 1-13 The Results of LineDraw

Congratulations. You've written your first Pascal program. Although this is a small, simple program, it has the essential features of every Pascal program—a name, a **begin**, an **end**, and a section that does something. Other programs will do more complex things, but you can always break them down into small simple actions like the actions in this program. If you can understand this program, you can understand any program.

You may have noticed that as you typed the program, Macintosh Pascal changed the appearance of what you typed. The type may have moved a bit, it may have changed into bold type, or, if you typed something incorrectly, the type may have changed into outlined type.

Whenever you type a semicolon or a <RETURN>, Macintosh Pascal reads the program and puts all the text into a particular format, indenting sections of the program so that they line up correctly. It makes it easier to spot errors and to understand what a program does.

Macintosh Pascal also checks to see if what you typed makes sense in the Pascal language. If it recognizes a reserved word, it changes the word to bold. If it recognizes a mistake in the way you used something, it changes the suspect words to outlined type.

From now on, this book does not show <RETURN> at the end of every line. You can type a <RETURN> after every line, if you want. However, Macintosh Pascal's automatic formatting puts in <RETURN> for you, as long as you include at least a space or semicolon.

Notes

Blank lines have absolutely no significance in the Pascal language. Macintosh Pascal ignores them. Similarly, additional spaces have no meaning. You can almost always put as many spaces as you like where one is required.

Sometimes when you correct a mistake that Macintosh Pascal displayed in outlined type, the letters remain in outlined type, even though they are now correct. That is because Macintosh Pascal checks what you've typed only at certain times, particularly after you type a <RETURN> or a semicolon. You can force Macintosh Pascal to check what you've typed and put it in the correct format by pressing the Enter key, which you can find to the right of the space bar.

Although Macintosh Pascal recognizes some errors right after you type them, other errors depend on context. When you typed "brandywine" earlier in this chapter, you could have built a program around it so that it made sense. Macintosh Pascal couldn't tell the word was an error until you asked it to run the "program."

Errors are no big deal. Don't worry about hurting Macintosh Pascal or your Macintosh. There is no way you can do that, short of physical violence.

Editing a Program

Macintosh Pascal has many of the same editing capabilities as MacWrite, and a few additional ones.

You can type in the ordinary way; you can use the Backspace key to delete text; and, in addition, you can select groups of text by:

- Clicking and dragging the mouse.
- **Double-clicking**, which selects the word under the mouse pointer.
- **Triple-clicking**, which selects the line under the mouse pointer.
- Clicking once, letting up the mouse button, moving the mouse, and holding the Shift key down while you click the mouse again.

That selects everything between the two mouse clicks.

You can also double-click or triple-click and then hold the mouse button down while sweeping the mouse across text. That selects text as whole words or lines.

At any time, there are three possible conditions that change what happens when you type. They are all indicated by what you can see on the screen.

- The programming window may have an **insertion point**. The insertion point is a blinking vertical bar that indicates where characters appear when you type on the keyboard. When you use the Backspace key, characters are removed starting at the insertion point and moving backwards.
- The programming window may have some text selected. If you type when text is selected, the text disappears, and whatever you type replaces it.
- The programming window may not be selected. A window shows it is selected by the thin lines that appear in the title bar at the top of the screen. If you type when the programming window is not selected, nothing is added to the programming window. Assuming a program is not running, you can select the programming window by clicking in it. The programming

window may also be hidden. You can bring it back to the screen and select it by choosing the name of the program from the **Windows** menu. (If the program has no name, the programming window is called *Untitled*.)

Programs are often long enough so that only part of the program is visible in the window. If so, it is possible that the insertion point or a selected block of text may be scrolled out of the window. If you type, the effect is the same as if the selection was visible, so you can accidentally delete part of your program if you are not careful.

Warning

Unlike many Macintosh programs, Macintosh Pascal does not have an **Undo** command. If you accidentally delete part of your program, you are out of luck. You can revert to the last saved version of your program, but that may destroy other changes you've made.

Finishing Up

Before going on, **save** the *LineDraw* program, because it is used in Chapter 2. When you save a program, it is stored on the disk along with Macintosh Pascal.

To save *LineDraw*:

1. Pull down the File menu and choose the **Save As...** command, as shown in Figure 1-14. A dialog box like the one in Figure 1-15 appears.

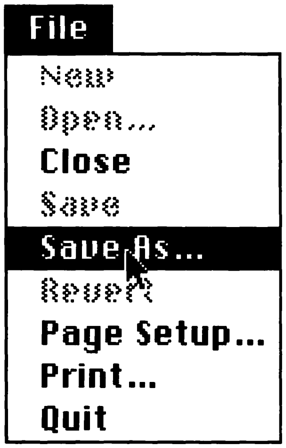


Figure 1-14 Choosing Save As...

2. Type:

   LineDraw

3. Click on the Save button.

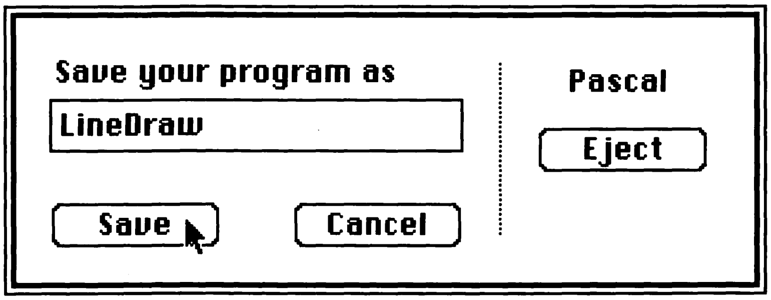


Figure 1-15 Save Dialog Box

Your disk drive hums while Macintosh Pascal saves your program. When the Mac is finished, pull down the File menu and pick **Quit**. The Macintosh Pascal windows disappear, and, after a few seconds, the desktop reappears.

There is a new icon next to the Macintosh Pascal icon. That icon represents the saved copy of *LineDraw*. To get *LineDraw* back, click on the LineDraw icon so it is selected, and choose Open from the File menu. Macintosh Pascal starts up, and the LineDraw programming window replaces the Untitled programming window.

Warning

You should never save a program with the name *Untitled.* If you do so, when you start Macintosh Pascal, that *Untitled* program will replace the normal Untitled program skeleton.

To get back to the desktop, choose Quit from the File menu again.

If you want to turn off your Mac:

1. If Pascal is now running, choose Quit from the File menu. (If you aren't sure if Pascal is running, look in the File menu. If there is a Quit command in the menu, choose it.)
2. When the desktop appears, choose **Shut Down** from the **Special** menu. The Macintosh Pascal disk and any others are ejected by the Macintosh, and the Macintosh beeps. You can now turn the Macintosh off or boot with another disk.

(If you have a finder earlier than 4.1, the Special menu has no Shut Down command. In that case, when the desktop appears, choose **Close All** from the File menu to close the Macintosh Pascal disk window and all other open windows, and then select the Macintosh Pascal disk icon and eject it. If you have any other disks showing on the desktop, eject them, too. When all disks have ejected from their drives, you can turn the Macintosh off.)

It is not a good practice to turn the Macintosh off when a disk is in the drive, and, especially, when Macintosh Pascal or some other application is running. If you do, you can damage the disk so the Mac won't be able to use it. Serious errors that force you to turn the Mac off with a disk in the drive do sometimes occur. If that happens, everything will usually be OK. Sometimes, though, your disk may need a little first aid. See Appendix B for some help.

Do More

Most chapters end with a Do More section. If you want to do more on your own (always a good idea when you are learning to program) follow the suggestions in the Do More section, or think up some of your own.

1. Try modifying *LineDraw* to get it to draw lines between different parts of the Drawing window. See if you can find exactly where the lower right hand corner of the window is located.
2. Try drawing several lines to make a shape, such as a box or a triangle. What happens to the pen after you draw a line?
3. Move the Drawing window, or make it bigger, and run your program. What happens to the coordinates? Does that tell you something about how the Macintosh handles windows?

QUICK SUMMARY

Chapter 1 introduces you to a very simple program that draws a line diagonally across the Drawing window. The chapter uses the following statements, routines, and concepts:

begin	is a Pascal language reserved word that tells Pascal, "Now do this— ". **begin** is never followed by a semicolon.
Brandywine	as used in this chapter, is used to represent something Macintosh Pascal can't understand. Otherwise, it is an alcoholic drink distilled from wine, now usually shortened to brandy, or a river in some imaginary world.
Coordinates	define a position in the drawing window.
end	is a Pascal language reserved word that tells Pascal, "That's it for this sequence."
Error message	is a message from Macintosh Pascal that attempts to explain why it couldn't follow the instructions contained in a program.
Go	is the menu command that tells Macintosh Pascal to carry out the instructions contained in the program, or give an error message that tells why it can't understand the instructions.
Insertion point	is the place where letters appear when you type on the keyboard. It is usually marked on the Macintosh by a blinking vertical line.
LineDraw	is the name of the program produced in this chapter, one of the simplest programs you can write in Pascal. There is nothing special about this name; you could call the program *Harvey* if you wanted, as long as the name you give contains nothing but letters, numbers, or underscores (_), and has no more than 255 characters.
LineTo(x,y)	is a pre-defined order that draws a line from the current pen position to the coordinates x and y.
MoveTo(x,y)	is a pre-defined order that moves the pen to the coordinates x and y, without drawing anything.
Pen	is the electronic equivalent of an ink pen that draws in the Macintosh Pascal drawing window.
Period (.)	is a Pascal language symbol that tells Pascal that this is the end of the program.
program	is a Pascal reserved word that tells Pascal that this is the beginning of a program.

Reserved words	are words that have a special meaning in the Pascal language, and cannot be used for any other purposes. Macintosh Pascal always changes the reserved words you type so that they appear in lower case bold type. Something to remember is that there are words with special meaning in the Pascal language that are not reserved words. In those cases, you can redefine those words to mean something else, although doing so is rarely a good idea. None of those words are used in this chapter, but they will be in Chapter 2, and the rest of this book.
Run	is the menu that contains the Go command, and a number of other commands to Macintosh Pascal. The verb "to run," as in "Run that program," means the action of ordering a computer to follow the instructions in a program.
Save	is the menu command asking Macintosh Pascal to make a permanent copy of your program. The first time you save your program, you use the Save As command so that you can give the permanent copy a name. After that, every time you choose Save from the menu, the program is recorded, so that the changes you've made appear in the permanent copy. A saved program has an icon that shows up in the Macintosh Pascal disk directory.
Semicolon	is a Pascal language symbol that separates program parts and complete statements.

CHAPTER 2

Lining Up

Variables and the Mouse

The *LineDraw* program in Chapter 1 is one of the simplest programs you can write. This chapter shows you how to extend that program so that it does a bit more.

Macintosh programs in general react to what the user does. Computers should respond to people.

You usually tell the Macintosh what to do by using the mouse to point at something on the screen. Your program can find out where the mouse pointer is located.

You should now have the Mac's desktop displayed. If your Macintosh is turned off, insert the Macintosh Pascal disk and turn the Mac on. After a few seconds, the desktop is displayed.

Follow these steps to open program *LineDraw*:

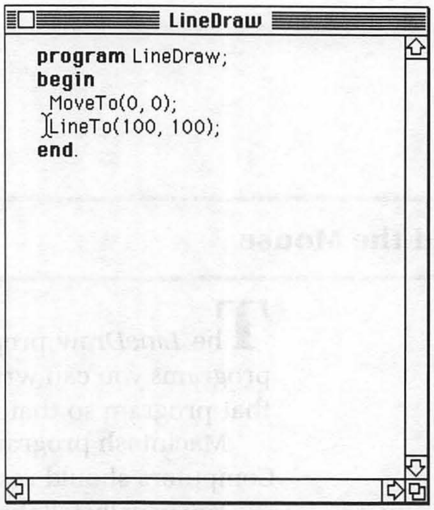
1. Open the Pascal disk icon by clicking on it and choosing Open from the File menu.
2. Click on the LineDraw icon.
3. Choose Open from the File menu.

After a few seconds, the three Macintosh Pascal windows are displayed, with the LineDraw window on the left.

Follow these steps to alter *LineDraw* so it uses the mouse:

1. Move the mouse pointer to just before the "L" in LineTo, as shown in Figure 2-1.
2. Click the button once, so you get an insertion point just before the "L".
3. Type:

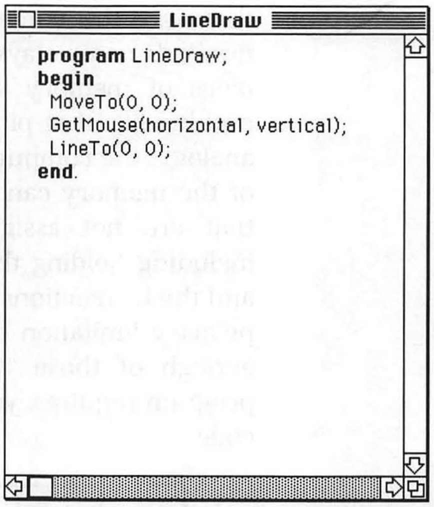
   GetMouse(horizontal, vertical);



```
program LineDraw;
begin
 MoveTo(0, 0);
 LineTo(100, 100);
end.
```

Figure 2-1 Placing Insertion Point

Be careful to include the semicolon. The programming window should now look like Figure 2-2.


LineDraw

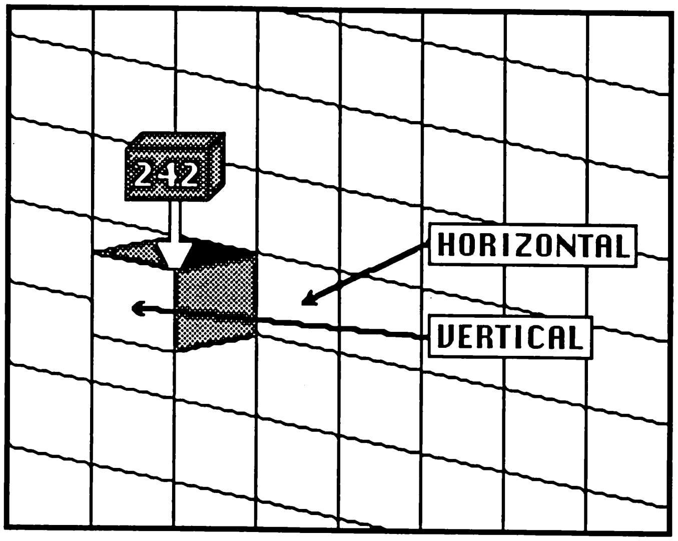
```
program LineDraw;
begin
  MoveTo(0, 0);
  GetMouse(horizontal, vertical);
  LineTo(0, 0);
end.
```

Figure 2-2 Adding GetMouse

GetMouse, like LineTo and MoveTo, is a special kind of order in Pascal, called a **procedure call**. A procedure is actually a little program by itself, sometimes called a **subprogram**, **subroutine**, or, simply, a **routine**. There are a large number of predefined subprograms you can call from Macintosh Pascal programs. Most (but not all) subprograms have a set of parentheses following them, in the way MoveTo, LineTo, and GetMouse do. You give information to the subprogram and get information back from the subprogram through items you put in the parentheses. Those items are called the parameters of the subprogram. Learning how to use a subprogram involves primarily learning the significance of the **parameters** of the subprogram.

GetMouse finds the position of the mouse pointer and returns it in terms of the number of pixels from the upper left corner of the Drawing window. In this case, the values come back to your program filed under the names *horizontal* and *vertical*.

Horizontal and *vertical* are **variables**. Variables are names used to represent values. One of the things that makes a computer so useful is that it can compute new values from some starting place, and then use the new values to continue. In order to do that, a computer must have a way of referring to those changing values. Variable names are like names on file drawers—the contents of the drawer may change, but the drawer's name stays the same. Figure 2-3 shows an imaginary piece of memory. Although computer memory really looks nothing like that picture, there is an important concept in the analogy: the computer has a huge array of memory, and pieces of the memory can be assigned to variables. Parts of memory that are not assigned to variables serve many purposes, including holding the instructions, or **code** of your program, and the instructions that make up Macintosh Pascal. This is the primary limitation on the length of programs: there must be enough of those "file drawers" to hold the variables your program requires, your program's code, and Macintosh Pascal's code.



An Imaginary Piece of Memory

Figure 2-3

Variables in Macintosh Pascal programs can have names up to 255 characters long. Names can contain only letters, numbers, and underscores. The first character must be a letter. You can use upper or lower case letters, but they don't make a difference in the Pascal language; VERTICAL is the same as vertical. Variable names in this book always start with a lower case letter. You can make up any variable name you want, except that you can never use the reserved words (the words that Macintosh Pascal always shows in bold, lower case letters). You can take advantage of the fact that long variable names are allowed to give your variables meaningful names, in the way *horizontal* and *vertical* are used in this program to hold the horizontal and vertical coordinates of the mouse.

Replace the 100,100 in the line:

```
LineTo(100,100);
```

with:

```
horizontal, vertical
```

That line should now be:

```
LineTo(horizontal, vertical);
```

(There are several ways you can accomplish that replacement. See the editing section at the end of Chapter 1 for some suggestions, if you need them.)

This new line replaces the old line.

When you used LineTo before, you gave the actual coordinates for the end point of the line. You could have used variables, and the effect would have been the same, as long as you gave the variables the values 100 and 100.

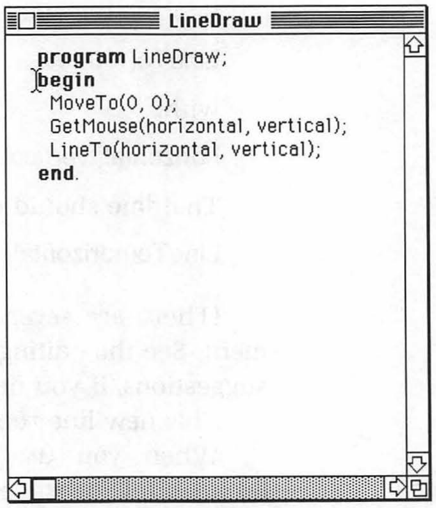
GetMouse puts the coordinates of the mouse position into the variables *horizontal* and *vertical*. LineTo(horizontal, vertical) draws a line to that point.

The program isn't finished yet, though. You have to define variables before you can use them in Pascal programs. Defining a variable means stating what kind of values it can hold. Variables can hold many different kinds of information, including numbers, letters, and words.

Mouse positions are given in terms of coordinates in the drawing window. Those are whole number values: 1, 2, 3, and so on. A mouse position can never be something like 3.5, or 2.17398. Whole numbers like 1, 2, and 3 are called **integers**. Therefore, you have to tell Macintosh Pascal that *horizontal* and *vertical* can only hold integers.

Move the mouse pointer to just before the word "**begin**", and click once to get an insertion point, as shown in Figure 2-4. Type:

```
var

horizontal, vertical: INTEGER;
```



```
program LineDraw;
begin
  MoveTo(0, 0);
  GetMouse(horizontal, vertical);
  LineTo(horizontal, vertical);
end.
```

Figure 2-4 Placing Insertion Point

Notice that there is no semicolon after the word "**var**", but there is one after INTEGER. "**var**", which is a reserved word, is an instruction in the Pascal language meaning, "This program is going to use the following variables—."

The word "INTEGER" in the declaration is a predefined **data type**. (All predefined data types are capitalized in this book. You can capitalize your data types or not, as you wish. With data types, as with all names in the Pascal language, case makes no difference to the computer.)

Notice that Macintosh Pascal does not change the word "INTEGER" to bold lower case. That is because INTEGER is not a reserved word.

Every variable in Pascal has a type which defines what kinds of values the variable can hold. Integers can have any whole-number value between −32767 and 32767. (Integers can hold numbers up to that magnitude because of the amount of memory space Pascal allots for them—the size of an Integer-type "file-drawer.")

The advantage of having to define the type of every variable you use is that Pascal will tell you if you use the variable incorrectly, or if you try to use the same variable name for a second variable.

You define variables by giving the variable name, a colon, and a data type. Every variable definition is ended by a semicolon, which indicates you are done with the definition. The format for a variable definition is:

varName : dataType;

There are many data types. You will use others later in this book.

This program draws a line from the upper left corner of the Drawing window to the current mouse position. Try running it. Pull down the Run menu and choose Go. After you choose Go, quickly move the mouse pointer to somewhere in the Drawing window. Run the program a few times, just to get a feel for it. Notice that the line does not appear outside the drawing window, even if you hold the mouse so the program attempts to draw a line outside the window. Drawing commands can never draw outside the Drawing window. Nothing bad happens if you try, the drawing just doesn't show on the screen.

Functions

The *LineDraw* program is not very sophisticated. It works because it takes a few seconds for Macintosh Pascal to set up the program, so there is time for you to move the mouse into the Drawing window. It would be better if the program waited for some kind of signal from the user that tells the program it is time to draw the line. A convenient signal on the Mac is the mouse button.

You can ask Macintosh Pascal whether the mouse button is pressed by calling the **Button** function.

Up to this point you have used the predefined procedures MoveTo, LineTo, and GetMouse. You gave information to MoveTo and LineTo, and got information back from GetMouse. Button also gives you information.

However, MoveTo, LineTo, and GetMouse are **procedures**, while Button is a **function**. All procedures and functions are subprograms; the difference is that, when you call a procedure, values may (or may not) be returned in the parameters of the procedure, while, when you call a function, a value is always returned represented by the function's name.

If a procedure is a car that goes off and may come back towing some values, a function is a car that always comes back with a value in its front seat.

You call a function by giving its name in almost any place you can use a value. The function returns the needed value.

Although Button does not have any parameters, functions, in general, can have parameters in the same way procedures can.

The return value of Button is the answer to a TRUE/FALSE question: "Is the mouse button pressed?" TRUE means that the mouse button is pressed; FALSE means that it is not pressed.

Function return values have defined types in the same way variables have defined types.

The return value of Button is of type **BOOLEAN**. A BOOLEAN can only have the values TRUE and FALSE. BOOLEANs are very useful in making decisions, so that the computer does something different depending on whether something is TRUE or FALSE.

So, getting back to the program, rather than drawing a line to wherever the mouse happens to be, you want the program to wait to draw the line until the mouse button is pressed. Every time you call Button, it tells you if the mouse button is down. You need the program to repeatedly call Button until it returns TRUE (which means the button is down), and then draw the line. You can use the **repeat/until** statement, which lets you repeatedly check some condition until it is TRUE. **repeat/until** is one of a number of statements that sets up a **loop**: a group of statements that repeat.

Position the mouse button before GetMouse, and click the button to get an insertion point. Type:

```
repeat
```

There is no semicolon after **repeat**. That is because, like **begin**, **repeat** indicates the beginning of a statement, and is not a complete statement in itself.

Place an insertion point before LineTo and click the button to get an insertion point. Type:

```
until Button;
```

The program should now look like Figure 2-5.

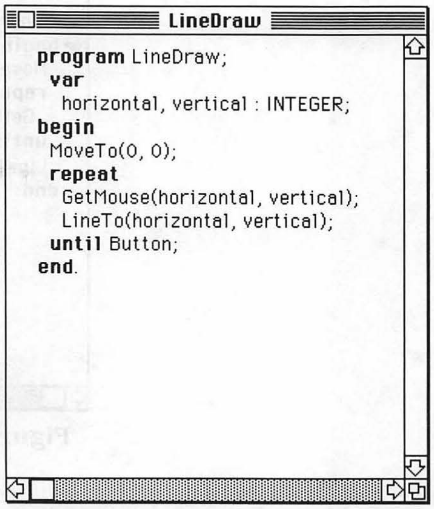
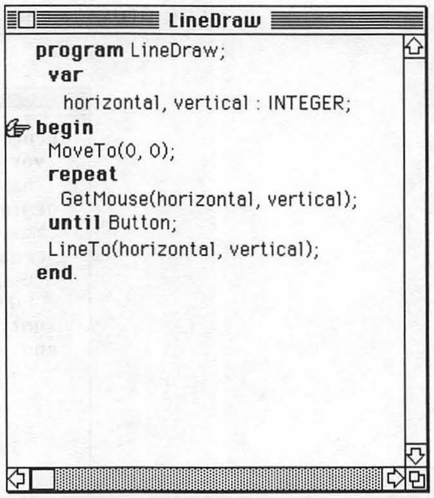

```
program LineDraw;
  var
    horizontal, vertical : INTEGER;
begin
  MoveTo(0, 0);
  repeat
    GetMouse(horizontal, vertical);
    LineTo(horizontal, vertical);
  until Button;
end.
```

Figure 2-5 Button LineDraw

Try running this program. As predicted, the line is not drawn until you press the mouse button.

Macintosh Pascal can show you how it is repeatedly executing the GetMouse statement. Pull down the Run menu. Do you see the **Step** command? That command tells Macintosh Pascal to carry out the instructions in the program one line at a time.

Choose the Step command. A little hand appears on the left hand side of the programming window, as you can see in Figure 2-6. That hand points at the statement that was just executed. By repeatedly giving the Step command, you can go through a program a line at a time. The Button function responds anytime you press the mouse button, so if you keep on picking the Step command from the menu, Button returns TRUE the first time the **until** is reached, because you must press the mouse button to pick Step from the menu.



```
program LineDraw;
  var
    horizontal, vertical : INTEGER;
begin
  MoveTo(0, 0);
  repeat
    GetMouse(horizontal, vertical);
  until Button;
  LineTo(horizontal, vertical);
end.
```

Figure 2-6 After Step Command

Pull down the Run menu again. Do you see the cloverleaf sign with a letter next to the Step and Go commands (Figure 2-7)? The cloverleaf is the **command** symbol, and represents the **Command key**, the key next to the space bar on the left side of your keyboard. Whenever that symbol appears in a menu, you don't have to chose the menu command with the mouse every time you want to give that command. You can hold down the Command key and type the given letter, instead. (Although the letter in the menu is usually capitalized, you can always use the lower or upper case letter.) The Command key combination always has the same effect as the corresponding menu command. Many commands have Command key equivalents; the equivalents always appear in the menu. For example, you can use Command-G in place of choosing the Go command from the Run menu.

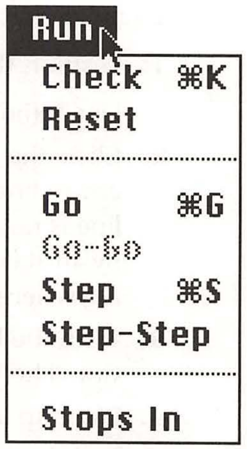


Figure 2-7 Command Symbols

As you get more comfortable with your Mac, you may find that using some Command key combinations is easier than pulling the menu down every time. The Command keys are also useful in situations like this one where you want to avoid pressing the mouse button.

The Step command can be given with Command-S. Use the Command-S combination to step through the program, and see how it loops and how it responds when you press the mouse button.

Alternately, you can use the **Step-Step** command from the Run menu. That command automatically executes the Step command repeatedly. The little hand moves quickly through the program as each statement is executed.

Giving Feedback

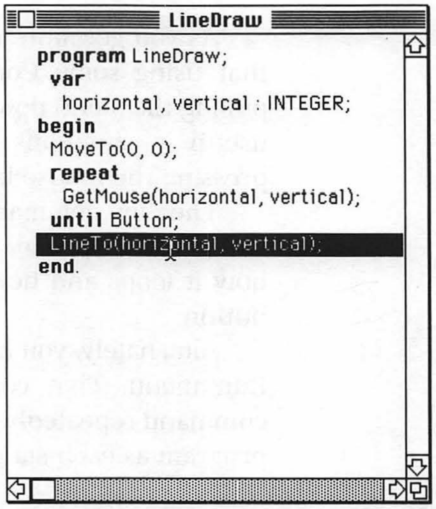
This program is better to users than the original *LineDraw*, but it could be improved. Good programs should give **feedback**. In other words, the program should give the user some idea of what is going on. In this case, it should show what the line would look like if the user pressed the button at that point.

The most obvious thing to do is to move the LineTo subprogram into the **repeat/until** loop.

To move that line:

1. Position the mouse pointer over the line:

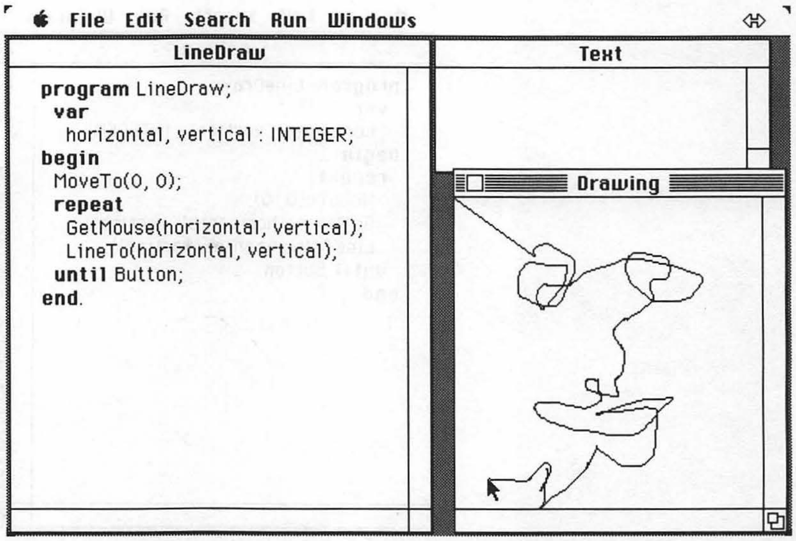
   LineTo(horizontal, vertical);
2. Click the mouse button three or more times quickly. The entire line should be selected, as shown in Figure 2-8. If the line is not selected, try again. (If you selected something else by mistake, deselect it by clicking the mouse button once anywhere.)
3. With the line selected, pull down the Edit menu, and choose Cut. The line disappears.
4. Place an insertion point immediately before the word **until**.
5. Pull down the Edit menu and choose Paste.



```
program LineDraw;
 var
  horizontal, vertical : INTEGER;
 begin
  MoveTo(0, 0);
  repeat
   GetMouse(horizontal, vertical);
  until Button;
  LineTo(horizontal, vertical);
 end.
```

Figure 2-8 Selecting the Entire Line

Your program should now be as shown in Figure 2-9. (If it doesn't look right, don't panic. If you don't know how to fix it, see the editing section of Chapter 1 for suggestions.)



```
program LineDraw;
 var
   horizontal, vertical : INTEGER;
begin
 MoveTo(0, 0);
 repeat
   GetMouse(horizontal, vertical);
   LineTo(horizontal, vertical);
 until Button;
end.
```

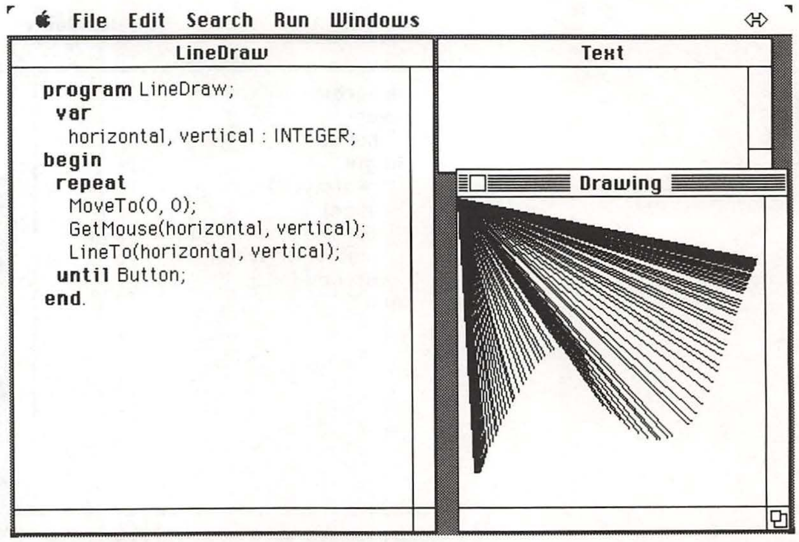
Figure 2-9 New LineDraw

Run that program. An example of what it produces is in Figure 2-9.

The result is interesting, but the program doesn't do what was required.

Do you see what is wrong? LineTo draws a line from the current pen position to the new coordinates. The pen is left at the new coordinates, so repeatedly calling LineTo makes the pen follow the mouse.

You need to move the line containing MoveTo(0,0) into the loop, so it gets repeated, also. That way, every time the program loops, the pen is moved to (0,0) and then draws a line to the mouse position. Your program should look like the program in Figure 2-10.



```
program LineDraw;
 var
  horizontal, vertical : INTEGER;
begin
 repeat
  MoveTo(0, 0);
  GetMouse(horizontal, vertical);
  LineTo(horizontal, vertical);
 until Button;
end.
```

Figure 2-10 New LineDraw

Run it a few times. A sample of this program's results is shown in Figure 2-10.

Although *LineDraw* now shows the results of a line from (0,0) to every position the mouse occupies, there is no way to distinguish the final line from any of the intermediate lines. You need to show what each line looks like, and then erase it. The next section shows you how to erase lines.

Changing the Mac's Pen

Something rather magical about the Mac's pen is that you can change the "ink" by just asking. The pen always starts out drawing thin, black lines, but you can change them to white, to gray, to thicker lines, or to lines that appear in patterns, like in MacPaint. You also can define how the "ink" is transferred to the screen. One way to erase a black line is to draw over it in white:

1. Put an insertion point before the word **repeat** and type:

   MoveTo(0,0);

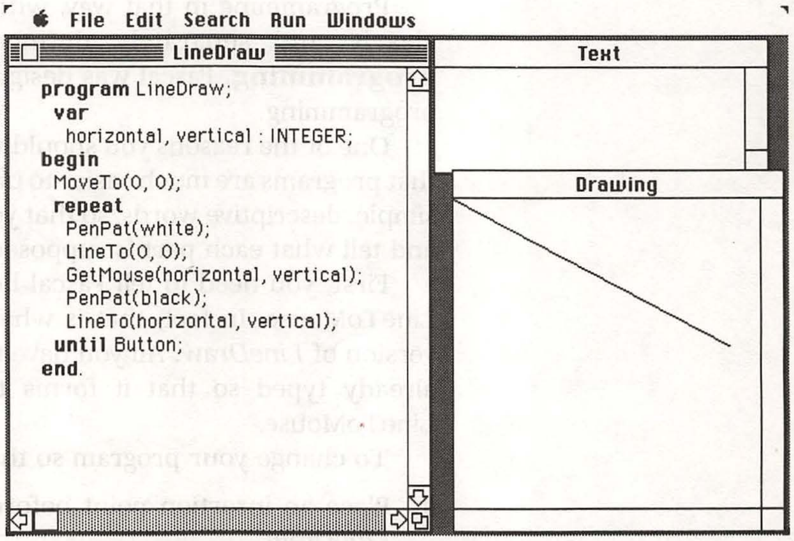
2. Place an insertion point before the second MoveTo and type:

   PenPat(white);

3. Replace the second MoveTo with LineTo.
4. Place an insertion point before the second LineTo and type:

```
PenPat(black);
```

Your program should now look like the one in Figure 2-11. When this program runs, the line is first drawn in black by LineTo(horizontal, vertical) and then drawn in white—effectively erased—by LineTo(0,0).



```
program LineDraw;
  var
   horizontal, vertical : INTEGER;
begin
 MoveTo(0, 0);
  repeat
   PenPat(white);
   LineTo(0, 0);
   GetMouse(horizontal, vertical);
   PenPat(black);
   LineTo(horizontal, vertical);
  until Button;
end.
```

Figure 2-11 New LineDraw

Run this program. A sample of what the program produces in the Drawing window is shown in Figure 2-11.

Writing Your Own Procedures

The way you can call complicated routines like **PenPat**, MoveTo, LineTo, and Button just by giving their names is an incredibly useful feature of the Pascal language. Take Button, for example. Although it represents a complicated subprogram, it has a simple name, and the name tells you very clearly and concisely what that routine does. In fact, it would be great if you could write *LineDraw* like that. Imagine if you could make *LineDraw* as simple as (don't type this):

```
begin
LineToMouse;
end.
```

Well, in fact, you can. You can make a procedure LineToMouse that goes off and draws the line, in the way that the predefined procedures LineTo, MoveTo, and PenPat go off and do their jobs.

Programming in that way, with every part of the program clearly and separately defined, is called **structured programming**. Pascal was designed to encourage structured programming.

One of the reasons you should write structured programs is that programs are much easier to understand if they are made of simple, descriptive words, so that you can just read the program and tell what each part is supposed to do.

First, you need to tell Pascal how to execute the statement LineToMouse. In fact, that is what you did before, in the last version of *LineDraw*. All you have to do is redefine what you've already typed so that it forms the body of the procedure LineToMouse.

To change your program so that it takes this new form:

1. Place an insertion point before **var**, in the second line of *LineDraw*.
2. Type:

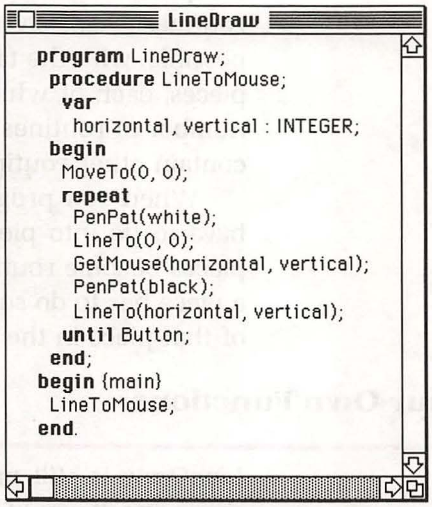
   ```
   procedure LineToMouse;
   ```

3. Replace the period after **end** with a semicolon.
4. After you type the semicolon, type:

   ```
   begin {main}
   LineToMouse;
   end.
   ```

   Be certain you typed the period after the new **end**. Also, be certain you typed curly brackets (the characters over the square brackets right of the "P" on your keyboard) around the word "main".

Your new program should look like the one in Figure 2-12.



```
program LineDraw;
  procedure LineToMouse;
  var
    horizontal, vertical : INTEGER;
  begin
    MoveTo(0, 0);
    repeat
      PenPat(white);
      LineTo(0, 0);
      GetMouse(horizontal, vertical);
      PenPat(black);
      LineTo(horizontal, vertical);
    until Button;
  end;
begin {main}
  LineToMouse;
end.
```

Figure 2-12 LineDraw with Procedure

Notes

All examples in this book include a blank line before every subprogram. Those empty lines are there solely for readability; blank lines have no meaning in the Pascal language. You can put them into your program, if you want, or leave them out.

The word "main" enclosed in curly brackets ({}) after the **begin** is a **comment**. You can put comments in your program to make the program more understandable. {main} marks **begin** as the start of the main part of the program. The new *LineDraw* works just like the old. Run it a few times to convince yourself. Try stepping through the program, either with Step-Step or with Command-S.

The part of the program between **begin** {main} and the last **end** is called the **main program**.

In a well written Pascal program, the main program should be as simple and clear as possible. All the difficult and complicated parts should be shuffled off to routines. The routines themselves should also be as simple and clear as possible, with the task ideally broken down into small, separate pieces, each of which does one thing. There is no limit to the number of routines you can use in a program, and routines can contain other routines.

When you program, you should first break whatever you have to do into pieces, and name each separate piece. Those pieces become routines that are used in your main program. If a piece has to do something complicated, you define the action of that piece in the same way, as if it were a main program.

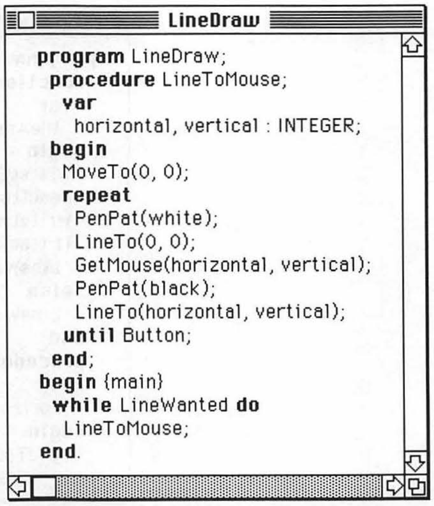
Writing Your Own Functions

LineDraw is still pretty limited. It only allows you to draw a single line. It would be better if it let you keep drawing lines until you didn't want to draw anymore. What you need is a TRUE/FALSE answer to the question "Do you want to draw another line?" Call the functional part that gives that answer: "LineWanted". If LineWanted is TRUE, another line should be drawn. If LineWanted is FALSE, the program should end.

Put an insertion point before LineToMouse in the main program (the part following **begin** {main}) and type:

```
while LineWanted do
```

Make sure there is no semicolon after the word "do". The program should now look like Figure 2-13.



```
program LineDraw;
 procedure LineToMouse;
  var
   horizontal, vertical : INTEGER;
 begin
  MoveTo(0, 0);
  repeat
   PenPat(white);
   LineTo(0, 0);
   GetMouse(horizontal, vertical);
   PenPat(black);
   LineTo(horizontal, vertical);
  until Button;
 end;
begin {main}
 while LineWanted do
  LineToMouse;
end.
```

Figure 2-13 LineDraw with While Statement

while (condition) **do** is a statement very much like **repeat/until** (condition). Both statements produce loops of statements that can be executed repeatedly.

The condition has to be something that is either TRUE or FALSE. As long as the condition is TRUE, the statement after the **do** is called repeatedly. As soon as the condition becomes FALSE, whatever follows the statement after the **do** is done.

In this case, when LineWanted is FALSE, the program stops because of the **end**. Figure 2-14 shows a diagram of what happens in this program.

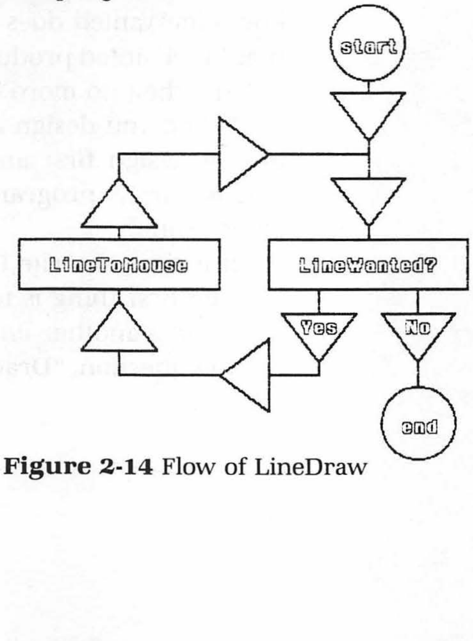
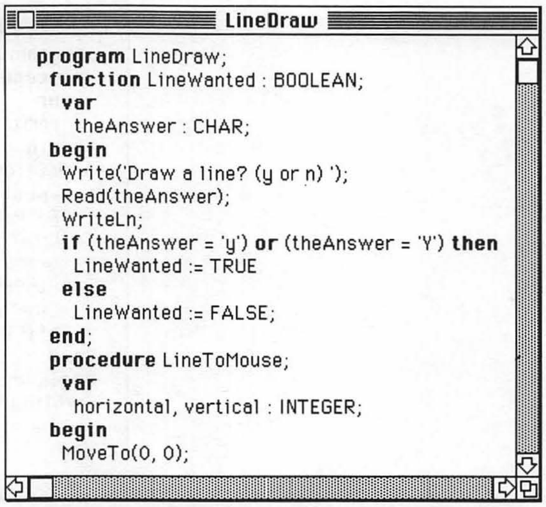


Figure 2-14 Flow of LineDraw



```
program LineDraw;
 function LineWanted : BOOLEAN;
  var
   theAnswer : CHAR;
 begin
  Write('Draw a line? (y or n) ');
  Read(theAnswer);
  WriteLn;
  if (theAnswer = 'y') or (theAnswer = 'Y') then
   LineWanted := TRUE
  else
   LineWanted := FALSE;
 end;
 procedure LineToMouse;
  var
   horizontal, vertical : INTEGER;
 begin
  MoveTo(0, 0);
```

Figure 2-15 Function LineWanted

The difference between **while** (condition) **do** and **repeat/until** (condition) is that the condition is checked at the beginning of the **while** loop, and at the end of the **repeat/until** loop. That means a **repeat/until** loop is always done at least once, even if the condition is always FALSE. A **while** loop is used here so that a line is never drawn when one is not wanted. (Programs should be obedient, as well as polite.)

LineWanted must be a function, like Button, because the loop depends on checking the value of LineWanted.

Notice that up to this point I've given no consideration to how LineWanted does what it is supposed to do. I just assume that LineWanted produces a TRUE when a line is wanted, and a FALSE when no more lines are wanted.

When you design a program, it is best to worry about the overall design first and the details later. If you worry about details first, programs tend to end up inefficient and error-prone.

It is time to write LineWanted.

The first thing is to decide how to find out if the program user wants another line. One way is to ask the user by printing out the question, "Draw a line?"

You can begin designing a subprogram like LineWanted by outlining what it must do. Here are the steps LineWanted must go through:

1. Ask "Draw a line?"
2. Get the answer.
3. If the answer is yes, set LineWanted to TRUE.
4. If the answer is no, set LineWanted to FALSE.
5. If you got some other answer, ask again.

Put an insertion point before the word "procedure" in the second line of the program. Type:

```
function LineWanted : BOOLEAN;
```

A BOOLEAN has either a TRUE or FALSE value. It can never have another value.

Type:

```
var
theAnswer: CHAR;
```

The variable *theAnswer* can hold single characters. CHAR is a predefined type. A CHAR value is any single character, such as letters, numbers, or special characters (*!@#$%^&*()__-+={}[]\|;:'" <,>.?/). CHAR values are always treated as **characters**. You can put a single digit (0, 1, 2, 3, 4, 5, 6, 7, 8, or 9) into a CHAR variable, but you can't do arithmetic with that character, without first changing it into an INTEGER or some other numerical type.

Type:

```
begin
Write ('Draw a line? (y or n) ');
Read(theAnswer);
WriteLn;
```

Write and WriteLn are predefined procedures that print in the **Text** window. The Text window is the smaller one in the upper right corner of the screen. (As with all windows, you can move the Text window around the screen, and can make it bigger or smaller. In any case, the Text window is the one that says "Text" in its title bar.)

Both Write and WriteLn display whatever is enclosed in parentheses. When WriteLn doesn't have any parentheses following it, it just moves the pen to the beginning of the next line. The "(y or n)" part of the message is to tell the user what should be typed in. Notice that the message is enclosed in single quotation marks. Macintosh Pascal prints what is within the quotes in the Text window. The quotation marks must be single quotation marks (' and '), not the double quotation marks (" and ") you usually see.

The **Read** procedure is a predefined procedure that reads something the user types in the Text window. It is impossible to type anything in the Text window unless the program first calls Read. Read reads enough to fill up whatever variables are in the parentheses. In this case, the variable *theAnswer* is of type CHAR, which can hold one character, so Read gets the one character and then returns that character to your program in answer.

Now that you have the user's decision about whether or not there should be another line drawn, you have to show the program how to deal with it. LineWanted is called as the condition in a **while** loop, so that the value of LineWanted determines whether or not the line is drawn. When you are inside a function, you give it a value simply by assigning the value to the function name, as if it were a variable. (In fact, you must assign the function name a value somewhere in every function; it is an error if you don't, because the value of the function is undefined.)

You need to tell the program "If *theAnswer* is Y, then LineWanted := TRUE, otherwise LineWanted := FALSE." You use a colon and a equals sign, ":=," which is typed with the two separate characters with no space in between to assign values in Pascal programs. It is important to remember that the := sign **assigns** values, unlike the mathematical = sign which expresses equality. You can use the := for statements like:

```
n := n+1
```

which adds 1 to the current value of *n* and assigns the result to *n*. The new value of *n* is equal to 1 plus the old value of *n*.

Before you type the next part of the program, you may want to make the programming window (the LineDraw window) a bit bigger, because the next line is too big to show entirely in the programming window Macintosh Pascal starts out with. The small box in the lower right corner of the window is the **size control box**. Put the pointer in the box and press the mouse button, and move the mouse to the right an inch or so.

Type:

```
if (theAnswer = 'y') or (theAnswer= 'Y') then
LineWanted := TRUE
else
LineWanted := FALSE;
end;
```

Function LineWanted should now look like the one in 2-15.

Notice that there is no semicolon after the first three lines you just typed. That is because the first four lines are all one statement: "If the condition is TRUE, then do this statement, else do this statement."

Before you run the program, if you enlarged the programming window so that it covers the Text and Drawing windows, uncover them, either by making the programming window smaller again or by clicking once in the Drawing window and once in the Text window, which brings them to the front.

Run the program a few times, to get a feel for it.

TRUE and FALSE Expressions

Notice the two sets of parentheses on the first line of the last section you typed:

```
(theAnswer = 'y') or (theAnswer= 'Y')
```

That line is a **Boolean expression**, also called a **logical expression**. BOOLEANs, as you may remember, are TRUE/FALSE values. So far, you have seen functions that return BOOLEAN values, and you have seen those values used in **while** and **repeat/until** statements. You can use Boolean expressions any place you can use a simple BOOLEAN value. A Boolean expression is a set of values that boil down to one TRUE/FALSE value. The expression

```
theAnswer = 'y'
```

is TRUE if the variable *theAnswer* contains a "y". The "y" given in the expression must be in quotation marks so Pascal can tell it is not a variable name. Notice that the Boolean expression uses a simple " = " sign, not ": =", which is used when you want to assign a value to a variable. The " = " alone means "is equal to," while the ": = " means "takes the value."

The second part of the expression

```
theAnswer = 'Y'
```

is there because "*theAnswer* = 'y'" is only TRUE if *theAnswer* contains a lower case "y". The user could easily enter an upper case "Y", as most human beings consider that pretty close to equivalent. It is a good practice to teach your Macintosh to anticipate the tastes and responses of humans.

The "**or**" between the two parenthetical expressions means that the value of both together is TRUE if either one or the other is TRUE (Figure 2-16).

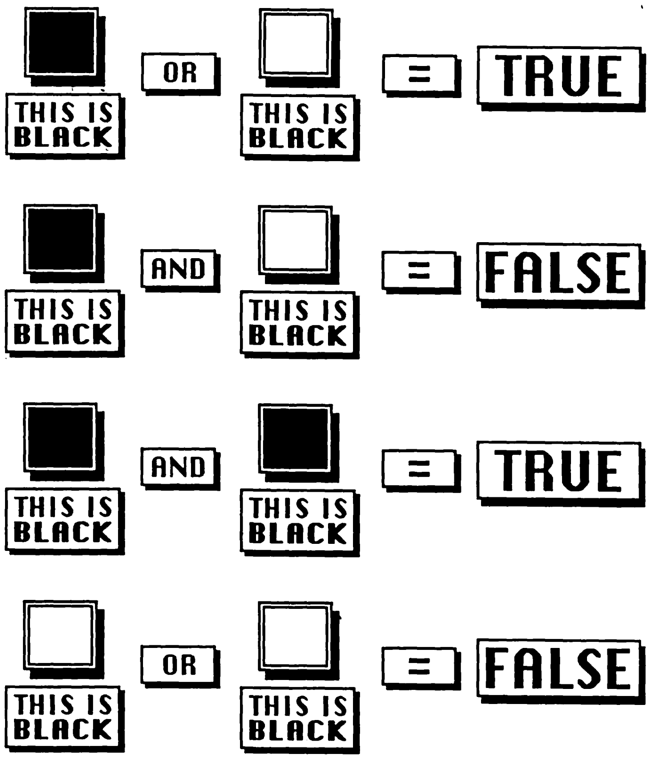


Figure 2-16 Boolean Operations

The **while** and **if** statements both check a condition and do a statement if the condition is TRUE. The **while** statement checks the condition and, if it is TRUE, does the statement after the **do**, and then loops back and checks the condition again. If the condition is still TRUE, the statement is done again, and so on. It can loop indefinitely, as long as the condition stays TRUE.

The **if** statement checks the condition and, if it is TRUE, does the statement after the **do**. Whether the condition was TRUE or FALSE, that statement is never done more than once, and the condition is only checked once.

A second difference between **while** and **if** is that **if** can have an **else** part. If the condition is FALSE, the statement following the **else** is executed.

So, if the user typed a "y" or "Y", LineWanted takes the value TRUE. If the user typed something else, LineWanted takes the value FALSE.

Do More

1. The program has one problem: It ends if the user types anything other than a "y" or "Y", even if it isn't an "n" or "N". Therefore, if the user makes a mistake, and types the wrong key, the program ends. That isn't too important in this little program, but it is sloppy. It violates the "Computers should be polite" rule. Try modifying LineWanted so it returns FALSE only if the user types an "n" or "N". Don't go too far in the other direction; a line should not be drawn unless the user types "y" or "Y".
2. Try modifying the program so that the user can type a number instead of a "Y", and then draw that number of lines before being asked if they want to draw another.
3. Computers sometimes seem to be able to think, the way a human being thinks. For the most part that is an illusion, carried out by clever programming, often using BOOLEANs. A Boolean expression allows a computer to "make a decision," based on factors determined by the programmer. In Pascal programs, you use **if**, **while**,

repeat, or **case** (sort of a multi-level **if**) statements to enable the computer to make decisions.

On paper, write a "pseudo-program" that would allow a robot to navigate a maze. (A pseudo-program is written to break a problem down into steps like in a program, and to follow a logical pattern that would work in a program, but without worrying about details of implementation. If you want the robot to go forward, for example, just say "robot forward" in the pseudo-program, and don't worry about how you would write a program to do that.)

QUICK SUMMARY

This chapter expands the LineDraw program produced in Chapter 1 so that it is a real program that responds to the user. It introduces subprograms, procedures and functions, variables and the use of the mouse and mouse button. The following routines, statements, commands, data types, and concepts are used.

BOOLEAN	is a predefined data type. A BOOLEAN value is either TRUE or FALSE.
Boolean expression	is a sequence of values, variables, functions, and operators that boil down to a single BOOLEAN value.
Button	is a predefined function. Button is TRUE when the mouse button is pressed. It is FALSE when the mouse button is not pressed.
Call	when referring to a subprogram, involves using the subprogram's name in a program. When you do that, the statements in the subprogram are executed.
CHAR	is a predefined data type. A CHAR value is a single character.
Command key	is the key with a cloverleaf symbol, to the left of the space bar.
Comment	is something enclosed in curly brackets ({}) in a program. Comments in programs are entirely ignored by Macintosh Pascal. You use them in programs for your convenience.

Data type	is a kind of value, such as an INTEGER, or a CHAR. You use data types to define the types of values that can be held by variables.
Function	is a subprogram that can be called by giving its name. A function is used in an expression, like a variable or constant, because it returns a value that is referenced by the function's name. When you give a function's name in a program, the statements in the function are executed.
Integer	is a whole number value.
INTEGER	is a predefined data type. An INTEGER value is a whole number from −32767 to 32767. Those limits result from the amount of space in memory that an INTEGER variable occupies.
GetMouse	is a predefined procedure that returns the coordinates of the mouse pointer.
Parameter	is a variable or value passed to or from a subprogram. See Formal parameter and Actual parameter in Chapter 3.
PenPat	is a predefined QuickDraw procedure that changes the color, or pattern, that the pen draws on the screen. To choose the pattern, you give PenPat one of a group of predefined variables: *white, black, gray, ltGray,* or *dkGray*. You can also define your own patterns.
Procedure	is a subprogram that can be called by giving its name. Calling a procedure is exactly equivalent to using the body of the procedure in place of the call. See Subprogram.
Read	is a predefined procedure that reads characters typed at the keyboard.
repeat	is a Pascal reserved word used to mark the beginning of a **repeat/until** loop. See **until** for more information.
Step	in reference to a program, is the command that executes one statement at a time.
Step-Step	is the command that executes the program by pausing briefly at each statement to display a small image of a hand next to the statement.
Subprogram	is a procedure or a function, a sequence of statements that can be called by giving the name of the subprogram. You can use subprograms to save space in programs, by placing sequences of statements that are repeated in your program into subprograms. You also use subprograms to make your programs readable and easier to fix and improve.
Routine	is another name for a subprogram.

until	is a Pascal reserved word that ends a **repeat/until** statement. **until** is followed by a Boolean expression. A **repeat/until** statement is a looping statement that normally contains other statements between **repeat** and **until**. The enclosed statements are repeated until the condition after the **until** becomes TRUE. Also see **while**.
Subroutine	is another name for a subprogram.
Variable	is a named item that can hold a value. Before you can use a variable, you must declare it and define its type in a **var** section.
while	is a Pascal reserved word that is used to define a looping statement like **repeat**, except that the condition is tested at the beginning of a **while** loop, and at the end of a **repeat** loop.
Write	is a predefined procedure that writes in the Text window.
WriteLn	is the same as write, except it moves the Text window's pen to the beginning of the next line after it prints out whatever is in its parameter list.

CHAPTER

3

Rectangles and Ovals

Drawing a Rectangle

It is time to abandon LineDraw and go on to two dimensions.

Most drawing on the Macintosh is done by the *QuickDraw* graphics package. QuickDraw is included in Macintosh Pascal's facilities.

LineTo and MoveTo are two of the routines defined in QuickDraw. QuickDraw provides many other routines, variables, and data types you can use in programs to draw in the Drawing window.

In particular, QuickDraw has a set of routines to draw various shapes. The most basic shape is a rectangle.

In QuickDraw, all rectangles are oriented so the top and the bottom are parallel to the top and bottom of the screen, and the sides are parallel to the sides of the screen. You define a rectangle by defining the positions of the four sides of the rectangle, as illustrated in Figure 3-1.

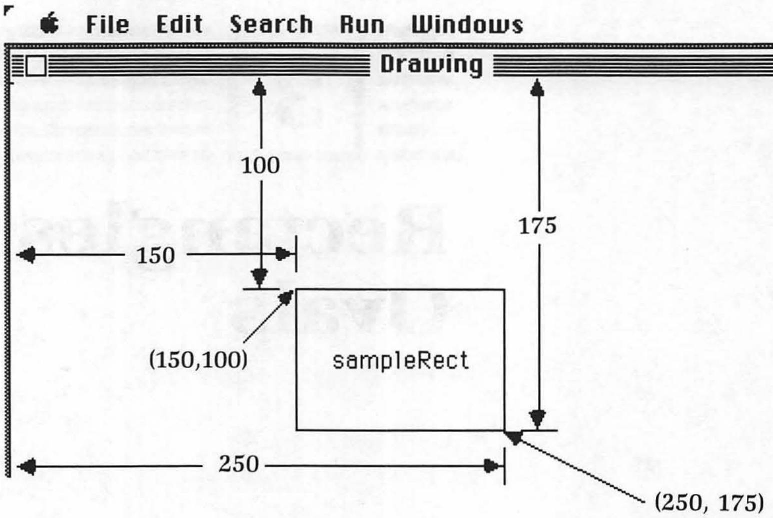


Figure 3-1 Defining a Rectangle

The first program in this chapter draws a rectangle on the screen. It uses the same basic approach as LineDraw. The program must:

1. Wait for the user to press the mouse button, and use that to define the top and left sides of the rectangle.
2. While the mouse button is still down, use the mouse position for the bottom and right sides of the rectangle, and draw the rectangle in black, so the user can see what the rectangle looks like at that point, and then quickly erases it by drawing it in white.
3. When the user releases the mouse button, draw the final rectangle.

This program gets the positions of the four sides of the rectangle from the user by reading the mouse position, as shown in Figure 3-2. The program must store those values in variables, so the rectangle can be drawn. You need four INTEGER variables to hold enough information to draw the rectangle.

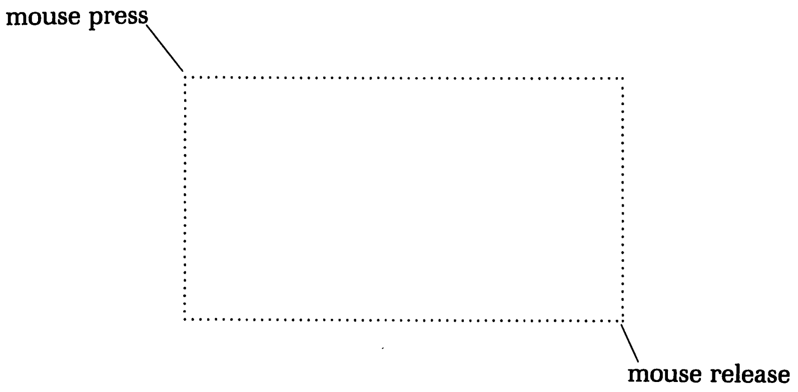


Figure 3-2 Defining a Box With the Mouse

The Pascal language lets you group variables together into **records**. You can think of a record as being a collection of variables, in the same way a house is a collection of rooms. Suppose a house has a living room, kitchen, and bedroom. All of these are in the house, but they are also individual rooms. You could identify the rooms like this:

house.livingRoom
house.kitchen
house.bedroom

The parts of a record are called **fields**. Each field is a variable by itself. A field has a name, just like any variable. You generally refer to a field of a record by giving the record's name, a period, and the field name (for example, *record.field*).

QuickDraw defines a record type called a RECT, which is intended for storing the numbers that define a rectangle. A RECT has four fields: top, left, bottom, and right, all of which are INTEGERs. Suppose, for example, you had a RECT called *theBox*. The four sides of *theBox* are *theBox.top*, *theBox,left*, *theBox.bottom*, and *theBox.right*. Aside from the fact that the four sides of *theBox* are all conveniently filed under one name, they act like other INTEGER variables. Figure 3-3 illustrates this idea.

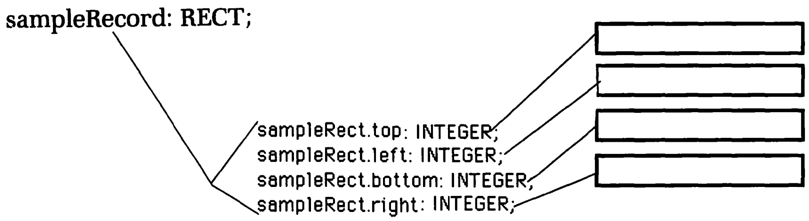

sampleRecord: RECT;
sampleRect.top: INTEGER;
sampleRect.left: INTEGER;
sampleRect.bottom: INTEGER;
sampleRect.right: INTEGER;

Figure 3-3 A RECT Variable

QuickDraw provides a number of procedures that let you deal with a RECT as a whole. For example, the procedure **FrameRect** takes a RECT variable and draws the outline of the rectangle. The procedure **EraseRect** takes a RECT variable and erases everything contained in that area. The procedure **SetRect** takes a RECT variable and four INTEGER values, and loads the four values into the RECT as the four sides of the rectangle.

Get a new program screen by:

- If you still have LineDraw displayed:

  1. Choose Close from the File menu.
  2. The Mac may beep and ask you if you want to save the latest changes. Click the mouse on Save.
  3. When the arrow pointer reappears, choose New from the File menu.

- If you have the desktop displayed, with the Macintosh Pascal disk inserted:

  1. Click once on the Macintosh Pascal icon. (You may have to open the Pascal disk icon first. See Chapter 1 if you don't know how to do that.)
  2. Choose Open from the File menu.
  3. When the Untitled program window appears, clear the program framework out of it as you did in Chapter 1.

In either case, you should now have a new Untitled Pascal programming window with nothing written in it.

Before typing anything:

1. Choose Save As from the File menu.
2. When the dialog box appears asking for a new file name, type:

   Shapes
3. Click in the Save button.

The programming window now says "Shapes" instead of "Untitled" in its title bar. While going through this chapter, if you want to stop for a while, choose Quit from the File menu. The Mac asks you if you want to save your changes. Choose Yes. Then, when you want to come back to where you left off, open the Shapes icon.

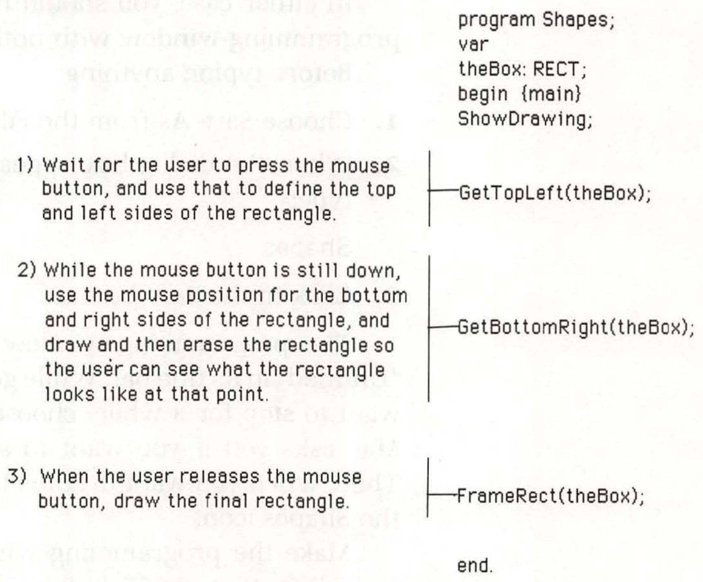
Make the programming window wider. This program has some lines that are fairly long.

Type the following program, which follows the steps given above. The parts in curly brackets ({}); are comments inserted to make the program more understandable. It is a good idea to include comments in programs, but they are just for the convenience of people. The computer ignores them.

Type:

```
program Shapes;
var
theBox: RECT;
begin {main}
ShowDrawing; {Predefined. Brings Drawing window to front.}
GetTopLeft(theBox);
GetBottomRight(theBox);
FrameRect(theBox); {Predefined. Outlines rectangle.}
end.
```

Don't run this yet. This is the main **control structure** of the program—these lines determine the sequence of actions in the program, but you still have to define GetTopLeft and GetBottomRight, which actually do the work. Figure 3-4 shows how the main program uses the plan given at the beginning of the chapter.



```
program Shapes;
var
theBox: RECT;
begin {main}
ShowDrawing;
```

1) Wait for the user to press the mouse button, and use that to define the top and left sides of the rectangle.

```
GetTopLeft(theBox);
```

2) While the mouse button is still down, use the mouse position for the bottom and right sides of the rectangle, and draw and then erase the rectangle so the user can see what the rectangle looks like at that point.

```
GetBottomRight(theBox);
```

3) When the user releases the mouse button, draw the final rectangle.

```
FrameRect(theBox);

end.
```

Figure 3-4 Program Plan and Program Code

ShowDrawing is a predefined routine that moves the Drawing window to the front. Because you may often enlarge the programming window to give you room to type your programs, it is a good idea to begin programs that use the Drawing window with this procedure call. That way you don't have to remember to make the Drawing window visible before you run the program.

The procedure GetTopLeft gets the top and left sides of the rectangle *theBox*. In the same way, GetBottomRight gets the bottom and right sides of the rectangle.

The predefined QuickDraw procedure FrameRect outlines the rectangle defined by *theBox*. There is no need to use MoveTo to move the pen before calling FrameRect. The position of the rectangle on the screen is completely defined by the variable *theBox*.

You now need to type the procedures GetTopLeft and GetBottomRight. These procedures work much like the one used in procedure LineToMouse in the *LineDraw* program.

GetTopLeft must:

1. Wait for the user to press the mouse button.
2. Store the mouse position.

   Place an insertion point before the **begin.** Type:

```
procedure GetTopLeft(var theBox: RECT);
begin
repeat
until Button;
GetMouse(theBox.left, theBox.top); {Gets left and top of box.}
end;
```

Look at the first line of the procedure. Following the procedure name is a set of parentheses, enclosing the reserved word **var** and a variable declaration:

```
procedure GetTopLeft(var theBox : RECT);
```

That is the **formal parameter list** of procedure GetTopLeft.

All subprograms, whether procedures or functions, can have parameters. The parameters are used to get values from the calling program to the subprogram, and to send values back from the subprogram to the calling program. That process, of sending values to or from the subprogram, is called **passing** values.

A formal parameter is a variable, that may get an initial value passed in when the procedure is called and that may pass its value back when the procedure finishes.

A subprogram can have any number of parameters. You declare all the subprogram's parameters in the formal parameter list just after the subprogram's name. When you call the subprogram, you give a list of variables and values, which are called the **actual parameters** of the subprogram. The actual parameters are associated with the formal parameters that are in the same position in the formal parameter list. For example:

Formal parameter list:

```
procedure ShowHow(x, y, z : INTEGER);
```

Actual parameter list:

```
ShowHow(100, 11, anIntVar);
```

The formal parameter list appears at the beginning of the subprogram itself; an actual parameter list occurs whenever the subprogram is called.

Every parameter declaration in the parameter list must be separated from the parameter declaration that follows by a semicolon. As with variables, you can declare any number of parameters of the same type by giving a list, each one separated by a comma. For example:

```
procedure ShowHow(x, y, z: INTEGER; var aBool:BOOLEAN);
```

Notes

Macintosh Pascal rearranges all formal parameter lists so only one declaration appears on a line. As usual in Pascal programs, blank lines and spaces have no meaning, and are only for the convenience of human readers.

There are two kinds of parameters: **variable parameters**, like the one in GetTopLeft, and **value parameters**. You define parameters in the same way you define variables: with the parameter's name (or a list of parameter names), a colon (:), and a data type.

Every variable parameter is preceded by the reserved word **var**. Value parameters are not preceded by **var**. The first parameter in this formal list is a variable parameter, while the second is a value parameter:

```
procedure ShowHow(var x : INTEGER; y : INTEGER);
```

In the next example, there are two value parameters, followed by two variable parameters:

```
procedure ShowHow(x, y : INTEGER; var w, z :INTEGER);
```

Within subprograms, variable and value parameters both act like ordinary variables. The difference is that a variable parameter is actually the same variable as the one given in the corresponding position when the subprogram was called, although it may have a different name. Whatever you do to the value of the variable parameter in the subprogram also changes the value of the variable in the calling program. Therefore, you use variable parameters when you want to pass a value back to the main program.

With value parameters, an entirely different variable is created for the subprogram. The subprogram's value parameter variable is initially assigned the value given in the corresponding position of the subprogram call.

Getting back to GetTopLeft, the procedure loops through the **repeat/until** loop until the user presses the mouse button. The mouse position values returned by GetMouse are then shoved into the *theBox.right* and *theBox.top*.

You may notice that the same name, *theBox*, is used for the rectangle in the main program and in the procedure GetTopLeft. This is not required. You could just as easily name the rectangle anything you like within the procedure GetTopLeft.

GetBottomRight uses the same approach, but it needs to do a couple of additional steps. It repeats the following until the user releases the mouse button:

1. Get the mouse position.
2. Draw the rectangle.
3. Erase the rectangle.

These steps are very much like what was used in LineToMouse, except that a rectangle is drawn instead of a single line, and you don't need to change the pen pattern because the EraseRect routine is available.

Place an insertion point before **begin** {main}. Type:

```
procedure GetBottomRight(var theBox: RECT);
begin
repeat
GetMouse(theBox.right, theBox.bottom);
FrameRect(theBox);
EraseRect(theBox);
until not Button;
end;
```

The program should look like the one in Figure 3-5.

```
program Shapes;
 var
  theBox : RECT;
 procedure GetTopLeft (var theBox : RECT);
 begin
  repeat
  until Button;
  GetMouse(theBox.left, theBox.top); {Gets left and top of box.}
 end;

 procedure GetBottomRight (var theBox : RECT);
 begin
  repeat
   Getmouse(theBox.right, theBox.bottom);
   FrameRect(theBox);
   EraseRect(theBox);
  until not Button;
 end;

begin {main}
 ShowDrawing; {Pre-defined. Brings Drawing window to front.}
 GetTopLeft(theBox);
 GetBottomRight(theBox);
 FrameRect(theBox); {Pre-defined. Outlines rectangle.}
end.
```

Figure 3-5 Program Shapes

One line in that procedure which may look odd to you is:

until not Button;

not is a logical operator that reverses the BOOLEAN value of whatever follows. In other words, if the statement following is TRUE, **not** makes it FALSE; if the statement following is FALSE, **not** makes it TRUE. **repeat/until** loops until the condition becomes TRUE. In this case, the program must loop until the user is no longer holding the mouse button down, i.e., when Button returns FALSE. The value of the logical expression

not Button

is FALSE when the mouse button is down, and becomes TRUE when the mouse button is up.

Notes

At this point, you can't see all of the text in the program at once, because the program has too many lines to fit in the window. You can **scroll** the window by using the **scroll bar** on the right side of the window. Click in the arrows at the top and bottom of the scroll bar to scroll the window up and down a line at a time. You can also use the mouse to click and drag the small rectangle that is inside the scroll bar. When you move that box, called the **elevator**, the document scrolls to the corresponding position. Another choice is to click in the gray part of the scroll bar. That scrolls the program to the next full window of text. You can also change the type size, so that more of the program is visible in the window. Choose the **Type Size**... option from the Windows menu. (You can also use that option to get larger type, if you find that the ordinary size strains your eyes.)

Run the program. To use it, put the mouse pointer somewhere in the Drawing window, and press the mouse button. Hold the button down and move the mouse down and to the right. A shadowy box appears. When you let go of the button, the box is drawn in solid lines, and the program ends.

Try running it with the Step-Step command from the Run menu, to see how the program responds to the mouse button.

RECT variables are also used to define ovals. An oval is drawn so it fits inside the rectangle defined by the RECT, as shown in Figure 3-6. The **FrameOval** call works the same as FrameRect. It has one parameter, like FrameRect, and the parameter must be a RECT. You can substitute the word FrameOval for FrameRect in the program, and it will draw ovals in place of rectangles.

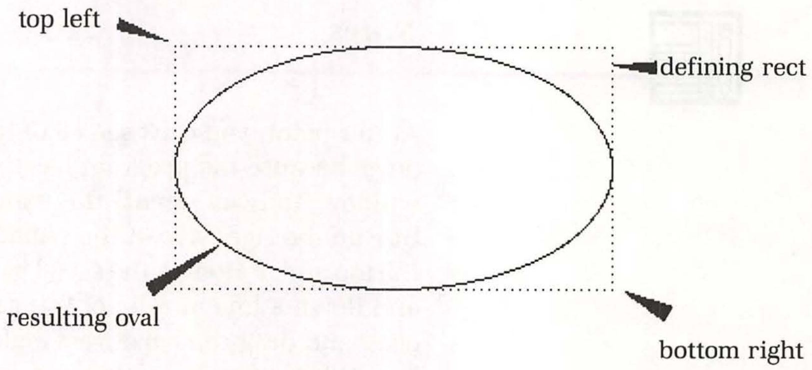


Figure 3-6 Defining an Oval With a RECT

Drawing More Shapes

The rest of this chapter builds the *Shapes* program into a small version of MacPaint.

It is easy enough to write little programs that draw one kind of shape, like the version of *Shapes* you have already used. It is not that much harder to make a program that allows the user to choose which kind of shape is drawn, the kind of lines used to draw the shape, and the fill pattern.

The hardest part of making this "mini-Paint" program is finding out what the program user wants to draw.

You have seen, in the last version of *LineDraw*, how WriteLn and Read can be used to find out what the user wants. That is how you normally communicate with the user in Pascal; the Macintosh is a graphics- and mouse-based computer, though. When you use MacPaint, you choose what shapes you want to draw with **onscreen buttons**. Onscreen buttons are used to take the place of physical buttons or function keys. When the program user clicks the mouse button within the borders of an onscreen button, the program carries out the action corresponding to that button.

Buttons are small outlined areas on the screen, often containing some identifying text. When you choose the Save command, for example, the box that asks you for the new file name has two buttons: Save and Cancel.

To use on-screen buttons in *Shapes* the program must:

1. Define the sizes and positions of the buttons on the screen, and draw them, complete with identifying text.
2. Repeat the following until a valid button is chosen.
   a. Wait for the user to press the mouse button.
   b. Get the mouse position.
   c. Check each onscreen button to see if the mouse is in that button. If the pointer is in a button, invert the image of the button. Wait for the mouse button to come up, then restore the onscreen button to its original appearance.

3. When a choice has been made, go off and do it.

The first improved version of *Shapes* has only two buttons: one that allows you to draw a rectangle, and one that stops the program. Later in this chapter the program is expanded so that there are a number of other choices. The basic program is the same, though. Notice that the program design given above makes no mention of a number of allowable choices. It is written to be general enough to allow expansion of the resulting program.

Before you begin typing in the new parts of the program, follow these steps to convert the original program *Shapes* to a procedure DrawRectangle:

1. If you changed the FrameRect calls to FrameOval calls in your program, change them back to FrameRect.
2. Remove the lines:

```
var
theBox : RECT;
```

3. Place an insertion point before **begin** {main} and type:

```
procedure DrawRectangle;
var
theBox : RECT;
```

4. Remove the comment {main} after **begin**.
5. Change the period at the end of the program to a semicolon.

You should now have three procedures, as shown in Figure 3-7. These procedures are now ready to be used in a new version of *Shapes*.

```
program Shapes;

 procedure GetTopLeft (var theBox : RECT);
 begin
  repeat
  until Button;
  GetMouse(theBox.left, theBox.top); {Gets left and top of box.}
 end;

 procedure GetBottomRight (var theBox : RECT);
 begin
  repeat
   GetMouse(theBox.right, theBox.bottom);
   FrameRect(theBox);
   EraseRect(theBox);
  until not Button;
 end;

 procedure DrawRectangle;
  var
   theBox : RECT;
 begin
  ShowDrawing; {Pre-defined. Brings Drawing window to front.}
  GetTopLeft(theBox);
  GetBottomRight(theBox);
  FrameRect(theBox); {Pre-defined. Outlines rectangle.}
 end;
```

Figure 3-7

This new program produces buttons that are all the same size. It is a good practice to collect fixed values, such as the sizes of buttons, in the beginning of the program, where they can be easily found and changed. Fixed values are called **constants**. Constants can have names that represent them, the same way variables have names. Unlike variables, constant values cannot change while the program is running. The advantage of using named constants instead of using the numbers directly in the program is that, if you later want to change the value of some constant, you don't have to search for every place you used it.

Place an insertion point before procedure GetTopLeft.

Type the following.

```
const
NUMBUTS = 2; {The number of buttons.}
BUTHEIGHT=20; {The height of each button, in pixels.}
BUTLEFT = 20; {The left side of all buttons.}
BUTRIGHT = 60; {The right side of all buttons.}
SPACE = 5; {Used to space button labels.}
```

I capitalize all my named constants everywhere they are used, so that they stand out. As with variables, you can use upper or lower case characters in constant names; BUTHEIGHT is the same as butheight.

The rules for constant names are the same as the rules for program and variable names: they can contain up to 255 letters, numbers, and underscores.

The insertion point should now still be just before procedure GetTopLeft. Type:

```
type
choices = array[1..NUMBUTS] of RECT; {The limits of each
button.}
```

The word "**type**", like **const** and **var**, names a section of the program, called a **declaration part**.

The **type** declaration part contains definitions of data types. Up to this point, you have used predefined data types including the simple types INTEGER, BOOLEAN, and CHAR and the structured data type RECT. You can declare your own data types in a huge variety of ways in Pascal programs.

An **array**, like a record, contains a number of parts, each of which is a variable by itself. The difference between an array and a record is that every part of an array has the same data type, and array parts, called elements, are identified by numbers in square brackets ([ ]) instead of field names following a period. If a record is like a house that contains several rooms, each identified by a name, an array is like a street lined with identical houses, each of which is identified by a number.

You declare a new type by giving the type name, an equal sign, and the type definition. You end the type definition with a semicolon. For example:

```
showHowType = array[1..10] of INTEGERS;
```

The numbers in square brackets define how many elements the array has and what their names are. An array of type *showHowType* has ten elements, each of which is an INTEGER. No array of this type exists until you declare one in a **var** declaration part or in the formal parameter list of a subprogram. Suppose the program's **var** part contains this declaration:

```
showHow : showHowType;
```

When the program starts running, an array *showHow* is created according to the plan specified in the declaration of *showHowType*. It has these ten elements:

showHow[1]
showHow[2]
showHow[3]
showHow[4]
showHow[5]
showHow[6]
showHow[7]
showHow[8]
showHow[9]
showHow[10]

The rules for type names are the same as the rules for program, variable, and constant names: they can contain up to 255 letters, numbers, and underscores.

The **type** declarations of a program must come before the **var** declarations of the program, if there are any. In any case, the **type** declarations must come before any functions, procedures, and the main program. Subprograms can also have their own **type** declarations.

The **const** declarations must come before the **type** and **var** declarations, if there are any, and also before any functions, procedures, and the main program. Subprograms can also have their own **const** declarations.

Once you've defined a data type, you can create a variable of that type in the same way you create a variable of a predefined type.

So, Choices is a new data type. It defines a group of RECTs. Notice the part of the definition [1..NUMBUTS], which defines the number of elements in the array. NUMBUTS is the constant that indicates the number of buttons. At the moment its value is 2. Figure 3-8 gives a representation of a Choices-type array. Note that Choices is *not* a variable, but it is a new *type* of variable. You can define a variable of type Choices in the same way you defined variables of type INTEGER, CHAR, BOOLEAN, or RECT. Pascal then sets up a storage area for that variable, a set of "file drawers" fitting the description given in the type definition.

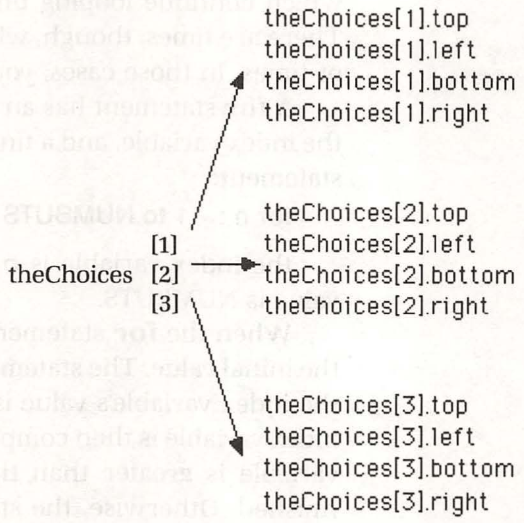


Figure 3-8 Choices-type Array

It is now time to define the variables for this program. Place an insertion point before procedure GetTopLeft and type:

```
var
theChoices : choices;
stopProgram : BOOLEAN; (When TRUE, the program
stops.)
oldDrawingRect : RECT;
whichBut : INTEGER; (The button that was pressed.)
```

Don't worry about these variables for now. They are explained later.

The insertion point should still be before GetTopLeft. Type:

```
procedure DrawButtons (var theChoices : choices);
var
n : INTEGER;
begin
for n := 1 to NUMBUTS do
```

The **for** statement is a type of looping statement.

So far you have seen the **repeat/until** and **while** loops, which continue looping until some condition becomes FALSE. There are times, though, when you need to loop a fixed number of times. In those cases, you use the **for** statement.

A **for** statement has an **index variable**, an initial value for the index variable, and a final value for the index variable. In the statement:

for n := 1 **to** NUMBUTS **do**

the index variable is *n*, the initial value is 1, and the final value is NUMBUTS.

When the **for** statement begins, the index variable is set to the initial value. The statement following the **do** is executed, and the index variable's value is increased by one. The value of the index variable is then compared with the final value. If the index variable is greater than the final value, the **for** statement is finished. Otherwise, the statement after the **do** is done again, the index variable's value is increased and compared with the final value, and so on, until the value of the index variable is greater than the final value.

This **for** loop is equivalent to:

```
n := 1;
while n <= NUMBUTS do
n := n + 1;
```

The symbols "< =" mean "less than or equal to."

Very often you need to have more than one statement in a **for** loop. Pascal allows you to use a **compound statement** any place you can use a single statement. A compound statement is any group of statements surrounded by **begin** and **end**.

The insertion point should be right after the "do" you just typed. Type:

```
begin
SetRect(theChoices[n], BUTLEFT,n * BUTHEIGHT, BUTRIGHT,
    (n+1) * BUTHEIGHT);
FrameRect(theChoices[n]);
MoveTo(BUTLEFT + SPACE,(n + 1) * BUTHEIGHT-SPACE);
```

Those statements draw the buttons on the screen. SetRect, as mentioned before, loads a RECT variable with the four sides of the rectangle. FrameRect draws that button's outline. MoveTo moves the pen into position to write the text that identifies that button.

Notice the way the index variable *n* is used in those statements. The most common use of **for** loops in Pascal programs is to go through an array and do something to each element. Suppose you have an array called *anArray,* which has ten elements numbered one through ten. Suppose further that you want to assign each element the value 25. You could give ten separate statements like this:

```
anArray[1] := 25;
anArray[2] := 25;
:
```

Because array elements are numbered, you can use a **for** loop to repeatedly act on each element. For example:

```
for n := 1 to 10 do
anArray[n]:= 25;
```

The program uses the same same sort of approach.

You also have to put a piece of text in each button that identifies the button. That means you have to do something different depending on the value of the index variable *n.*

You can do that using **if** statements (don't type this):

```
if (n = 1) then WriteDraw('Box');
if (n = 2) then WriteDraw('Stop');
```

(**WriteDraw** is a predefined procedure much like WriteLn, except it writes in the Drawing window and is defined in QuickDraw.) That would be perfectly fine, but there is an easier way. Type:

```
case n of
1: WriteDraw('Box');
2: WriteDraw('Stop');
end; {ends case}
end; {ends for loop}
end; {ends DrawButtons}
```

The action of a **case** statement is exactly like a sequence of **if** statements, except that it involves less typing.

The **case** statement compares the value of *n*, called the **selector** to each value given on the left, called **case constants**. When a match is found, the statement following the colon (:) is executed. **case** statements, like **if** statements, are used to make decisions. While an **if** statement has only two choices (either do something or don't do it), a **case** statement can have as many choices as you like.

The way this procedure is written, when you want to add a new button, all you have to do is increase the value of NUMBUTS and add a new case constant for the new button, along with a WriteDraw statement giving the text that appears in the button. The button is then drawn automatically.

The next procedure sets up the screen for this program.

This program needs to have the drawing window expanded so that it fills the whole screen, so that you have room to draw. Windows are defined in terms of RECTs. You can call the predefined procedure SetDrawingRect to change the Drawing window to a new size and position. That has the same effect as changing the size and position of the Drawing window with the mouse. In this case, the window is expanded to fill the screen. The predefined procedure GetDrawingRect is also called in this procedure. That gets the initial size of the Drawing window, so that you can put it back the way it was it when the program is finished.

The insertion point should still be before procedure GetTopLeft. Type:

```
procedure SetUp (var oldDrawingRect: RECT; var theChoices:
choices);
var
tempBox: RECT; {Holds new Drawing window size.}
begin
GetDrawingRect(oldDrawingRect);
SetRect(tempBox, 0, 40, 520, 350);
SetDrawingRect(tempBox);
ShowDrawing;
DrawButtons(theChoices);
end;
```

The next procedure waits for the user to "press" a button, and returns the number of the button picked. The basis of the procedure is a predefined QuickDraw function PtInRect, which returns TRUE if a point in the Drawing window is within a given rectangle. You give a variable of type RECT to define the rectangle, and a variable of type POINT to define the point on the screen. POINT is a QuickDraw-defined **record** that has two INTEGER fields: *v*, which contains a vertical coordinate and *h*, which contains a horizontal coordinate. The procedure goes through the following steps:

1. Reset the variable *whichBut* that indicates the choice.
2. Wait for the mouse button to be pressed.
3. Get the mouse position.
4. Go through the **array** of buttons (theChoices) with a **for** loop and use PtInRect to find out if the mouse pointer is in one of the buttons.
5. If the mouse pointer is in a button, invert the image of the button with the QuickDraw procedure InvertRect. That procedure makes every black pixel in the RECT white, and every white pixel black. Also, set *whichBut* to the index number of the button containing the mouse pointer. Then, wait for the user to let go of the mouseButton. When the user lets go of the mouse button, call InvertRect again, to restore the button to its original appearance, and go on.
6. When the **for** loop finishes, this procedure returns to the main program.

If the user clicks the mouse button with the mouse inside a button, this procedure returns the index number of that button in the array *theChoices*. If the user clicks the button outside one of the buttons, this procedure returns a 0 as the index number.

The insertion point should still be before procedure GetTopLeft. Type:

```
procedure PickedOne(var theChoices : choices; var whichBut:
INTEGER);
var
n: INTEGER;
thePoint: POINT;
begin
whichBut:= 0; {Resets this value.}
repeat {Loops here until the mouse button is pressed.}
until Button;
GetMouse(thePoint.h, thePoint.v);
for n := 1 to NUMBUTS do
if PtInRect(thePoint, theChoices[n]) then
begin
InvertRect(theChoices[n]);
whichBut := n;
repeat
until not Button;
InvertRect(theChoices[n]);
end; {if}
end; {picked}
```

The final part of the program is the main program.

The main program must:

1. Set the BOOLEAN stopProgram to FALSE. This is set to TRUE when the user picks the Stop button from the screen.
2. Set up the screen and buttons. (Call procedure SetUp.)
3. Loop through the following until *stopProgram* is TRUE:
   a. Call PickedOne to get *whichBut*.
   b. Either do nothing (if *whichBut* is 0), call DrawRectangle (if *whichBut* is 1) or set *stopProgram* to TRUE, if *whichBut* is 2.

4. Finally, after *stopProgram* becomes TRUE, restore the Drawing window to its original size.

Make an insertion point past the end of the program, after the last **end** (the one for procedure DrawRectangle). Type:

```
begin {main}
stopProgram := FALSE;
SetUp(oldDrawingRect, theChoices);
repeat
PickedOne(theChoices, whichBut);
case whichBut of
0:; {Empty statement.}
1: DrawRectangle;
2: stopProgram := TRUE;
end;
until stopProgram;
SetDrawingRect(oldDrawingRect);
end.
```

Choose Save from the File menu to save this version of the program.

The program should now look like the program in Figure 3-9.

```
program Shapes;
 const
  NUMBUTS = 2;  {The number of buttons.}
  BUTHEIGHT = 20; {The height of each button, in pixels.}
  BUTLEFT = 20;   {The left side of all buttons.}
  BUTRIGHT = 60;  {The right side of all buttons.}
  SPACE = 5;      {Used to space button labels.}
 type
  choices = array[1..NUMBUTS] of RECT; {The limits of each button.}
 var
  theChoices : choices;
  stopProgram : BOOLEAN;  {When TRUE, the program stops.}
  oldDrawingRect : RECT;
  whichBut : INTEGER;  {The button that was pressed.}

 procedure DrawButtons (var theChoices : choices);
  var
   n : INTEGER;
 begin
  for n := 1 to NUMBUTS do
   begin
```

```
      SetRect(theChoices[n], BUTLEFT, n * BUTHEIGHT, BUTRIGHT, (n + 1) * BUTHEIGHT);
      FrameRect(theChoices[n]);
      MoveTo(BUTLEFT + SPACE, (n + 1) * BUTHEIGHT - SPACE);
      case n of
       1 :
        WriteDraw('Box');
       2 :
        WriteDraw('Stop');
      end;  {ends case}
     end;  {ends for loop}
end;  {ends DrawButtons}

procedure SetUp (var oldDrawingRect : RECT;
          var theChoices : choices);
 var
   tempBox : RECT; {Holds new Drawing window size.}
begin
 GetDrawingRect(oldDrawingRect);
 SetRect(tempBox, 0, 40, 520, 350);
 SetDrawingRect(tempBox);
 ShowDrawing;
 DrawButtons(theChoices);
end;

procedure PickedOne (var theChoices : choices;
          var whichBut : INTEGER);
 var
   n : INTEGER;
   thePoint : POINT;
begin
 whichBut := 0; {Resets this value.}
 repeat {Loops here until the mouse button is pressed.}
 until Button;
 GetMouse(thePoint.h, thePoint.v);
 for n := 1 to NUMBUTS do
   if PtInRect(thePoint, theChoices[n]) then
     begin
      InvertRect(theChoices[n]);
      whichBut := n;
      repeat
      until not Button;
      InvertRect(theChoices[n]);
     end; {if}
end; {picked}
```

```
procedure GetTopLeft (var theBox : RECT);
begin
 repeat
 until Button;
 GetMouse(theBox.left, theBox.top); {Gets left and top of box.}
end;

procedure GetBottomRight (var theBox : RECT);
begin
 repeat
  GetMouse(theBox.right, theBox.bottom);
  FrameRect(theBox);
  EraseRect(theBox);
 until not Button;
end;

procedure DrawRectangle;
 var
  theBox : RECT;
begin
  ShowDrawing;  {Pre-defined. Brings Drawing window to front.}
  GetTopLeft(theBox);
  GetBottomRight(theBox);
  FrameRect(theBox); {Pre-defined. Outlines rectangle.}
 end;

begin  {main}
 stopProgram := FALSE;
 SetUp(oldDrawingRect, theChoices);
 repeat
  PickedOne(theChoices, whichBut);
  case whichBut of
   0 :
    ;  {Empty statement.}
   1 :
    DrawRectangle;
   2 :
    stopProgram := TRUE;
  end;
 until stopProgram;
 SetDrawingRect(oldDrawingRect);
end.
```

Figure 3-9

Notice that the first case constant, 0, is followed by a semicolon. That creates an empty statement, so nothing is done if *whichBut* is equal to 0. You must have a case constant for every value the selector might have. In this program, *whichBut* can be equal to 0, 1, or 2.

(Actually, although a case constant must exist for every possible value of the selector in a **case** statement in standard Pascal, you can end a **case** statement in a Macintosh Pascal program with a case selector of **otherwise**, which is chosen if none of the case constants match the selector.)

Run the program a few times.

To use it:

1. Click in the box button.
2. Move the mouse pointer to the place where you want the upper left corner of the box to appear.
3. Press the mouse button, and hold it down while moving the mouse down and to the right.
4. Release the mouse button when you have reached the spot you've chosen for the bottom right corner.
5. Click in a button for another box, or click in the Stop button.

If you want to add new buttons, all you have to do is :

1. Increase the value of the constant NUMBUTS.
2. Add a new case constant to the **case** statement in DrawButtons with some text for the button.
3. Write a procedure that does whatever that button is supposed to do.
4. Add a new case constant and procedure call to the **case** statement in the main program.

The buttons are displayed in numerical order, so that the button represented by the case constant 1 (as well as the position in the array *theChoices*[1]) is the top button. If you add a new button, you may want to move the Stop button down to the bottom, so your new choice is invoked by the case constant 2, and stopped by the case constant 3.

As an example, the following section adds ovals to the shapes that can be draw by this program. Follow these steps:

1. At the beginning of the program, change the 2 following NUMBUTS to 3, so that line is:

   NUMBUTS = 3; {The number of buttons.}

2. Place an insertion point after the "2:" in procedure DrawButtons and type:

   ```
   WriteDraw('Oval');
   3:
   ```

3. Place an insertion point before the line **begin** {main} and type:

   ```
   procedure DrawOval;
   var
   size : RECT;
   begin
   GetTopLeft(size);
   GetBottomRight(size);
   FrameOval(size);
   end;
   ```

4. Place an insertion point after the "2:" in the **case** statement in the main program and type:

   ```
   DrawOval;
   3:
   ```

5. Choose Save from the File menu.

   The program should now look like the one in Figure 3-10.

```
program Shapes;
 const
  NUMBUTS = 3;  {The number of buttons.}
  BUTHEIGHT = 20; {The height of each button, in pixels.}
  BUTLEFT = 20;   {The left side of all buttons.}
  BUTRIGHT = 60;  {The right side of all buttons.}
  SPACE = 5;      {Used to space button labels.}
 type
  choices = array[1..NUMBUTS] of RECT; {The limits of each button.}
 var
  theChoices : choices;
  stopProgram : BOOLEAN;  {When TRUE, the program stops.}
  oldDrawingRect : RECT;
  whichBut : INTEGER;  {The button that was pressed.}
```

```
procedure DrawButtons (var theChoices : choices);
 var
  n : INTEGER;
begin
 for n := 1 to NUMBUTS do
  begin
   SetRect(theChoices[n], BUTLEFT, n * BUTHEIGHT, BUTRIGHT, (n + 1) * BUTHEIGHT);
   FrameRect(theChoices[n]);
   MoveTo(BUTLEFT + SPACE, (n + 1) * BUTHEIGHT - SPACE);
   case n of
    1 :
     WriteDraw('Box');
    2 :
     WriteDraw('Oval');
    3 :
     WriteDraw('Stop');
   end;  {ends case}
  end;  {ends for loop}
end;  {ends DrawButtons}

procedure SetUp (var oldDrawingRect : RECT;
          var theChoices : choices);
 var
  tempBox : RECT; {Holds new Drawing window size.}
begin
 GetDrawingRect(oldDrawingRect);
 SetRect(tempBox, 0, 40, 520, 350);
 SetDrawingRect(tempBox);
 ShowDrawing;
 DrawButtons(theChoices);
end;

procedure PickedOne (var theChoices : choices;
          var whichBut : INTEGER);
 var
  n : INTEGER;
  thePoint : POINT;
begin
 whichBut := 0; {Resets this value.}
 repeat {Loops here until the mouse button is pressed.}
 until Button;
 GetMouse(thePoint.h, thePoint.v);
 for n := 1 to NUMBUTS do
  if PtInRect(thePoint, theChoices[n]) then
   begin
```

```
     InvertRect(theChoices[n]);
     whichBut := n;
     repeat
     until not Button;
     InvertRect(theChoices[n]);
    end; {if}
end; {picked}

procedure GetTopLeft (var theBox : RECT);
begin
 repeat
 until Button;
 GetMouse(theBox.left, theBox.top); {Gets left and top of box.}
end;

procedure GetBottomRight (var theBox : RECT);
begin
 repeat
  GetMouse(theBox.right, theBox.bottom);
  FrameRect(theBox);
  EraseRect(theBox);
 until not Button;
end;

procedure DrawRectangle;
 var
  theBox : RECT;
begin
 ShowDrawing; {Pre-defined. Brings Drawing window to front.}
 GetTopLeft(theBox);
 GetBottomRight(theBox);
 FrameRect(theBox); {Pre-defined. Outlines rectangle.}
end;

procedure DrawOval;
 var
  size : RECT;
begin
 GetTopLeft(size);
 GetBottomRight(size);
 FrameOval(size);
```

```
end;
begin {main}
  stopProgram := FALSE;
  SetUp(oldDrawingRect, theChoices);
  repeat
   PickedOne(theChoices, whichBut);
   case whichBut of
    0 :
     ; {Empty statement.}
    1 :
     DrawRectangle;
    2 :
     DrawOval;
    3 :
     stopProgram := TRUE;
   end;
  until stopProgram;
  SetDrawingRect(oldDrawingRect);
end.
```

Figure 3-10

Notice another advantage of writing small modules that do simple things: the DrawOval procedure takes advantage of the work you already did writing DrawRectangle, and calls the same procedures to get the top left corner and the bottom right corner.

Run the program. This time you are presented with two types of shapes that you can draw: ovals and boxes. Use this program as you used the last version.

Shapes is used for the rest of this chapter to explore some of the capabilities of QuickDraw.

All About the Macintosh Pen

I mentioned before that you can change the "ink" in the Macintosh "pen" by simply asking. The Mac pen has a set of characteristics, any of which can be changed. These are stored in a record of type PENSTATE. A PENSTATE has the following five fields:

- *pnLoc,* which is of the type POINT. This holds the location of the pen. You change this value by using MoveTo or LineTo, or any other procedure that draws on the screen.

- *pnSize,* which is also a POINT. This is the height and width of the pen. Although a POINT is normally used to store a location, it can actually hold any two INTEGER values. In this case the two numbers have nothing to do with a position on the screen, but determine the width and height in pixels of a line drawn by the pen. You use the procedure PenSize to change this value.
- *pnPat,* which is of the type PATTERN. (You don't need to worry about the details of this QuickDraw data type for now. Chapter 9 uses it in more detail.) If you have used MacPaint, you are familiar with a wide variety of pen patterns. Each of the "paint" choices at the bottom of the screen is actually a different pen pattern. So far, you have only used the normal pen pattern, black, in this book. Actually, though, the pen pattern is a 8–by–8 block of pixels, any of which can be black or white. When you draw a line, it is drawn using the current pen pattern. You change the pen pattern with PenPat, as you did in Chapter 2. You can give one of the standard QuickDraw patterns (*black, white, gray, ltGray,* and *dkGray*) or you can define your own pattern.
- *pnMode,* which is an INTEGER. When you draw a line, the pnMode determines exactly how the pen pattern is appears on the screen. The normal mode is ***patCopy***, which does the simplest thing: it copies the pen pattern to the screen. Because the pen starts out with a black pattern and a mode of *patCopy,* you get simple, black lines. In other modes, QuickDraw compares the pixel that is already on the screen and the corresponding pixel in the pen pattern. The pen mode determines a logical operation that is used in comparing the two pixels. In this logical comparison, a black pixel is equivalent to TRUE, and a white pixel is equivalent to FALSE. If the result of the comparison is black, the pixel on the screen becomes black. Here are descriptions of the three main logical operations, AND, OR, and XOR. You have seen AND and OR before, in Chapter 2.

1. (conditionOne AND conditionTwo) is TRUE only if both conditions are TRUE. If either is FALSE, the result is FALSE.
2. (conditionOne OR conditionTwo) is TRUE if either one is TRUE or if both are TRUE. The result is FALSE only if both conditions are FALSE.

3. (conditionOne XOR conditionTwo) is TRUE if only one of the conditions is TRUE. If both conditions are TRUE or if both conditions are FALSE, the result is FALSE. XOR is short for exclusive or.

Extending that concept to black and white pixels, the result of pixelOne XOR pixelTwo is black if only one of them is black to begin with. If both are black, the result is white. If both are white, the result is white. If only one is black, the result is black. This mode is often used because it is fully reversible. If you use ***patXOr*** on a portion of the screen once, and then on the same portion again with the same PenPat, the result looks exactly like it did before you did anything.

There are eight pen modes on the Mac. They are described in the QuickDraw appendix of the manual that came with Macintosh Pascal.

You can extend the program *Shapes* to change any of these pen characteristics.

The following additions to *Shapes* add buttons to make the Pen's line wider or thinner.

1. Change the number following NUMBUTS to 5. That line should now be:

```
NUMBUTS = 5; {The number of buttons.}
```

2. Place an insertion point after the "3:" in procedure DrawButtons and type:

```
WriteDraw('Thick');
4: WriteDraw('Thin');
5:
```

3. Place an insertion point before **begin** {main} and type:

```
procedure DemoPen;
var
tempBox : RECT;
begin
SetRect(tempBox, BUTLEFT, (NUMBUTS+2)*BUTHEIGHT,
    BUTRIGHT, (NUMBUTS+4)*BUTHEIGHT);
EraseRect(tempBox);
FrameRect(tempBox);
end;
```

This procedure demonstrates the way the pen is currently set. One of the rules of good Macintosh programming is that you should always let your user know what is going on. This procedure draws a little sample below the buttons, so that user can see the current pen size. It begins by creating a RECT that is calculated to fall below all of the buttons. It then calls the QuickDraw procedure EraseRect to erase that section of the screen, so that whatever is there already is removed. Then an oval is drawn.

4. Type the routine that makes the line thicker. It gets the current pen state by calling the predefined procedure GetPenState, and then adds one pixel to the pen's width and height. It then calls PenSize to change the pen's size.

   Place an insertion point before **begin** {main} and type:

```
procedure ThickLine;
var
oldState : PENSTATE;
width, height : INTEGER;
begin
GetPenState(oldState);
width := oldState.pnSize.h + 1; {Thickens pen 1 pixel in width.}
height := oldState.pnSize.v + 1; {Thickens 1 pixel in height.}
PenSize(width, height);
DemoPen;
end;
```

5. Type the routine that makes the line thinner. It uses the same kind of logic as ThickLine.

   The insertion point should still be right before **begin** {main}. Type:

```
procedure ThinLine;
var
oldState : PENSTATE;
width, height : INTEGER;
begin
GetPenState(oldState);
width := oldState.pnSize.h – 1;
height := oldState.pnSize.v – 1;
PenSize(width, height);
DemoPen;
end;
```

6. Alter the main program to use the new routines.
Place an insertion point before the **repeat** and type:

DemoPen;

Place an insertion point after the "3 :" and type:

ThickLine;
4 : ThinLine;
5 :

7. Choose Save from the File menu to save the current version of this program.

The three new procedures and the altered main program are shown in Figure 3-11.

```
procedure DemoPen;
 var
  tempBox : RECT;
begin
 SetRect(tempBox, BUTLEFT, (NUMBUTS + 2) * BUTHEIGHT, BUTRIGHT, (NUMBUTS + 4) *
      BUTHEIGHT);
 EraseRect(tempBox);
 FrameRect(tempBox);
end;

procedure ThickLine;
 var
  oldState : PENSTATE;
  width, height : INTEGER;
begin
 GetPenState(oldState);
 width := oldState.pnSize.h + 1; {Thickens pen 1 pixel in width.}
 height := oldState.pnSize.v + 1; {Thickens 1 pixel in height.}
 PenSize(width, height);
 DemoPen;
end;

procedure ThinLine;
 var
  oldState : PENSTATE;
  width, height : INTEGER;
```

```
begin
 GetPenState(oldState);
 width := oldState.pnSize.h - 1;
 height := oldState.pnSize.v - 1;
 PenSize(width, height);
 DemoPen;
end;

begin {main}
 stopProgram := FALSE;
 SetUp(oldDrawingRect, theChoices);
 DemoPen;
 repeat
  PickedOne(theChoices, whichBut);
  case whichBut of
   0 :
    ;  {Empty statement.}

   1 :
    DrawRectangle;
   2 :
    DrawOval;
   3 :
    ThickLine;
   4 :
    ThinLine;
   5 :
    stopProgram := TRUE;
  end;
 until stopProgram;
 SetDrawingRect(oldDrawingRect);
end.
```

Figure 3-11

Run this program.

You use the program in a way much like the way you used the previous versions of *Shapes*. You can now change the thickness of the lines that are drawn by using the Thick and Thin buttons. Click on the Thick buttons a couple of times. Then click in one of the shape buttons (box or oval) and draw a shape.

You might want to try drawing something—a face for example.

Do More

When you make the changes suggested in this section, you may need help getting your program to run. Problems in programs are called "bugs," and the process of finding the bugs and fixing them is called "debugging." Macintosh Pascal gives you some capabilities that make debugging pretty easy. If you need help debugging your program, read the debugging chapter of this book.

1. Try adding a new button to change the pen mode. Use the PenMode QuickDraw procedure to change the mode.

You call PenMode like this:

```
PenMode(theMode);
```

The value *theMode* given above should be one of these mode constants:

- *patCopy*, which is the normal mode. The pattern is copied directly to the screen.
- *patXOr*, which performs an exclusive OR operation on the pixels in the pattern and on the screen. If both are black or white, the resulting pixel is white. If one pixel is black and one is white, the resulting pixel is black.
- ***patOr***, which performs an OR operation on the pixels in the pattern and on the screen. If one or both of the pixels is black, the resulting pixel is black.
- ***patBic*** turns all pixels that are black in the pattern white on the screen. This is essentially erase mode.

You can also add the prefix "not" to each of these, to form one word. These four additional modes are: ***notPatCopy***, ***notPatOr***, ***notPatXOr***, and ***notPatBic***. These reverse all the pixels in the pattern, and then perform the transfer operation.

2. Add another button that changes the pen pattern. Use the PenPat procedure to change the pen pattern. Call PenPat like this:

   ```
   PenPat(thePattern);
   ```

   The placeholder *thePattern* can be one of the predefined pattern variables: *white, black, gray, ltGray,* or *dkGray*. You can also create your own patterns. The way to do that is described in Chapter 9, Advanced QuickDraw.

   You should show some feedback so that the user can see the state of the pen. Change DemoPen so it demonstrates the pen's mode as well as its pattern and thickness. You should call DemoPen after changing the pen, so the new pen settings are demonstrated.

3. Try altering your pen mode or pen pattern button so a single button can handle all the possible choices. You can have the choice determined by how long the mouse button is held down, or by how many mouse clicks there are, or by which part of the button the mouse is in when it is clicked.

4. Try adding some MacPaint features, such as mirrors, which creates a mirror image of what the user is drawing.

QUICK SUMMARY

Chapter 3 explores the QuickDraw graphics package. It discusses how to draw rectangles and ovals, the details of the Macintosh's electronic pen, and how to use onscreen buttons to get information from the program user. It also discusses **arrays**, **case** statements, and how good program structure makes a program easy to understand and easy to improve or extend. The following statements, routines, and concepts are introduced.

Actual parameter	is a value or variable given in a call to a subprogram.
array	is a reserved word used to define a variable made up of a numbered set of identical elements. You define an array by giving the limits of the indexes of the array, enclosed in square brackets and separated by two periods. You refer to a specific element of the array by giving the array name, followed by the element number enclosed in square brackets.
case	is a reserved word used for a compound statement that takes different actions based on the value of a selector. The **case** statement has a list of case constants. When the value of the selector matches the value of one of the case constants, the statement following that case constant is executed.
const	is a reserved word used in a program or subprogram to name the declaration part that defines named constants. The **const** declaration part must come before the **var**, **type**, procedure, and function declarations, and before the main statement part of the program.
Field	of a record is one of the parts of the record. See **record** in this section for more information.
for	is a reserved word used to create a loop that loops for a specified number of times. An index variable is updated every time the loop loops.
Formal parameter	is a special kind of variable that is defined in the first line of a subprogram. There are two kinds of formal parameters: value parameters and variable parameters. You give the reserved word var before declaring a variable parameter. Use variable parameters when you want to pass a value back to the calling program.
not	is a reserved word used as a logical operator that reverses the BOOLEAN value of the following Boolean expression.

Parameter	of a subprogram is a variable used to get information when the subprogram is called, or to give information back when the subprogram finishes. There are two kinds of parameters: value parameters and variable parameters. A value parameter cannot pass values back to the calling program, while a variable parameter always does.
PenMode	is a QuickDraw procedure that changes the way the current pen pattern is transferred to the screen.
POINT	is a QuickDraw record data type that defines a point of the screen. A POINT has two INTEGER fields: *v* and *h*.
QuickDraw	is a library of graphics subprograms that are used to draw lines, shapes, and text on the Macintosh screen. Although you can call it from Pascal programs, it is written in Assembler, the basic instruction code of the Mac's processor, and thus works very quickly and efficiently. QuickDraw is documented in the *Macintosh Pascal Technical Appendix*.
record	is a reserved word used to define a variable that is a collection of other variables, each of which has a field name. You refer to a field by giving the record name, a period, and the field name. The fields of records can be of any type. Although Macintosh Pascal itself does not contain any predefined record types, the QuickDraw package defines POINT and RECT.
RECT	is a QuickDraw record data type. A RECT has INTEGER fields for the four sides of a rectangle: top, left, bottom, and right. You can also access the RECT through fields that define the top left and bottom right corners or the rectangle: *topLeft* and *botRight*.
Rectangle	is the basic shape for much of QuickDraw's drawing. A rectangle can be defined by the predefined type RECT.
type	is a reserved word used in a program or subprogram to name the declaration part that defines new data types. The **type** declaration part must appear before the **var** declaration part and statement part of the program.

CHAPTER

4

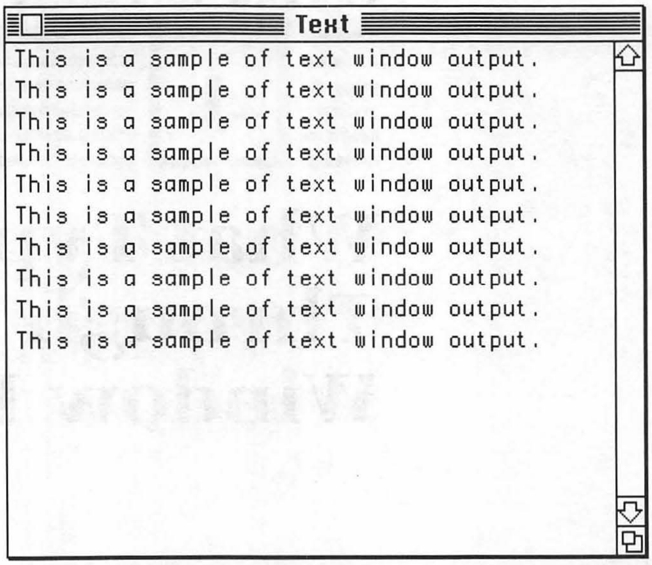
What Type Through Yonder Window Breaks

Macintosh Text

A picture may be worth a thousand words, but you more often need to store words in your computer.

This chapter shows you how to get text from *here* to *there* and back again. More particularly, it shows how to get text to the Macintosh screen, and how to make it look the way you want it to look.

Programs in this chapter print text in the Drawing window rather than in the Text window. The reason is that you have a great deal of control over the way text shows in the Drawing window and very little over how it appears in the Text window. The Text window always prints in fixed, evenly spaced lines. Figure 4-1 gives a sample of text in the Text window.



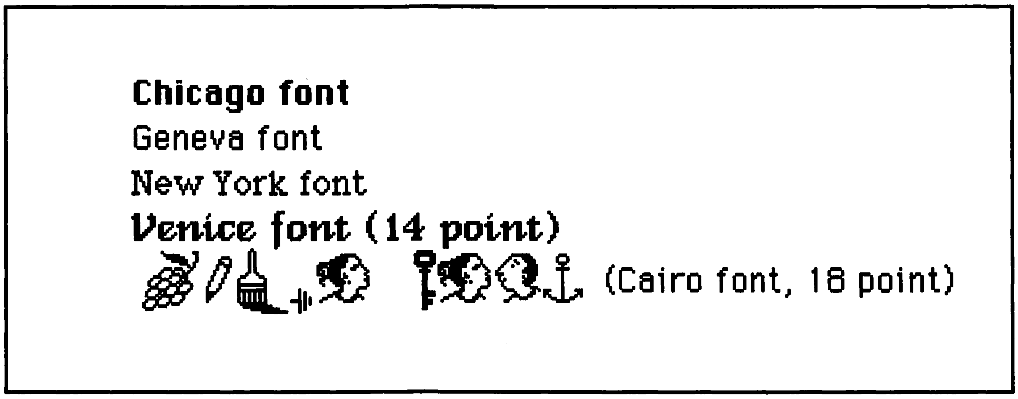
```
This is a sample of text window output.
This is a sample of text window output.
This is a sample of text window output.
This is a sample of text window output.
This is a sample of text window output.
This is a sample of text window output.
This is a sample of text window output.
This is a sample of text window output.
This is a sample of text window output.
This is a sample of text window output.
```

Figure 4-1 Sample Text Window Type

In the Drawing window, on the other hand, you have almost total control over the appearance of text.

Text on the Macintosh has the following characteristics. You can change any of them when you draw text in the Drawing window.

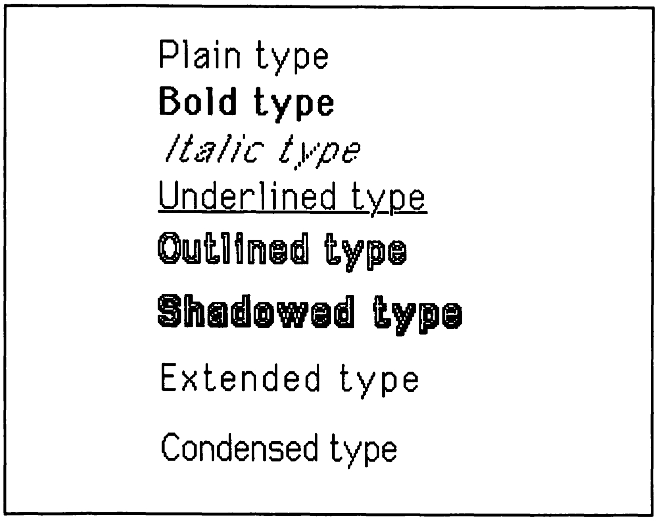
- **Font**. A font is a set of type defining what each letter, number, and character looks like in a particular style. Macintosh Pascal uses the Geneva font by default, but the Macintosh has a number of other fonts. There are four on the Macintosh Pascal disk. Fonts generally contain the different styles of normal English characters, but they actually can contain any set of images. The Macintosh Cairo font is an example of a non-character font. Figure 4-2 shows some sample fonts.
- **Face**. The font characters can be modified with a set of face characteristics: *bold, italic, underline, outline,* and *shadow*, all of which may be familiar to you if you've used MacWrite or MacPaint. There are also two rarely used characteristics: *condense* and *extend*. All of these are described more fully in the section on face characteristics later in this chapter. Figure 4-3 demonstrates the effect of the face characteristics on the Geneva font.



Chicago font
Geneva font
New York font
Venice font (14 point)
(Cairo font, 18 point)

Figure 4-2 Sample Fonts

- **Size**. Standard type size on the Macintosh is 12 **points**. A point is a typographic measurement that is about 1/72 of an inch, about the size of a pixel. You can tell the Mac to draw characters of any size. Figure 4-4 shows the Geneva font in a few different styles.



Plain type
Bold type
Italic type
Underlined type
Outlined type
Shadowed type
Extended type
Condensed type

Figure 4-3 Face Characteristics

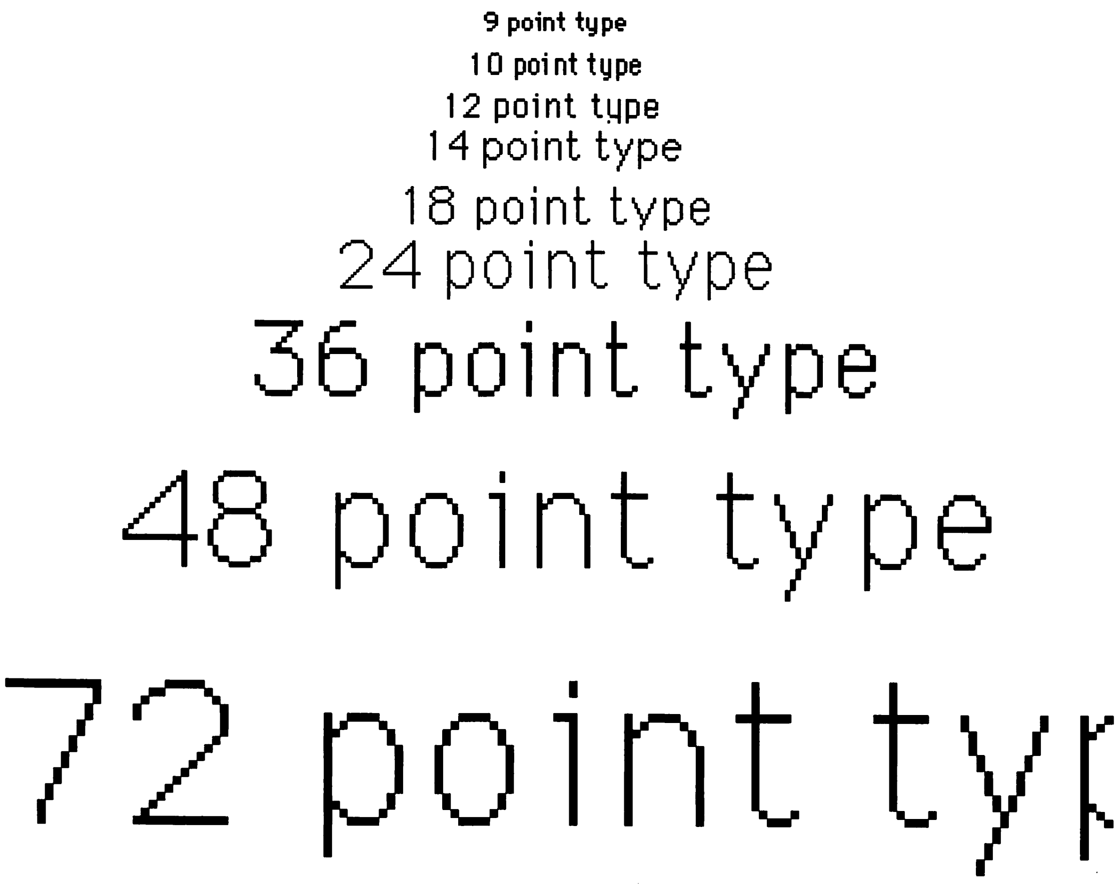
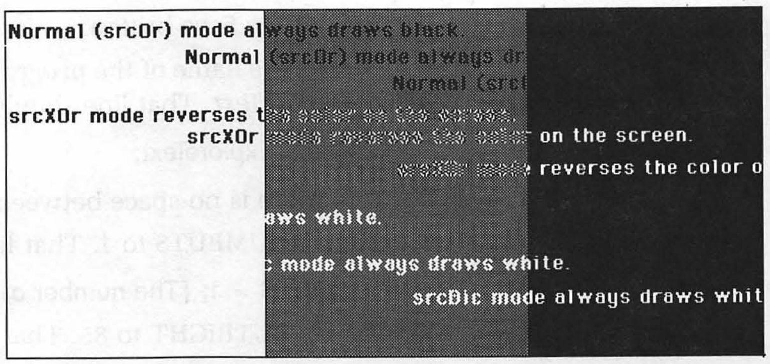


Figure 4-4 Sample Type Sizes

- **Transfer Mode**. Drawing text in QuickDraw is the same as drawing anything. You can set a transfer mode that compares what is already on the screen with what you are drawing on the screen, and uses some logical operation to determine what is actually drawn. That way, for example, you can make sure text is still visible even though part of the screen is already painted black. Figure 4-5 demonstates the three text modes.



```
Normal (srcOr) mode always draws black.
                  Normal (srcOr) mode always dr
                                      Normal (srcO
srcXOr mode reverses the color on the screen.
                  srcXOr mode reverses the color on the screen.
                                      srcXOr mode reverses the color o

                    aws white.

                    c mode always draws white.
                                      srcBic mode always draws whit
```

Figure 4-5 Text Modes

At any time, each of these characteristics has one setting. You can change the settings by using calls to QuickDraw.

In certain applications, such as MacPaint and MacWrite, you can change some of these characteristics (font, face, and size) with menu commands. Actually, those applications call QuickDraw to change the settings.

When you tell QuickDraw to print text, it uses the current settings to determine how to draw the text.

The program *Shapes*, produced in the last chapter, has a fine skeleton for trying out text characteristics. This chapter begins with a section on each characteristic given above. Later on, the chapter goes into how you get text from the user, rather than what it looks like. The next chapter shows the various things you can do with text once you've collected it.

Before you go through these sections, you should prepare a special stripped-down version of program *Shapes*, which is used in the following sections to explore QuickDraw's type capabilities.

1. Open program *Shapes*, if it is not already opened.
2. Choose Save As... from the File menu.
3. When you are asked "Save your program as," type:

   Explore Text

4. Click in the Save button.
5. Change the name of the program, the word after **program**, to *ExploreText*. That line should now be:

```
program ExploreText;
```

   Note there is no space between Explore and Text.
6. Change NUMBUTS to 1. That line should now be:

```
NUMBUTS = 1; {The number of buttons.}
```

7. Change BUTRIGHT to 85. That line should now be:

```
BUTRIGHT = 85; {The right side of all buttons.}
```

8. Remove everything from just after the end of procedure PickedOne (the beginning of procedure GetTopLeft) to just before **begin** {main}.
9. Delete the body of the **case** statement in procedure DrawButtons except for the "1 :" and "WriteDraw('Stop');". The **case** statement in DrawButtons should look like:

```
case n of
1:
WriteDraw('Stop');
end;
```

10. In the **case** statement in the main program, remove everything between "1:" and "stopProgram:= TRUE;". The **case** statement in the main program should look like:

```
case n of
0:
; {Empty Statement.}
1:
stopProgram := TRUE;
end;
```

11. From the main program, delete the line:

```
DemoPen;
```

12. Place an insertion point before **begin** {main} and type:

```
procedure WriteSample;
var
h, v : INTEGER;
begin
repeat
until Button;
GetMouse(h, v);
MoveTo(h, v);
WriteDraw('A rose is a rose');
end;
```

This procedure places a sample string of text at the position where you press the mouse button. It is called to demonstrate text after you have used the onscreen buttons to change text characteristics.

13. Place an insertion point after "0:"in the main program and type:

```
WriteSample
```

14. Remove the comment:

```
{Empty statement.}
```

15. Choose Save from the File menu.

The program should now look like the program in Figure 4-6.

```
program ExploreText;
 const
  NUMBUTS = 1;  {The number of buttons.}
  BUTHEIGHT = 20; {The height of each button, in pixels.}
  BUTLEFT = 20;   {The left side of all buttons.}
  BUTRIGHT = 85;  {The right side of all buttons.}
  SPACE = 5;      {Used to space button labels.}
 type
  choices = array[1..NUMBUTS] of RECT; {The limits of each button.}
 var
  theChoices : choices;
  stopProgram : BOOLEAN;  {When TRUE, the program stops.}
  oldDrawingRect : RECT;
  whichBut : INTEGER; {The button that was pressed.}
```

```
procedure DrawButtons (var theChoices : choices);
 var
  n : INTEGER;
begin
 for n := 1 to NUMBUTS do
  begin
   SetRect(theChoices[n], BUTLEFT, n * BUTHEIGHT, BUTRIGHT, (n + 1) * BUTHEIGHT);
   FrameRect(theChoices[n]);
   MoveTo(BUTLEFT + SPACE, (n + 1) * BUTHEIGHT - SPACE);
   case n of
    1 :
     WriteDraw('Stop');
   end;  {ends case}
  end;  {ends for loop}
end;  {ends DrawButtons}

procedure SetUp (var oldDrawingRect : RECT;
         var theChoices : choices);
 var
  tempBox : RECT; {Holds new Drawing window size.}
begin
 GetDrawingRect(oldDrawingRect);
 SetRect(tempBox, 0, 40, 520, 350);
 SetDrawingRect(tempBox);
 ShowDrawing;
 DrawButtons(theChoices);
end;
procedure PickedOne (var theChoices : choices;
         var whichBut : INTEGER);
 var
  n : INTEGER;
  thePoint : POINT;
begin
 whichBut := 0; {Resets this value.}
 repeat {Loops here until the mouse button is pressed.}
 until Button;
 GetMouse(thePoint.h, thePoint.v);
 for n := 1 to NUMBUTS do
  if PtInRect(thePoint, theChoices[n]) then
   begin
    InvertRect(theChoices[n]);
    whichBut := n;
    repeat
    until not Button;
    InvertRect(theChoices[n]);
   end;  {if}
  end; {picked}

  procedure WriteSample;
   var
    h, v : INTEGER;
  begin
   repeat
   until Button;
   GetMouse(h, v);
   MoveTo(h, v);
   WriteDraw('A rose is a rose');
  end;

 begin {main}
  stopProgram := FALSE;
  SetUp(oldDrawingRect, theChoices);
  repeat
   PickedOne(theChoices, whichBut);
   case whichBut of
    0 :
     WriteSample;
    1 :
     stopProgram := TRUE;
   end;
   until stopProgram;
   SetDrawingRect(oldDrawingRect);
  end.
```

Figure 4-6

Fonts

Every character in any book, magazine, or newspaper, or on any computer screen, has been designed by somebody. Generally, an entire set of type, covering every letter, number, and character is designed at once. That set of characters is normally in a single style, so that it provides a pleasing, or at least coherent, appearance.

In the world of typography, a font generally has a fixed size and face. On the Macintosh, though, a font is simply a set of type in one style, and the size and face characteristics can change. Figure 4-2 shows some sample fonts.

On the Macintosh, the fonts are usually identified by the names of cities. Macintosh Pascal displays programs with the Geneva font. System information, such as menus and window titles, use the Chicago font.

Macintosh Pascal, though, does not use the font names. Instead, it uses identifying numbers. "0" identifies the system font. The other integers identify the fonts on that disk. You can add or remove fonts with the Font Mover utility, which you should have on the system disk that came with your Macintosh. To change the fonts on the Pascal disk, copy the Font Mover to that disk, and run it. You may have to experiment to find the identification numbers for the fonts you've added.

Fonts are set with the QuickDraw TextFont procedure. To create the *Explore Fonts* program:

1. Open the *Explore Text* program if it is not already open.
2. Choose Save As... from the File menu.
3. Type "Explore Fonts" as the file name.
4. Click in the Save button.
5. Change the value of NUMBUTS to 10. That line should now be:

   ```
   NUMBUTS = 10; {The number of buttons.}
   ```
6. Place an insertion point before the **case** statement in procedure DrawButtons and type:

   ```
   if (n<10) then
   WriteDraw(n-1:1)
   else
   ```
7. Still in procedure DrawButtons, delete the two lines:

   ```
   case n of
   1 :
   ```

   The line:

   ```
   WriteDraw('Stop');
   ```

   should follow the word **else**. (Notice there is no semicolon between **else** and WriteDraw.)
8. Still in procedure DrawButtons, delete:

   ```
   end; {ends case}
   ```

(If you didn't type the comment, just remove an **end** statement and a semicolon from the end of procedure DrawButtons.)

Procedure DrawButtons should now look like Figure 4-7.

```
procedure DrawButtons (var theChoices : choices);
 var
  n : INTEGER;
begin
 for n := 1 to NUMBUTS do
  begin
   SetRect(theChoices[n], BUTLEFT, n * BUTHEIGHT, BUTRIGHT, (n + 1) * BUTHEIGHT);
   FrameRect(theChoices[n]);
   MoveTo(BUTLEFT + SPACE, (n + 1) * BUTHEIGHT - SPACE);
   if (n < 10) then
    WriteDraw(n - 1 : 1)
   else
    WriteDraw('Stop');
  end; {ends for loop}
end; {ends DrawButtons}
```

Figure 4-7

The effect of this addition is to print a font number in each button except the last. Stop is printed in the last button. Look at the first line:

if (n<10) **then**

The symbol "<" means "less than." This statement checks the value of the **for** loop's index variable *n*. When *n* is less than 10, the statement following is executed. That statement prints the value of *n* in the button. When the value of *n* is equal to 10, "Stop" is printed in the button.

The statement that prints the button numbers needs some explanation. Look at it again:

WriteDraw(n-1:1)

The first part inside the parentheses, "n-1," subtracts 1 from the index variable to get the number of the font that will be used when this button is pressed. The second part, ":1," indicates a **minimum field width**. The field width tells WriteDraw how many characters to expect. When printing INTEGER values, if you specify a minimum field width, the number is printed starting at the current pen position. Additional characters are printed to the right. If you do not give a minimum field width, the number is printed centered a few pixels to the right of the pen position.

9. Place an insertion point before the **case** statement in the main program and type:

```
if (whichBut>0) and (whichBut<10) then
TextFont(whichBut-1)
else
```

10. In the **case** statement in the main program, change the line:

```
1 :
```
to:
```
10 :
```

The main program should now look like Figure 4-8.

11. Choose Save from the File menu.

```
begin {main}
 stopProgram := FALSE;
 SetUp(oldDrawingRect, theChoices);
 repeat
  PickedOne(theChoices, whichBut);
  if (whichBut > 0) and (whichBut < 10) then
   TextFont(whichBut - 1)
  else
   case whichBut of
    0 :
     WriteSample;
    10 :
     stopProgram := TRUE;
   end;
 until stopProgram;
 SetDrawingRect(oldDrawingRect);
end.
```

Figure 4-8

Run this program. Whenever you click in a button with the mouse, you change the font to the font number indicated. Then, when you click anywhere in the Drawing window, the sample string of text is printed in the current font.

Notice that several different font numbers produce the same characters. That is because you do not have nine fonts on the Macintosh Pascal disk, unless you put them there with the Font Mover. When you try to set the font to a number that doesn't exist, the Geneva font is displayed.

You can use any of these fonts in any program simply by calling TextFont, and giving the number of the font you want to use.

Faces

Aside from a font's basic style, any font can have a set of characteristics imposed on it. Face characteristics don't change the basic design of the font. They change the appearance of the characters in other ways.

The available characteristics are:

- **Bold. In this style, characters are repeated an appropriate number of times slightly offset, so that they appear darker than normal.**
- *Italic. In this style, the characters are skewed so that they have a distinct slant.*
- Underline. To underline characters, you give the underline characteristic. Then they appear with a line beneath them.
- Outline. Outline characters appear with a dark outline and hollow center :
- Shadow. Shadowed characters are outlined, and also have the outline thickened to the bottom right.
- Condense. When you condense text, the spaces between characters are made smaller by an appropriate amount. The characters themselves are not changed.
- Extend. When you extend text, the spaces between characters are made larger by an appropriate amount. The characters themselves are not changed.

All of these face characteristics are shown in Figure 4-3. You can combine any or all of these characteristics. However, *condense* and *extend* cancel each other out, and *outline* adds nothing to *shadow*. For example, type can be *bold, italic, shadowed* and *extended,* all at the same time.

Alter *ExploreText* to use these characteristics. If the *Explore Fonts* program is now displayed, choose Close from the File menu.

1. Open the *Explore Text* file.
2. Choose Save As... from the File menu.
3. Type "Explore Faces" as the new file name.
4. Click in the Save button.
5. Change NUMBUTS to 8, so that line is:

   ```
   NUMBUTS =8; {The number of buttons.}
   ```

6. Place an insertion point after the "1:" in the **case** statement in procedure DrawButtons. Type:

   ```
   WriteDraw('Plain');
   2 : WriteDraw('Bold');
   3 : WriteDraw('Italic');
   4 : WriteDraw('Outline');
   5 : WriteDraw('Shadow');
   6 : WriteDraw('Condense');
   7 : WriteDraw('Expand');
   8 :
   ```

7. Place an insertion point after the "1:" in the **case** statement in the main program. Type:

   ```
   TextFace([]);
   2 : TextFace([BOLD])
   3 : TextFace([ITALIC]);
   4 : TextFace([OUTLINE]);
   5 : TextFace([SHADOW]);
   6 : TextFace([CONDENSE]);
   7 : TextFace([EXTEND]);
   8 :
   ```

8. Choose Save from the File menu. The program should now look like Figure 4-9.

```
program ExploreText;
 const
  NUMBUTS = 8;  {The number of buttons.}
  BUTHEIGHT = 20; {The height of each button, in pixels.}
  BUTLEFT = 20;   {The left side of all buttons.}
  BUTRIGHT = 85;  {The right side of all buttons.}
  SPACE = 5;      {Used to space button labels.}
 type
  choices = array[1..NUMBUTS] of RECT; {The limits of each button.}
 var
  theChoices : choices;
  stopProgram : BOOLEAN;  {When TRUE, the program stops.}
  oldDrawingRect : RECT;
  whichBut : INTEGER; {The button that was pressed.}

 procedure DrawButtons (var theChoices : choices);
  var
   n : INTEGER;
 begin
  for n := 1 to NUMBUTS do
   begin
    SetRect(theChoices[n], BUTLEFT, n * BUTHEIGHT, BUTRIGHT, (n + 1) * BUTHEIGHT);
    FrameRect(theChoices[n]);
    MoveTo(BUTLEFT + SPACE, (n + 1) * BUTHEIGHT - SPACE);
    case n of
     1 :
      WriteDraw('Plain');
     2 :
      WriteDraw('Bold');
     3 :
      WriteDraw('Italic');
     4 :
      WriteDraw('Outline ');
     5 :
      WriteDraw('Shadow');
     6 :
      WriteDraw('Condense');
     7 :
      WriteDraw('Expand');
     8 :
      WriteDraw('Stop');
    end; {ends case}
   end; {ends for loop}
```

```
end; {ends DrawButtons}

procedure SetUp (var oldDrawingRect : RECT;
        var theChoices : choices);
 var
  tempBox : RECT; {Holds new Drawing window size.}
begin
 GetDrawingRect(oldDrawingRect);
 SetRect(tempBox, 0, 40, 520, 350);
 SetDrawingRect(tempBox);
 ShowDrawing;
 DrawButtons(theChoices);
end;

procedure PickedOne (var theChoices : choices;
        var whichBut : INTEGER);
 var
  n : INTEGER;
  thePoint : POINT;
begin
 whichBut := 0; {Resets this value.}
 repeat {Loops here until the mouse button is pressed.}
 until Button;
 GetMouse(thePoint.h, thePoint.v);
 for n := 1 to NUMBUTS do
  if PtInRect(thePoint, theChoices[n]) then
   begin
    InvertRect(theChoices[n]);
    whichBut := n;
    repeat
    until not Button;
    InvertRect(theChoices[n]);
   end; {if}
end; {picked}

procedure WriteSample;
 var
  h, v : INTEGER;
begin
 repeat
 until Button;
 GetMouse(h, v);
 MoveTo(h, v);
```

```
   WriteDraw('A rose is a rose');
  end;

begin {main}
 stopProgram := FALSE;
 SetUp(oldDrawingRect, theChoices);
 repeat
  PickedOne(theChoices, whichBut);
  case whichBut of
   0 :
    WriteSample;
   1 :
    TextFace([]);
   2 :
    TextFace([BOLD]);
   3 :
    TextFace([ITALIC]);
   4 :
    TextFace([OUTLINE]);
   5 :
    TextFace([SHADOW]);
   6 :
    TextFace([CONDENSE]);
   7 :
    TextFace([EXTEND]);
   8 :
    stopProgram := TRUE;
  end;
 until stopProgram;
 SetDrawingRect(oldDrawingRect);
end.
```

Figure 4-9

Run this program. When you choose a button, the text face is changed to the one given on that button. Then, when you click anywhere in the window, the sample text is printed with the new face characteristic. Click in the Stop button when you are done.

The words "[BOLD]," "[ITALIC]," "[OUTLINE]," "[SHADOW]," "[CONDENSE]," and "[EXTEND]" are used by the Mac to determine which face characteristics should be set. They appear in square brackets because they are members of a **set**. A set is a kind of data type that can hold a group of values. In this case, the parameter list of TextFace can hold any or all of the style characteristics. If you want to set more than one characteristic at a time, you should list them all in square brackets, separated by commas. For example, to set b*old* and *italic* at the same time:

```
TextFace([BOLD, ITALIC]);
```

Every time you write characters to the Drawing window, the characters print with the current text face characteristics.

Always separate each characteristic by a comma, and always put all characteristics within the same set of square brackets.

Size

You can set any size that can be held in an INTEGER variable (that is, up to 32767), although, in practice, you can only see sizes from 3 or 4 points to around 300. A point is about one seventy-second of an inch, about the size of a pixel.

Figure 4-4 shows samples of different type sizes.

To experiment with the QuickDraw procedure TextSize, which changes the current text size, follow the following steps. First, if you still have *ExploreFaces* or some other program showing, close that file.

1. Open the *ExploreText* program if it is not already opened.
2. Choose Save As... from the File menu.
3. Type "ExploreSizes" as the file name.
4. Click in the Save button.
5. Change the value of NUMBUTS to 5, so that line is:

   ```
   NUMBUTS = 5; {The number of buttons.}
   ```

6. In procedure DrawButtons, place an insertion point after the "1:" in the **case** statement and type:

   ```
   WriteDraw('3 point');
   2 : WriteDraw('12 point');
   3 : WriteDraw('48 point');
   4 : WriteDraw('300 point');
   5 :
   ```

7. In the main program, place an insertion point after the "1:" in the **case** statement in the main program and type:

   ```
   TextSize(3);
   2 : TextSize(12);
   3 : TextSize(48);
   4 : TextSize(300);
   5 :
   ```

8. Choose Save from the File menu.
   The program should now look like Figure 4-10.

```
program ExploreText;
 const
  NUMBUTS = 5;  {The number of buttons.}
  BUTHEIGHT = 20; {The height of each button, in pixels.}
  BUTLEFT = 20;    {The left side of all buttons.}
  BUTRIGHT = 85;  {The right side of all buttons.}
  SPACE = 5;       {Used to space button labels.}
 type
  choices = array[1..NUMBUTS] of RECT; {The limits of each button.}
 var
  theChoices : choices;
  stopProgram : BOOLEAN;  {When TRUE, the program stops.}
  oldDrawingRect : RECT;
  whichBut : INTEGER; {The button that was pressed.}

 procedure DrawButtons (var theChoices : choices);
  var
   n : INTEGER;
 begin
  for n := 1 to NUMBUTS do
   begin
    SetRect(theChoices[n], BUTLEFT, n * BUTHEIGHT, BUTRIGHT, (n + 1) * BUTHEIGHT);
    FrameRect(theChoices[n]);
    MoveTo(BUTLEFT + SPACE, (n + 1) * BUTHEIGHT - SPACE);
    case n of
     1 :
      WriteDraw('3 point');
     2 :
      WriteDraw('12 point');
     3 :
      WriteDraw('48 point');
     4 :
      WriteDraw('300 point');
```

```
      5 :
        WriteDraw('Stop');
     end; {ends case}
    end; {ends for loop}
end; {ends DrawButtons}

procedure SetUp (var oldDrawingRect : RECT;
         var theChoices : choices);
 var
  tempBox : RECT; {Holds new Drawing window size.}
begin
 GetDrawingRect(oldDrawingRect);
 SetRect(tempBox, 0, 40, 520, 350);
 SetDrawingRect(tempBox);
 ShowDrawing;
 DrawButtons(theChoices);
end;

procedure PickedOne (var theChoices : choices;
         var whichBut : INTEGER);
 var
  n : INTEGER;
  thePoint : POINT;
begin
 whichBut := 0; {Resets this value.}
 repeat {Loops here until the mouse button is pressed.}
 until Button;
 GetMouse(thePoint.h, thePoint.v);
 for n := 1 to NUMBUTS do
  if PtInRect(thePoint, theChoices[n]) then
   begin
    InvertRect(theChoices[n]);
    whichBut := n;
    repeat
    until not Button;
    InvertRect(theChoices[n]);
   end; {if}
end; {picked}

procedure WriteSample;
 var
  h, v : INTEGER;
 begin
  repeat
  until Button;
  GetMouse(h, v);
  MoveTo(h, v);
  WriteDraw('A rose is a rose');
 end;

begin {main}
 stopProgram := FALSE;
 SetUp(oldDrawingRect, theChoices);
 repeat
  PickedOne(theChoices, whichBut);
  case whichBut of
   0 :
    WriteSample;
   1 :
    TextSize(3);
   2 :
    TextSize(12);
   3 :
    TextSize(48);
   4 :
    TextSize(300);
   5 :
    stopProgram := TRUE;
  end;
 until stopProgram;
 SetDrawingRect(oldDrawingRect);
end.
```

Figure 4-10

Run this program.

Text looks best if the disk you are using has a font of the given size. A number of the font styles on the Macintosh are available in different sizes; the basic style is the same, only the size is different. If the current font is not available in the size you request, QuickDraw takes the version of the current font closest to that size and shrinks or enlarges the characters to the required size. That process is called **scaling**. The result is often choppy, as you can discover by running this program and choosing 300. In addition, if the requested size is not a whole-number multiple of the available size, the result looks even worse. This program uses size 48 instead of size 50 because fonts are most commonly available in size 12. You can check which size fonts are on your disk with the Font/DA Mover or FontMover utility. If you use FontMover, you must copy FontMover from your system disk to your Pascal disk to examine the Pascal fonts.

Experiment some more with text size by changing the numbers in the TextSize calls to different values.

The current text size is used whenever you draw text in the drawing window. You can call TextSize at any time in your programs to change the size of text that you subsequently print.

If you use a text size of 0, the size is automatically changed to the default, 12 points.

Transfer Mode

QuickDraw uses a transfer mode whenever you draw to the screen, whether you are drawing lines, boxes, text, or whatever. Text has its own mode setting, controlled by the QuickDraw procedure TextMode.

There are three modes for drawing text:

- *srcOr* always draws black characters. If the background is black, the characters are not visible. This is the initial mode.
- *srcXOr* draws black characters on white background, and white characters on a black background. Each pixel of the character is drawn by reversing the color of the pixel that occupies its destination on the screen, so that characters may be drawn partly black and partly white.
- *srcBic* always draws white characters. If the background is white, the characters are not visible.

The modes are demonstrated in Figure 4-5.

To try out the modes, follow these instructions. First, if you still have *ExploreSizes* or some other program showing, close that file.

1. Open the *ExploreText* program if it is not already opened.
2. Choose Save As... from the File menu.
3. Type "ExploreModes" as the file name.
4. Click in the Save button.

5. Change the value of NUMBUTS to 4, so that line of the program is:

   ```
   NUMBUTS = 4; {The number of buttons.}
   ```

6. Change the value of BUTRIGHT to 60, so that line of the program is:

   ```
   BUTRIGHT = 60; {The right side of all buttons.}
   ```

7. In procedure DrawButtons, place an insertion point after the "1:" in the **case** statement and type:

   ```
   WriteDraw('Or');
   2 : WriteDraw('XOr');
   3 : WriteDraw('Bic');
   4 :
   ```

8. In the main program, place an insertion point before the **repeat** statement and type:

   ```
   PaintRect(100, 100, 200, 200);
   ```

9. In the main program, place an insertion point after the "1:" in the **case** statement and type:

   ```
   TextMode(srcOr);
   2 : TextMode(srcXOr);
   3 : TextMode(srcBic);
   4 :
   ```

10. Choose Save from the File menu.

    The main program should now look like Figure 4-11.

```
program ExploreText;
 const
  NUMBUTS = 4;  {The number of buttons.}
  BUTHEIGHT = 20; {The height of each button, in pixels.}
  BUTLEFT = 20;   {The left side of all buttons.}
  BUTRIGHT = 60;  {The right side of all buttons.}
  SPACE = 5;      {Used to space button labels.}
 type
  choices = array[1..NUMBUTS] of RECT; {The limits of each button.}
 var
  theChoices : choices;
  stopProgram : BOOLEAN;  {When TRUE, the program stops.}
  oldDrawingRect : RECT;
  whichBut : INTEGER; {The button that was pressed.}
```

```
procedure DrawButtons (var theChoices : choices);
 var
  n : INTEGER;
begin
 for n := 1 to NUMBUTS do
  begin
   SetRect(theChoices[n], BUTLEFT, n * BUTHEIGHT, BUTRIGHT, (n + 1) * BUTHEIGHT);
   FrameRect(theChoices[n]);
   MoveTo(BUTLEFT + SPACE, (n + 1) * BUTHEIGHT - SPACE);
   case n of
    1 :
     WriteDraw('Or');
    2 :
     WriteDraw('XOr');
    3 :
     WriteDraw('Bic');
    4 :
     WriteDraw('Stop');
   end;  {ends case}
  end;  {ends for loop}
end;  {ends DrawButtons}

procedure SetUp (var oldDrawingRect : RECT;
          var theChoices : choices);
 var
  tempBox : RECT; {Holds new Drawing window size.}
begin
 GetDrawingRect(oldDrawingRect);
 SetRect(tempBox, 0, 40, 520, 350);
 SetDrawingRect(tempBox);
 ShowDrawing;
 DrawButtons(theChoices);
end;

procedure PickedOne (var theChoices : choices;
          var whichBut : INTEGER);
 var
  n : INTEGER;
  thePoint : POINT;
begin
 whichBut := 0; {Resets this value.}
 repeat {Loops here until the mouse button is pressed.}
 until Button;
 GetMouse(thePoint.h, thePoint.v);
 for n := 1 to NUMBUTS do
  if PtInRect(thePoint, theChoices[n]) then
```

```
      begin
       InvertRect(theChoices[n]);
       whichBut := n;
       repeat
       until not Button;
       InvertRect(theChoices[n]);
      end; {if}
  end; {picked}

  procedure WriteSample;
   var
    h, v : INTEGER;
  begin
   repeat
   until Button;
   GetMouse(h, v);
   MoveTo(h, v);
   WriteDraw('A rose is a rose');
  end;

 begin {main}
  stopProgram := FALSE;
  SetUp(oldDrawingRect, theChoices);
  PaintRect(100, 100, 200, 200);
  repeat
  PickedOne(theChoices, whichBut);
  case whichBut of
   0 :
    WriteSample;
   1 :
    TextMode(srcOr);
   2 :
    TextMode(srcXOr);
   3 :
    TextMode(srcBic);
   4 :
    stopProgram := TRUE;
  end;
 until stopProgram;
 SetDrawingRect(oldDrawingRect);
end.
```

Figure 4-11

Run this program. Notice the way the text in the different modes look in the black box and on white areas of the screen.

You can use any mode in your programs by calling TextMode. As with the other text characteristics, the most recently set value is the one used. You usually need to use text modes to make text visible on complicated or dark backgrounds. The default mode, srcOr, is fine for drawing on white backgrounds.

A Simple Editor

This section shows you how to handle text in your program: how to get it and how to store it.

If you haven't used MacWrite yet, please do, or, if you don't have that, run the text editor that came on your Macintosh Pascal disk. (You can find it in the Tools folder.)

Those programs do the basic things that an editor needs to do. They allow the user to type in a natural (that is, typewriter-like) way: allowing backspaces that do what you expect, carriage returns that do what you expect, and so on.

The editor created in this chapter is not as complicated as MacWrite or the Pascal text editor. For one thing, it does not handle more than a few lines. If you need to enter that much text, you can use MacWrite or the text editor.

What we will create is a small, flexible editor. It will allow a few short lines, enough for an address, for example.

Our editor should accept typing, including backspaces and carriage returns, until the user is done.

Nothing happens automatically when you type on a computer's keyboard. When you hit a key on a manual typewriter, a mechanical linkage moves a lever which moves one of the strikers which hits the ribbon, and a character is printed. When you hit the Backspace key, the carriage moves back one space. When you hit the Return bar, you mechanically move the platen so that you can continue typing at the beginning of the next line.

A computer is nothing like that. When you hit one of the keys, a small processor in the keyboard (actually a little computer in itself) figures out which key was hit, and sends a code to the Macintosh. What happens next depends on what the

Mac is programmed to do at that moment. Hitting the Return key, for example, does not move typing to the next line unless the computer is programmed to do that.

The simplest way to get the characters that the user types is with the Pascal Read procedure, which you used in Chapter 2.

Get a new programming screen.

Type the following program:

```
program TryEditing;
var
aChar : CHAR;
begin
ShowDrawing;
MoveTo (15,15);
repeat
Read(aChar);
WriteDraw(aChar);
until FALSE;
end.
```

The body of this program is surrounded by a **repeat/until** loop, with the **until** condition FALSE. That sets up an **infinite loop**: this program loops until you stop it with the Halt command in the Pause menu. (If you stop the program and want to start it again, choose Reset from the Run menu before giving the Go command. If you don't give the Reset command first, the program resumes running from where it was when you stopped it.)

Run this program, and type on the keyboard. The program reads characters from the keyboard, and prints them in the Drawing window. Letters, numbers, and other characters work perfectly well, but notice what happens if you press the Backspace or Return keys. The box that is printed when you enter a Backspace indicates that there is no symbol for that key in the font. Read gives you the character the user typed. The line:

```
WriteDraw(aChar);
```

tells Macintosh Pascal to draw the character. A Backspace is a non-printing character. Macintosh Pascal won't automatically erase the previous character in the Drawing window, as you'd expect a Backspace to do. You will later alter the program to erase characters in response to a Backspace.

You might expect to get a similar box when you press the Return key, but instead you get a space. The Read statement translates Returns into spaces. This is a serious problem in constructing an editor, because we can't find out when the user enters a Return. Read does too much processing before giving the characters to the program.

When you type a character or press the mouse button, that action creates an **event** for the Macintosh. An event is some occurrence requiring the attention of the Mac's processor. Events can come from the keyboard, the mouse, the disk drive, or from several other sources.

You can intercept events from a Macintosh Pascal program. When you do that, you get the events in a fairly "raw" state; that is, the Mac hasn't processed the information very much. You get the exact characters the user typed on the keyboard, and can even find out if the Option, Command, or Shift keys were pressed.

Warning

Macintosh Pascal normally gets events for you and checks if the event is a pressing of the mouse button in the Pause menu or the menu bar (which halts the program). When your program asks for events, though, Macintosh Pascal no longer gets them. That means the Pause menu does not work. If you use events in a program like the *TryEditing* program, programs that can't be stopped except through the Pause menu, the only way to stop the program is to turn off your Macintosh. So, whenever you use events, make certain there is a way for your program to end.

You use the GetNextEvent Macintosh Pascal function to get Mac events. When you call GetNextEvent, you specify the kind of events you want. If an event of that type is waiting, GetNextEvent becomes TRUE, and the event is returned. An event is returned in a data record of type EVENTRECORD, which has fields giving the event, its time, its location, and some other information. Unlike Read, GetNextEvent does not wait for an event before

returning to your program. It simply chec
see if there is an event of the type you've re
event of that type, it returns FALSE.

You generally use a **repeat** or **v**
GetNextEvent repeatedly until you get the

To alter *TryEditing* so that it uses event

1. Place an insertion point before *aChar* in the line:

   ```
   aChar : CHAR;
   ```

   and type:

   ```
   event: EVENTRECORD;
   ```

2. Replace the line:

   ```
   Read(aChar);
   ```

   with:

   ```
   repeat
   until GetNextEvent(mDownMask + keyDownMask, event);
   if (event.what < > MOUSEDOWN) then begin
   aChar:= Chr(event.message mod 256);
   ```

   The two symbols "<" and ">" when typed together, with no intervening space, mean "not equal to."

3. Place an insertion point after WriteDraw(*aChar*); and type:

   ```
   end;
   ```

4. Replace the word FALSE in the line "**until** FALSE;" with:

   ```
   event.what = MOUSEDOWN
   ```

   The program should now look like Figure 4-12.

```
program TryEditing;
 var
  event : EVENTRECORD;
  aChar : CHAR;
begin
 ShowDrawing;
 MoveTo(15, 15);
 repeat
  repeat
  until GetNextEvent(mDownMask + keyDownMask, event);
  if (event.what <> MOUSEDOWN) then
```

```
      begin
        aChar := Chr(event.message mod 256);
        WriteDraw(aChar);
      end;
    until event.what = MOUSEDOWN;
  end.
```

Figure 4-12

mDownMask and keyDownMask are predefined values that tell the Event Manager, the part of the Macintosh that controls events, that you are interested in finding out when the mouse button is pressed and when keys are typed. GetNextEvent returns FALSE until there is an event of one of the types you've asked for. The line:

```
if event.what < > MOUSEDOWN
```

checks the *what* field of the *event* record. The *event.what* field contains a predefined constant that indicates the type of event returned. MOUSEDOWN indicates a "mouse-button-down" event, that is, a pressing down of the mouse button.

The next line:

```
aChar := Chr(event.message mod 256);
```

requires some explanation.

Everything on a computer is represented by numbers. Every character has a code that represents that character. When a value is stored in a character variable, Pascal automatically interprets the number that is in that variable's "file drawer" as a character. The **Chr** function converts a number to the character that has the corresponding code. The number representing the key that was pressed is returned in *event.message*. There is also some other information there. To get the character code, you must find the remainder of the division of *event.message* by 256. The **mod** operator, short for modulus, finds the remainder of the division of the two operands. (Modulus, in this case, means the same as remainder.)

Here are some samples of results of the **mod** operator.

```
5 mod 2 is 1
4 mod 2 is 0
101 mod 50 is 1
101 mod 2 is 1
101 mod 102 is 101
```

Run this new version of the program. Notice that it has the same response as the version using Read, except that now nothing, not even a space, shows when you enter a Return. The character is there, but, because it is non-printing, nothing shows up on the screen.

To stop the program, just click the mouse button. You don't have to use the Pause menu. (In fact, as mentioned before, the Pause menu won't work while you're using events.)

It is fairly easy to modify this program so that it handles Returns. First, you must add some constants.

1. Place an insertion point before **var**, and type:

   ```
   const
   CR = 13; {The character code for Return.}
   LINEHEIGHT = 15;
   BASE = 10;
   ```

2. Place before the first **begin**, two lines after **var** and type:

   ```
   currentLine : INTEGER;
   ```

3. Remove the line:

   ```
   MoveTo(15, 15);
   ```

   and replace it with:

   ```
   MoveTo(BASE, LINEHEIGHT);
   ```

4. Place an insertion point before the first **repeat** and type:

   ```
   currentLine := 1;
   ```

5. Place an insertion point before WriteDraw and type:

   ```
   if aChar = Chr(CR) then
   begin
   currentLine := currentLine+1;
   MoveTo(BASE, currentLine*LINEHEIGHT);
   end
   else
   ```

   The program should now be as shown in Figure 4-13.

```
program TryEditing;
 const
  CR = 13;
  LINEHEIGHT = 15;
  BASE = 10;
 var
  event : EVENTRECORD;
  aChar : CHAR;
  currentLine : INTEGER;
begin
 ShowDrawing;
 MoveTo(BASE, LINEHEIGHT);
 currentLine := 1;
 repeat
  repeat
  until GetNextEvent(mDownMask + keyDownMask, event);
  if (event.what <> MOUSEDOWN) then
   begin
    aChar := Chr(event.message mod 256);
    if aChar = Chr(CR) then
     begin
      currentLine := currentLine + 1;
      MoveTo(BASE, currentLine * LINEHEIGHT);
     end
    else
     WriteDraw(aChar);
   end;
 until event.what = MOUSEDOWN;
end.
```

Figure 4-13

Notice that there is no semicolon after the **end** or **else**. It is an error if you have one in either place, because these are really in the middle of the compound **if** statement.

With these additions, whenever the character typed on the keyboard is equal to Chr(13), nothing is printed in the Drawing window. Instead, the pen is moved to the beginning of the next line. Chr(13), of course, is a Return. The position of the next line is defined by the constant LINEHEIGHT. If you wanted to double-space the lines, you would simply change LINEHEIGHT to 30.

You could deal with a Backspace in a similar way, but it would be repetitive here. Instead, discard this program, and go on to the *Simple Editor*, a program which also stores the characters the user types.

Storing Text

So far, the only way you have seen to store characters on the computer is with the CHAR data type. That only holds a single character, so it is not very good for most uses. What if you want to store a whole word, or even a sentence? You could create an array of type CHAR. In fact, that is the standard Pascal way to store groups, or **strings** of characters.

Most versions of Pascal, including Macintosh Pascal, define a special data type just for holding strings of characters, called the **string** data type. You can define variables of type **string**. In addition, any character or group of characters enclosed in single quote marks is a string. For example:

'This is a string.'

A string can hold up to 255 characters. Within that limit, the size of the string automatically varies to hold the number of characters you need. (Arrays are always of a fixed size: if you define an array of 255 characters, it will always contain 255 characters, even if most of them are blanks or undefined.) As with arrays, you can access each character in the string by number. For example, the first character of the string *thisString* is *thisString[1]*, the second is *thisString[2]*, and so on.

There is a special set of predefined string functions and procedures that help you manipulate strings. Those functions are:

- The Length function, which returns the current number of characters in a string.
- The Pos function, which searches a string and returns the position of the first occurrence of some sequence of characters.
- The Concat function, which attaches some number of strings together, and returns the result.
- The Copy function, which copies some range of characters from a string and returns it.
- The Delete procedure, which removes some range of characters from a string.
- The Omit function, which returns a string minus some range of characters. This is the same as Delete, except that the original string is not changed.

- The Insert procedure, which inserts characters into a string at a given place.
- The Include function, which returns a string with some characters inserted at a given place. This is the same as Insert, except that the original string is not changed.
- The StringWidth function, which returns the size, in pixels, of a string in the current font, size, and face. This is a QuickDraw function. There is also a related function, CharWidth, that returns the pixel size of a character in the current font, size, and face.

Notes

Although strings are not part of Standard Pascal, most versions of Pascal have a **string** type and string subprograms similar or identical to those used in Macintosh Pascal.

Close *TryEditing* (you can throw out *TryEditing*, if you want) and get a new programming window.

Choose Save As... from the File menu. When you are asked for a file name, type "Simple Editor". Click in the Save button.

Type:

```
program Simple_Editor;
const
CR = 13; {Character code for Return.}
BS = 8; {Character code for Backspace.}
LINEHEIGHT = 15; {The distance between lines in pixels.}
BASE = 15; {The beginning of each line.}
ENDSTR = 256; {Guaranteed to be past the end of the string.}
var
theString : string;
begin {main}
ShowDrawing;
theString := ''; {Two single quote marks. Initializes string }
                 {to empty.}
ShowString(theString);
DoEditing(theString);
end.
```

That is the main program.

The constant BASE defines the distance of the beginning of each line from the left edge of the Drawing window, in pixels.

ENDSTR has a value one above the maximum number of characters that can be in a string. It is used to find a position past the end of any string, and to find out when the string is full.

The main program has only one variable *theString* which holds the characters typed by the user.

ShowDrawing is called at the beginning of the main program to make certain the Drawing window is visible.

The line

```
theString := '';
```

makes *theString* an **empty string**, which simply means that *theString* contains nothing.

Notes

As always in Pascal, the characters '' are two single quotation marks (' and ') not one double quotation mark ("). Please make certain you typed two single quote marks, or the program won't work.

Although you might think that *theString* would contain nothing anyway at this point, that is not necessarily true. When you define a variable, Pascal allocates some space for the variable. That space is simply a part of memory that Pascal allows you to refer to by the variable name. That part of memory might have been used for something else at an earlier time and still contain other values. Nothing is done to make certain the memory contains something appropriate; new variables are said to be **undefined** until you assign them some value. You could just as easily give *theString* some other initial value, such as:

```
theString := 'An initial string.'
```

The line

```
ShowString(theString);
```

calls the procedure ShowString, written later in this chapter, to display the string. Although this may seem unnecessary given that *theString* contains nothing, calling ShowString here allows you to give *theString* another initial value with no change to the program. (In a later version of this program, you will use values stored on disk to give *theString* an initial value.) In addition, ShowString places the pen in a position to begin writing characters to the Drawing window as they are typed.

The line

```
DoEditing(theString);
```

calls the procedure DoEditing. That procedure, which you will define in the next few pages, does most of the editing work of the program. It does all the editing of the string. When editing is finished, DoEditing stops, and the main program again takes over.

Now, define procedure ShowString. Place an insertion point before **begin** {main} and type:

```
procedure ShowString (theString : string);
var
n, currentLine : INTEGER;
begin
MoveTo(BASE, LINEHEIGHT);
currentLine := 1;
for n := 1 to Length(theString) do
if theString[n] = chr(CR) then
begin
currentLine := currentLine + 1;
MoveTo(BASE, currentLine * LINEHEIGHT);
end
else
DrawChar(theString[n]); {Predefined.}
end;
```

This procedure depends on *theString* containing ordinary characters and Returns. It uses the loop

```
for n := 1 to Length(theString) do
```

to scan through the string a character at a time. If the character is a Return (that is, if *theString* [*n*]=Chr(CR)) the pen is moved to the beginning of the next line. Otherwise, the QuickDraw procedure DrawChar is used to draw the character.

Notice the way *theString* is defined in the parameter list for procedure ShowString.

```
procedure ShowString (theString :string);
```

theString is *not* defined as a **var** parameter. That is because ShowString does not change the value of *theString*. A programmer looking at the definition of the procedure would know that fact simply from the parameter list: if ShowString does change the value of *theString*, those changes are not passed back to the main program, because *theString* is not defined to be a **var** parameter. As a general rule of good programming, any possible effects of a routine should be clear from the routine's parameter list. Otherwise you have to examine the routine to see exactly what variables it changes.

Now, for the heart of the program. Place an insertion point before the line **begin** {main} and type:

```
procedure DoEditing (var theString : string);
var
event : EVENTRECORD;
typedChar: CHAR;
begin
while (event.what < > MOUSEDOWN) do
begin
repeat
until GetNextEvent(mDownMask + keyDownMask, event);
if (event.what < > MOUSEDOWN) then
begin
typedChar := Chr(event.message mod 256);
ShowNewChar(typedChar, theString);
AddChar(typedChar, theString);
end; {if}
end; {while}
end; {DoEditing}
```

Notice that *theString* is a **var** parameter for DoEditing, because this routine does modify the value of *theString*.

The loop:

```
repeat
until GetNextEvent(mDownMask + keyDownMask, event);
```

is identical to the equivalent loop in *TryEditing*. It waits until the user types a key or presses the mouse button.

The resulting event is tested by the **if** statement to see if it is a MOUSEDOWN, which indicates that the mouse button has been pressed. If it is not a MOUSEDOWN, then it is a typing event, and the following sequence of statements is executed:

```
typedChar := Chr(event.message mod 256);
ShowNewChar(typedChar, theString);
AddChar(typedChar, theString);
```

The first line should look familiar, as it uses the same method used in *TryEditing*. The procedure ShowNewChar, which you will define later in this chapter, displays the new character. AddChar adds the character to the string. You could add the character first, and then call ShowString to display the entire string. However, one of the realities of computing is that screen display is always relatively slow. If you redisplayed the entire string every time a character is typed, even a relatively slow typist would have to wait to see the characters on the screen. It is much quicker to define a special procedure that just displays the typed character, only drawing the entire string when necessary.

Finally, the **while** loop and DoEditing end when the mouse button is pressed.

Here is the procedure that displays the new character. Place an insertion point before procedure DoEditing and type:

```
procedure ShowNewChar (typedChar:CHAR; theString:string);
var
lastChar : CHAR;
len : INTEGER;
begin
len := length(theString);
if len >0 then
lastChar := theString[len];
case Ord(typedChar) of
BS :
if theString<>'' then {Note: two single quote marks.}
if lastChar = chr(CR) then
ShowString(Omit(theString, len, 1))
else
begin
Move(-CharWidth(lastChar), 0);
TextMode(srcXOr);
```

```
DrawChar(lastChar);
TextMode(srcOr);
Move(-CharWidth(lastChar), 0);
end;
CR :
if len + 1 < ENDSTR then
ShowString(Concat(theString, Chr(CR)));
otherwise
if len+ 1 < ENDSTR then
DrawChar(typedChar);
end;
end;
```

This is the most complicated procedure in the program. It is built around a **case** statement that uses the **Ord** of the *typedChar* as the case constants. Ord is a predefined Pascal procedure that returns the code for a character. It is the opposite of Chr. There are three possible situations that must be dealt with:

1. *typedChar* is a Backspace, in which case the program must erase the previous character, and reposition the pen.
2. *typedChar* is a Return, in which case the program must move the pen to the beginning of the next line.
3. *typedChar* is an ordinary character, in which case it needs to be drawn on the screen at the current pen position.

The first case is the most complicated. Characters on the Macintosh do not fit neatly into little boxes. They are of different widths and heights, depending on the character, the font, and the size. You can't even depend on the character ending at the current line, because many characters (such as "y," "j," and "p") drop below the line. The easiest way to erase the previous character is to look in the string to find out what it is, back the pen up so that it is in a position to draw the character, set the pen to *srcXOr* mode, which is the text mode that draws white on black, and draw the character, which neatly erases it without changing anything else. Then, the pen must be backed up once more, to position it for drawing the next character. Before doing that, however, ShowNewChar:

- Must check to make certain the string is not empty, in which case trying to find the previous character would be an error, and
- Must find out if the previous character is a Return.

If the character is a Return the pen must be moved to the end of the previous line. The program would run more quickly if you wrote a routine that figured out the position of the end of the previous line, but it turns out that, as this situation is relatively infrequent, the program works acceptably if you call ShowString to display the string without the final Return. That positions the pen correctly. Notice that the backspaced character is not actually removed from the string in ShowNewChar. ShowNewChar only displays the new character, and does not change *theString*, as can be seen from the parameter list of ShowNewChar.

When *typedChar* is a Return, ShowNewChar calls ShowString with the new Return added on the end with the predefined function Concat. Once again, it would be faster to define a procedure that calculated the new pen position, and simply move the pen, but Returns are sufficiently infrequent that the program's performance is acceptable. Very often performance and program complexity are trade-offs, and you need to use your judgment to decide if some improvement is worth the effort.

The final procedure is the only one that actually changes the value of *theString*. Place an insertion point before procedure DoEditing and type:

```
procedure AddChar ( typedChar : CHAR;
var theString : string);
begin
if typedChar = Chr(BS) then {This is a Backspace.}
Delete(theString, length(theString), 1)
else if Length(theString) + 1 < ENDSTR then {Forward typing.}
Insert(typedChar, theString, ENDSTR);
end;
```

I'll let you figure this one out yourself.

Choose Save from the File menu.

The program should now be as shown in Figure 4-14.

```
program Simple_Editor;
 const
  CR = 13;  {Character code for <RETURN>.}
  BS = 8;   {Character code for backspace.}
  LINEHEIGHT = 15;  {The distance between lines in pixels.}
  BASE = 15;        {The beginning of each line.}
  ENDSTR = 256;     {Guaranteed to be the past end of the string.}
 var
  theString : string;

 procedure ShowString (theString : string);
  var
   n, currentLine : INTEGER;
 begin
  MoveTo(BASE, LINEHEIGHT);
  currentLine := 1;
  for n := 1 to Length(theString) do
   if theString[n] = chr(CR) then
    begin
     currentLine := currentLine + 1;
     MoveTo(BASE, currentLine * LINEHEIGHT);
    end
   else
    DrawChar(theString[n]);  {Pre-defined.}
 end;

 procedure ShowNewChar (typedChar : CHAR;
          theString : string);
  var
   lastChar : CHAR;
   len : INTEGER;
 begin
  len := length(theString);
  if len > 0 then
   lastChar := theString[len];
  case Ord(typedChar) of
   BS :
    if theString <> '' then  {Note: two single quote marks.}
     if lastChar = chr(CR) then
      ShowString(Omit(theString, len, 1))
     else
      begin
       Move(-CharWidth(lastChar), 0);
       TextMode(srcXOr);
       DrawChar(lastChar);
       TextMode(srcOr);
       Move(-CharWidth(lastChar), 0);
      end;
   CR :
    if len + 1 < ENDSTR then
     ShowString(Concat(theString, Chr(CR)));
   otherwise
    if len + 1 < ENDSTR then
     DrawChar(typedChar);
  end;
 end;

 procedure AddChar (typedChar : CHAR;
          var theString : string);
 begin
  if typedChar = Chr(BS) then {This is a backspace.}
   Delete(theString, length(theString), 1)
  else if Length(theString) + 1 < ENDSTR then {Forward typing.}
   Insert(typedChar, theString, ENDSTR);
 end;

 procedure DoEditing (var theString : string);
  var
   event : EVENTRECORD;
   typedChar : CHAR;
 begin
  while (event.what <> MOUSEDOWN) do
   begin
    repeat
    until GetNextEvent(mDownMask + keyDownMask, event);
    if (event.what <> MOUSEDOWN) then
     begin
      typedChar := Chr(event.message mod 256);
      ShowNewChar(typedChar, theString);
      AddChar(typedChar, theString);
     end
   end;
 end;

begin {main}
 ShowDrawing;
```

```
  theString := '';      {Two single quote marks. Initializes string }
                        {to empty.}
  ShowString(theString);
  DoEditing(theString);
end.
```

Figure 4-14

Run this program. Notice the way it responds to Backspaces, Returns and ordinary typing.

Do More

1. Write a routine PenToEnd that moves the pen to the end of a given string so that you can replace the two calls to ShowString in ShowNewChar with:

   ```
   PenToEnd (Omit(theString, Length(theString), 1))
   PenToEnd (Concat(theString, chr(CR)));
   ```

2. The *event* record also specifies whether the Shift, Option, Caps Lock, and Command keys are pressed. You can find out the state of those keys by examining the *modifiers* field of the *event* record.

   The value of *event.modifiers* depends on which of the modifier keys are pressed. A set of constants is defined to help you decode *event.modifiers.* The constants and their values are:

   CMDKEY = 256
   SHIFTKEY = 512
   ALPHALOCK = 1024
   OPTIONKEY = 2056

   In effect, each of these values is added to the value of *event.modifiers* if that particular key was down when the event occurred.

   The following program fragment shows one way to decode *event.modifiers* into a set of TRUE/FALSE values, one each for the Option, Caps Lock, Shift, and Command keys. *Opt, lock, shift*, and *cmd* are defined as BOOLEANs.

```
opt := event.modifiers >= OPTIONKEY;
if opt then
event.modifiers := event.modifiers - OPTIONKEY;
lock := event.modifiers >= ALPHALOCK;
if lock then
event.modifiers := event.modifiers - ALPHALOCK;
shift := event.modifiers >= SHIFTKEY;
if shift then
event.modifiers := event.modifiers - SHIFTKEY;
cmd := event.modifiers >= CMDKEY;
```

Notice that a Boolean expression is used in each case to set the value of the key's BOOLEAN.

Is it necessary to initialize the BOOLEANs?

Notice, also, that the check begins with the greatest value, OPTIONKEY (which equals 2048). Do you understand why?

How would you decode *event.modifiers* if you only wanted to know if the Command key was pressed?

3. Use *event.modifiers* for some new kind of action in the program. For example, when you receive a Backspace, check to see if the Option key is also pressed. If it is, delete the entire previous word. Do that by deleting characters backwards in the string until you reach a space or a Return.

QUICK SUMMARY

This chapter explores what you can do with text on the Macintosh. It shows how a modularized program, such as *Shapes*, can be converted to other uses. It discusses and uses strings and string functions and procedures. Events are explained and used. It then develops a simple text editor that solves the problem of what to do with Backspaces and Returns. The following routines, operators, and concepts are introduced.

CharWidth	is a QuickDraw function that returns the width in pixels of a character in the current font, face, and size.

Concat	is a predefined string function that attaches two or several strings together into one string.
Copy	is a predefined string function that copies some range of characters from a string and returns it.
Delete	is a predefined string procedure that removes some range of characters from a string.
Editor	is a program that allows you to change text in a way similar to the way you type on a typewriter.
Empty string	is a string with no characters in it; that is, with a Length of zero.
Event	is an occurrence that requires the attention of the Macintosh's processor. More formally, an event is a data record of type EVENTRECORD, which contains a record of such an occurrence. Every keystroke causes an event, as does pressing the mouse button. Many events come from other parts of the computer, such as the disk drive.
Face	in relation to type is a characteristic that may be added to any font. There are seven face characteristics: *bold, italic, underline, outline, shadow, condense*, and *extend*.
Field width	is an optional parameter to the Write, WriteLn, and WriteDraw predefined procedures which helps determine the way the printed text appears. You specify the field with with a colon and a number following the printing item.
Font	is a set of characters in a single style. There are many thousands of fonts in the world. The Mac has less than a dozen, although it is likely that more will become available as time goes on. On the Mac, a font is defined as a set of images stored on the disk. Outside the Macintosh world, a particular font has a fixed size, but the Macintosh can scale a font to be of any size.
GetNextEvent	is a predefined function that returns TRUE when an event of a type you've requested occurs, and also returns an event of a type you've requested, if there is one of that type. If there is no event of the type you've requested, GetNextEvent returns FALSE. You give one or a number of masks to specify the events you want. Caution: when you use this call, the Pause menu does not work.
Include	is a predefined string function that returns a string with some characters inserted at a given place.
Infinite loop	is a loop that has no way to end. It will run until you somehow intervene to stop it, such as by using the Pause menu or turning off your Macintosh.

Insert	is a predefined string procedure that inserts characters into a string in a given place.
keyDownMask	is the event mask that specifies a keystroke event. See GetNextEvent.
Length	is a predefined string function that returns the number of characters in a string.
Mask	is a value whose pattern of bits has special meaning. In general, you can add masks together to indicate that all the masks separately should be in effect. See GetNextEvent.
mDownMask	is the event mask that specifies mouse down events. See GetNextEvent.
Mode	see Transfer mode.
mod	is a reserved word used as an operator that produces the remainder of the division of its two operands.
MOUSEDOWN	is the value of *event.where* in a mouse-down event. One mouse-down event is produced each time you press the mouse button.
Omit	is a predefined string function that returns a string minus some range of characters.
Point	is a unit of text measurement. About 1/72 of an inch.
Pos	is a predefined string function that searches a string and returns the position of the beginning of the first occurrence of a given group of characters.
Remainder	is what is left over when you divide one number into another a whole number of times. In other words, the remainder is the dividend minus the quotient times the divisor.
Scaling	is the process of enlarging or shrinking a character or picture. You can specify any type size, and the Macintosh automatically scales the information that it has so that the characters appear in the desired size.
Size	in relation to type is the average size of each character, expressed in points. A point is about 1/72 of an inch, about the size of a pixel. The Mac will scale type to any size; however, it looks best when displayed in a size that is actually stored on the disk.
srcBic	see Transfer mode.
srcOr	see Transfer mode.
srcXOr	see Transfer mode.

string	is a reserved word used to define a data type that can hold from 0 to 255 characters. A string automatically changes size to hold the required number of characters. Every element of a string is of type CHAR, and can be accessed with an element number as if the string were an array. Unlike most other reserved words, **string** is not included in all other versions of the Pascal language.
String	is a group of characters. See **string**.
StringWidth	is a QuickDraw function which returns the size in pixels of a given string in the current font, face, and size.
System font	the font used to print system information, such as menu titles and menu commands.
TextFace	is a QuickDraw procedure that changes the text face setting.
TextFont	is a QuickDraw procedure that changes the text font setting.
TextMode	is a QuickDraw procedure that changes the text mode setting.
TextSize	is a QuickDraw procedure that changes the text size setting.
Transfer mode	determines the relationship between the pixel image defined for whatever is to be drawn and what actually appears on the screen. When drawing text, the mode must be one of the three allowable text modes: *srcOr*, which prints black type; *srcXOr*, which reverses what is on the screen; and *srcBic*, which draws white characters.

CHAPTER 5 Fielding

This chapter takes methods developed for the *Simple Editor* and generalizes them, so you can edit text inside a box of any size located anywhere on the screen. That program is then extended so that you can edit in any number of boxes simultaneously.

This chapter's program is built in stages. The first stage is a one-field editor that allows you to create a record, edit it, and stop the program. The next chapter shows you how to store the edited text on disk, so you have a permanent copy.

Developing a One-field Editor Program

You could use this editor to keep a telephone book, a list of recipes, or any other set of information. Information collected together is referred to as a **data base**.

The main program goes through the following steps.

First, in the initial stage it:

1. Sets up the display.
2. Gets the size and location of the editing box from the user.
3. Draws the editing box.

Then, there is a loop based around a **case** statement. The value of the **case** variable depends on the position of the mouse button press—that is, where the mouse button is pressed. There are three possibilities:

1. The mouse was in the editing field.
2. The mouse was in the Stop button.
3. The mouse was in neither.

Initially, the case selector, *choice*, is set to indicate the third choice. When *choice* is set to that value, the program waits for another mouse button press. When one is received, the program drops through to the end of the **case** statement, and the Picked procedure is called. Picked figures out where the mouse was when the button was pressed, and sets *choice* accordingly. If the mouse was in the Stop button, the program then stops. Otherwise, it loops back, and the **case** statement is executed with the new value of *choice*.

When the mouse is pressed in the editing field, the value of *choice* causes the DoEditing function to begin execution. DoEditing executes until the mouse button is pressed again, at which point the program again drops through the **case** statement and calls Picked with the new mouse position.

Here is the main program. Get a new programming window and type:

```
program Field_Editor;
const
CR = 13; {Character code for Return.}
BS = 8; {Character code for Backspace.}
LINEHEIGHT = 15; {The distance between lines in pixels.}
ENDSTR = 256; {Guaranteed to be past the end of the string.}
SPACE = 3;
BUTHEIGHT = 20;
BUTLEFT = 20;
BUTWIDTH = 40;
type
stringRec = record
wPos : POINT;
theString : string;
itsRect : RECT;
end;
butType = record
```

```
stop : RECT;
end;
choiceType = (STOPIT, AFIELD, NOTHING);
var
choice : choiceType;
aRecord : stringRec;
buttons: butType;
drawingRect : RECT;
event : EVENTRECORD;
begin {main}HideAll;
SetUpDisplay(drawingRect);
{Preceding line expands the Drawing window to full screen.}
ChangeTheBox(aRecord);
DrawScreen(aRecord, buttons);
choice := NOTHING;
repeat
case choice of
AFIELD :
begin
ShowCell(aRecord, TRUE);
DoEditing(aRecord, event);
ShowCell(aRecord, FALSE);
end;
NOTHING :
begin
repeat
until GetNextEvent(mDownMask, event);
GlobalToLocal(event.where);
end
end;
Picked(choice, aRecord, event.where, buttons);
until choice = stopIt;
SetRect(drawingRect, 293, 124, 508, 339);
SetDrawingRect(drawingRect); {Restores drawing window.}
end.
```

The predefined HideAll procedure removes the Text, Drawing, and programming windows from the display. Doing that clears up some space in memory for the program. It is a good idea to use the HideAll call in any sizable program, particularly if you have a 128K Macintosh. After the program finishes running, however, the programming window remains hidden. Use the Windows menu to redisplay it.

The new data type butType is used to store the RECT of the onscreen button, Stop, that is used in this program. This uses a record, rather than a simple variable, so that other buttons can be easily added.

Enumerated Data Types

The data type definition:

```
choiceType = (STOPIT, AFIELD, NOTHING);
```

needs special mention. This is an **enumerated type**. That is a long word for a simple concept. An enumerated type is a Pascal type whose values are defined by the programmer. A variable of an enumerated type can only take on the values defined for that type. In other words, you can make the following assignments to the variable *choice*, defined to be of the type choiceType:

```
choice := STOPIT;
choice := AFIELD;
choice := NOTHING;
```

You can't assign the variable *choice* any other value.

Enumerated types are also **ordinal types**. Other ordinal types are INTEGER and CHAR. An ordinal type's possible values have a sequential order. You can see how that is with INTEGERs: the order of the set of INTEGERs is simply the normal counting system—1 is less than 2 which is less that 3, and so on. CHAR values have their order determined by the numerical order of the character's codes: A is less than B which is less than C, and so on.

Enumerated types have their order set by the type definition. The first identifier in the type's list of identifiers has the lowest value, and the last identifier has the highest value. Given the definition of choiceType given in the program, the following logical expressions are all TRUE.

```
STOPIT < NOTHING
AFIELD > STOPIT
NOTHING > STOPIT
```

The identifiers NOTHING, STOPIT, and AFIELD are the **constants** of choiceType, just as "e", "a", and "g" are three of the constants of type CHAR, and 3, 3200, and -256 are three constants of type INTEGER. Notice that the values of choiceType are used in the **case** statement and in the **if** statement in the main program, in the same way constants of any type can be used.

You could define *choice* as an INTEGER, and let the **case** statement branch on INTEGER values. An enumerated type is used because it is clearer.

Global and Local Coordinates

Towards the end of the main program are the lines:

```
repeat
until GetNextEvent(mDownMask, event);
GlobalToLocal(event.where);
```

You have seen GetNextEvent used before. In this case, though, the most important question is where the event took place. The where field of the event (in this case, *event.where*) always contains the position of the mouse at the time of the event.

However, the Event Manager returns the mouse position in pixels measured from the top left corner of the screen, rather than the top left corner of the Drawing window. Such coordinates are referred to as **global coordinates**. QuickDraw generally (but not always) uses coordinates relative to a single window. Those coordinates are referred to as **local coordinates**. The QuickDraw procedure GlobalToLocal converts a point from global coordinates to the current local coordinates. When you use Macintosh Pascal, the local coordinates are almost always relative to the Drawing window, so that the upper left corner of the Drawing window is (0,0). (The only way you can change the local coordinate system is by using the QuickDraw **grafport** subprograms, which are not used in this book. See the QuickDraw appendix of the *Macintosh Pascal Technical Appendix* for more information.) There is a corresponding QuickDraw procedure LocalToGlobal, which converts a point from local coordinates to global coordinates.

Figure 5-1 illustrates the difference between *global* and *local* coordinates. Notice that any point can be expressed in global or local coordinates. You just have to know which system you are using.

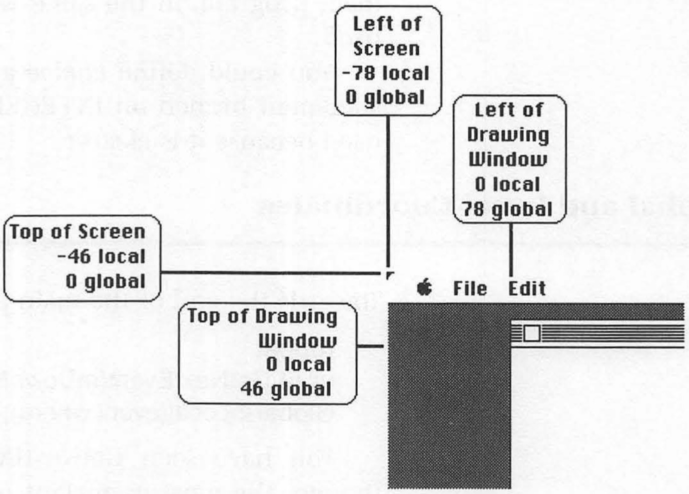

Left of Screen -78 local 0 global
Left of Drawing Window 0 local 78 global
Top of Screen -46 local 0 global
Top of Drawing Window 0 local 46 global
File
Edit

Figure 5-1 Global and Local Coordinates

The adjectives *global* and *local*, which are used in in many different contexts, refer to points of view which are more or less general than each other. A global point of view might look at the way something changes the entire world, while a local point of view looks at how it affects a single village, as if the outside world didn't exist. A common use of the terms concerns variables. Variables defined in a program's **var** section are *global to the program*; they can be referred to from any procedure or function anywhere in the program. Variables defined in a procedure or function's **var** section, in contrast, are *local to the procedure or function*; they are meaningless outside the procedure or function. When you are inside the procedure or function, you usually cannot tell whether a variable is global or local. (An important exception concerns **for** loop index variables. Those must be local; they must be defined in the part of the program that uses them.) One important point: you can use the same names to define global variables and local variables. When there is a local variable of a given name, it is as if the global variable of the same name does not exist. Many programs use global variables to make variables available to procedures and functions without passing the variables to the subprograms as parameters. Although that is very easy and tempting, it is generally not advisable.

When a subprogram modifies a global variable, that action is called a **side-effect**. The problem is that it is then difficult to find all subprograms that change a given variable. If, for example, the value of the global variable is mistakenly changed in a procedure, that error may be very difficult to find.

The rest of the main program is explained in connection with the corresponding subprograms.

Creating Onscreen Buttons

The action of the program turns on the procedure Picked, so that is defined first.

Place an insertion point before **begin** {main} and type:

```
procedure Picked (var choice : choiceType;
aRecord : stringRec;
mouse : POINT;
buttons : butType);
function ButtonPressed (whichButton : RECT;
mouse : POINT) : BOOLEAN;
begin
ButtonPressed := FALSE;
if PtInRect(mouse, whichButton) then
begin
InvertRoundRect(whichButton, 6, 6); {QuickDraw procedure.}
ButtonPressed := TRUE;
end;
end;
begin {Picked}
choice := NOTHING;
if PtInRect(mouse, aRecord.itsRect) then
choice := AFIELD
else if ButtonPressed(buttons.stop, mouse) then
choice := STOPIT
end;
```

The first thing in this procedure is a function that checks the special case of an onscreen button. At this point, the only onscreen button is the Stop button, whose RECT is stored in *buttons.stop*.

A subprogram that is enclosed in another procedure or function in that way is called a **local subprogram**. It cannot be called from outside the procedure or function that contains it.

ButtonPressed is a separate function for two reasons: first, for clarity, so that the procedure Picked is clearer; second, for generality. Although this program has only one button, it is not hard to add others. ButtonPressed can check any button, as long as you pass it the button's RECT.

Notice that InvertRoundRect is used here in place of InvertRect. The onscreen buttons in this program are drawn as round-cornered rectangles, because they look better that way. The two 6's in the InvertRoundRect call define the size of the curves drawn at each corner of the round rectangle.

The next two procedures define the button. These are capable of handling more than the one button used in this program; the same procedures are also used later when more buttons are added.

Place an insertion point before the words "procedure Picked" and type:

```
procedure DrawButton (var butName : RECT;
vertBase : INTEGER; theLabel : string);
begin
SetRect(butName, BUTLEFT, vertBase, BUTLEFT +
BUTWIDTH, vertBase + BUTHEIGHT);
FillRoundRect(butName, 6, 6, white);
FrameRoundRect(butName, 6, 6);
MoveTo(BUTLEFT + SPACE, vertBase + 14);
TextFont(0);
WriteDraw(theLabel);
TextFont(1);
end;
procedure DrawScreen(aRecord: stringRec; var buttons :
butType);
begin
FillRect(drawingRect, gray);
DrawButton(buttons.stop, 310, 'Stop');
ShowCell(aRecord, FALSE);
end;
```

These procedures are straightforward. Notice that the font is changed to the system font (font 0) before the button's label is drawn. That makes the text in the buttons look like the text in menus.

Notice also that, because a record is used to store the RECT of the Stop button, you won't have to change the parameter list of DrawScreen when you add more buttons.

Defining the Drawing Window

The following procedure sets up the display. Place an insertion point before the words "procedure Picked" and type:

```
procedure SetUpDisplay (var drawingRect : RECT);
begin
SetRect(drawingRect, 0, 0, 532, 358);
SetDrawingRect(drawingRect);
ShowDrawing;
FillRect(drawingRect,gray);
end;
```

This procedure uses global coordinates: the RECT that defines the Drawing window must be defined in terms of the entire screen. The predefined procedure SetDrawingRect changes the Drawing window to the given RECT. The variable *drawingRect* is set by the line:

```
SetRect(drawingRect, 0, 0, 532, 358);
```

The dimensions given there define a rectangle somewhat larger than the screen. After the line:

```
SetDrawingRect(drawingRect);
```

is executed, the Drawing window fills the entire screen. In fact, it is slightly larger than the screen so that the borders of the window are hidden. This is a cosmetic touch, so that this program can use the entire Macintosh screen. (You cannot draw outside the Drawing window, remember.) Notice there is no way to reach the mouse controls of the Drawing window after it is enlarged by this procedure. The last two lines in the main program are:

```
SetRect(drawingRect, 293, 124, 508, 339);
SetDrawingRect(drawingRect);
```

which resets the Drawing window to its size when Macintosh Pascal first starts. If, by some chance, your program halts before executing that last statement, the Drawing window will be stuck in its expanded state. You can restore it to its original size by quitting Macintosh Pascal, and then restarting it, or you can wait until you run this program again. (It is not a serious problem if the Drawing window is stuck in its expanded state. As usual, you can display the programming window by picking the program's same from the Windows menu.)

Sizing the Editing Field

The following procedure is called to set the size of the editing field.

Place an insertion point before **begin** {main} and type:

```
procedure ChangeTheBox (var aRecord : stringRec);
var
event : EVENTRECORD;
begin
PenMode(patXOr);
with aRecord do
begin
repeat
repeat
until GetNextEvent(mDownMask, event);
GlobalToLocal(event.where);
itsRect.topLeft := event.where;
repeat
GetMouse(itsRect.right, itsRect.bottom);
FrameRect(itsRect);
FrameRect(itsRect);
until not Button;
until not EmptyRect(itsRect);
PenNormal;
SetPt(wPos, itsRect.left + SPACE, itsRect.top +
     LINEHEIGHT);
end;
end;
```

If that procedure looks familiar, it should. It is essentially the DrawRectangle procedure used in Chapter 3. In that chapter, this procedure is broken down into three smaller procedures, one to get the top left corner, one to get the bottom right, and one that calls the other two. That is actually a better way to write the procedure. In this case they have been combined to save some space. The final program developed by this chapter fills up a 128K Mac. One way to make programs smaller is to eliminate procedures that are only called once. Good programming style and program size are sometimes trade-offs. In general, it is better to ignore the size problem as long as you can. In this case, the resulting procedure is still fairly small and understandable, so little is lost.

Another difference between this routine and the ones in Chapter 3 is that, instead of drawing the box with FrameRect and immediately erasing it with EraseRect, the pen mode is set to srcXOr, and then the box is framed twice. When the pen is in srcXOr mode, all pixels are reversed. Drawing the same object twice in srcXOr mode leaves the screen in its original state. That has a great advantage over simply drawing and erasing the box: any drawing that was originally in that part of the screen is not changed. Therefore, srcXOr mode is a good choice in giving feedback when there is already some drawing on the screen.

The **with** statement actually contains the entire body of this procedure, because the rest of the procedure is enclosed by **begin** and **end** statements. The **with** statement is a convenience for use with records. Within a **with** statement, every variable name is checked to see if it is a field of the given record. Therefore, as the procedure begins with:

```
with aRecord do
```

every name is checked to see if it is a field of *aRecord*. For example, the lines:

```
with aRecord do
FrameRect(itsRect);
```

are exactly equivalent to:

```
FrameRect(aRecord.itsRect);
```

with has no other effect; its sole purpose is to save you from having to keep typing the record's name. Using it can make your program smaller.

Displaying the Editing Field

The following procedure may be the most difficult in this chapter. It displays the editing field, along with the text in the field.

The next procedure, ShowCell is called from DrawScreen. You must define procedures before you use them.

Notes

There is a reserved word, **forward,** which can be used so that you can call a subprogram before it is defined. This reserved word is not used in this book because you rarely need **forward**. The most common situation where it is used is where two subprograms call each other. If you need to use forward, read about it in Chapter 7 of the *Macintosh Pascal Reference Manual.*

Place an insertion point before the words "procedure DrawScreen" and type:

```
procedure ShowCell (aRecord: stringRec;
highlight: BOOLEAN);
var
n, currentLine : INTEGER;
begin
with aRecord do
begin
ClipRect(itsRect);
if highlight then
PenSize(3, 3);
FillRect(itsRect, white);
FrameRect(itsRect);
PenNormal;
if highlight then
InsetRect(itsRect, 3, 3);
ClipRect(itsRect);
MoveTo(wPos.h, wPos.v);
```

```
currentLine := 0;
for n := 1 to Length(theString) do
if theString[n] = Chr(CR) then
begin
currentLine := currentLine + 1;
MoveTo(wPos.h, wPos.v + currentLine * LINEHEIGHT);
end
else
DrawChar(theString[n]); {Predefined.}
end;
if not highlight then
ClipRect(drawingRect);
end;
```

Notice that neither of the parameters are **var** parameters. This procedure does not alter values; it only changes the appearance of the screen.

The highlight parameter changes the way the field is displayed. In general on the Macintosh, when something is selected, it is highlighted in some way, to draw attention to it. When the editing field is selected, ShowCell highlights it by drawing a thick border around it. When highlight is FALSE, a thin border is drawn around the editing field.

The predefined procedure ClipRect is called several times in ShowCell. ClipRect tells QuickDraw not to draw outside the given rectangle. The drawing is **clipped** outside the given rectangle. You can still order drawing anywhere—but what you draw is cut off and not displayed outside the ClipRect. It is this facility that makes the illusion of windows possible, because it preserves the borders of the windows.

The rest of the procedure ShowCell is much like procedure ShowString, used in *Simple Editor*.

Editing Procedures

The following procedure is essentially the same as the two procedures AddChar and ShowNewChar in *Simple Editor*. They have been combined here to make the program a bit smaller.

Place an insertion point before **begin** {main} and type:

```
procedure AddChar (typedChar : CHAR;
var aRecord : stringRec);
var
len : INTEGER;
lastChar : CHAR;
begin
with aRecord do
begin
len := Length(theString);
if len > 0 then
lastChar := theString[len];
case Ord(typedChar) of
BS :
if theString < > '' then {Note: two single quote marks.}
if lastChar = Chr(CR) then
begin
Delete(theString, len, 1);
ShowCell(aRecord, TRUE);
end
else
begin
Move(-CharWidth(lastChar), 0);
TextMode(srcXOr);
DrawChar(lastChar);
TextMode(srcOr);
Move(-CharWidth(lastChar), 0);
Delete(theString, len, 1)
end;
CR :
if len + 1 < ENDSTR then
begin
Insert(typedChar, theString, ENDSTR);
ShowCell(aRecord, TRUE);
end;
otherwise
if len + 1 < ENDSTR then
begin
DrawChar(typedChar);
Insert(typedChar, theString, ENDSTR);
end;
{if}
end;
{case}
```

```
end;
{with}
end;
{AddChar}
```

One important point about AddChar is that it is so much like the equivalent procedures in *Simple Editor*. It works on a string in a general way. It does not deal with absolute screen positions, but only relative screen positions. Therefore, AddChar works no matter whether the editing occurs in the full Drawing window, as with *Simple Editor*, or in a field that may move around the screen, as with *Field Editor*. The only difference is that AddChar in *Field Editor* calls ShowCell, rather than ShowString, and passes ShowCell a *stringRec*, instead of a simple string.

The final piece of the program is also nearly identical to a function used in *Simple Editor*. DoEditing is called when the field is selected by the user. As long as the user types, DoEditing continues executing.

The most important difference between this procedure and the DoEditing procedure in *Simple Editor* is that this one passes the event record back to the main program. It does that so that Picked can find out where the mouse was when the button was pressed.

When the user clicks the mouse button, DoEditing finishes, execution returns to the main program just after the DoEditing call, and control falls through to procedure Picked, which figures out the position of the mouse button press.

Place an insertion point before **begin** {main} and type:

```
procedure DoEditing (var aRecord : stringRec;
var event : EVENTRECORD);
begin
repeat
repeat
until GetNextEvent(mDownMask + keyDownMask, event);
if event.what < > MOUSEDOWN then
AddChar(Chr(event.message mod 256), aRecord)
until event.what = MOUSEDOWN;
GlobalToLocal(event.where);
end;
```

That is the entire program. It should now look like Figure 5-2.

```
program Field_Editor;
 const
  CR = 13;  {Character code for <RETURN>.}
  BS = 8;    {Character code for backspace.}
  LINEHEIGHT = 15;  {The distance between lines in pixels.}
  ENDSTR = 256;    {Guaranteed to be the past end of the string.}
  SPACE = 3;
  BUTHEIGHT = 20;
  BUTLEFT = 20;
  BUTWIDTH = 40;
 type
  stringRec = record
     wPos : POINT;
     theString : string;
     itsRect : RECT;
    end;
  butType = record
     stop : RECT;
    end;
  choiceType = (STOPIT, AFIELD, NOTHING);
 var
  choice : choiceType;
  aRecord : stringRec;
  buttons : butType;
  drawingRect : RECT;
  n : INTEGER;
  event : EVENTRECORD;

 procedure DrawButton (var butName : RECT;
           vertBase : INTEGER;
           theLabel : string);
 begin
  SetRect(butName, BUTLEFT, vertBase, BUTLEFT + BUTWIDTH, vertBase + BUTHEIGHT);
  FillRoundRect(butName, 6, 6, white);
  FrameRoundRect(butName, 6, 6);
  MoveTo(BUTLEFT + SPACE, vertBase + 14);
  TextFont(0);
  WriteDraw(theLabel);
  TextFont(1);
 end;

 procedure ShowCell (aRecord : stringRec;
           highlight : BOOLEAN);
 var
  n, currentLine : INTEGER;
begin
 with aRecord do
  begin
   ClipRect(itsRect);
   if highlight then
    PenSize(3, 3);
   FillRect(itsRect, white);
   FrameRect(itsRect);
   PenNormal;
   if highlight then
    InsetRect(itsRect, 3, 3);
   ClipRect(itsRect);
   MoveTo(wPos.h, wPos.v);
   currentLine := 0;
   for n := 1 to Length(theString) do
    if theString[n] = Chr(CR) then
     begin
      currentLine := currentLine + 1;
      MoveTo(wPos.h, wPos.v + currentLine * LINEHEIGHT);
     end
    else
     DrawChar(theString[n]);  {Pre-defined.}
  end;
 if not highlight then
  ClipRect(drawingRect);
end;

procedure DrawScreen (aRecord : stringRec;
        var buttons : butType);
begin
 FillRect(drawingRect, gray);
 DrawButton(buttons.stop, 310, 'Stop');
 ShowCell(aRecord, FALSE);
end;

procedure SetUpDisplay (var drawingRect : RECT);
begin
 SetRect(drawingRect, 0, 0, 532, 358);
 SetDrawingRect(drawingRect);
 ShowDrawing;
 FillRect(drawingRect, gray);
```

```
end;

procedure Picked (var choice : choiceType;
        aRecord : stringRec;
        mouse : POINT;
        buttons : butType);
 var
  n : INTEGER;

 function ButtonPressed (whichButton : RECT;
         mouse : POINT) : BOOLEAN;
 begin
  ButtonPressed := FALSE;
  if PtInRect(mouse, whichButton) then
   begin
    InvertRoundRect(whichButton, 6, 6);  {QuickDraw procedure.}
    ButtonPressed := TRUE;
   end;
 end;

begin {Picked}
 choice := NOTHING;
 if PtInRect(mouse, aRecord.itsRect) then
  choice := AFIELD
 else if ButtonPressed(buttons.stop, mouse) then
  choice := STOPIT
end;

procedure AddChar (typedChar : CHAR;
        var aRecord : stringRec);
 var
  len : INTEGER;
  lastChar : CHAR;
begin
 with aRecord do
  begin
   len := Length(theString);
   if len > 0 then
    lastChar := theString[len];
   case Ord(typedChar) of
    BS :
     if theString <> '' then  {Note: two single quote marks.}
      if lastChar = Chr(CR) then
```

```
          begin
            Delete(theString, len, 1);
            ShowCell(aRecord, TRUE);
          end
         else
          begin
            Move(-CharWidth(lastChar), 0);
            TextMode(srcXOr);
            DrawChar(lastChar);
            TextMode(srcOr);
            Move(-CharWidth(lastChar), 0);
            Delete(theString, len, 1)
          end;
      CR :
       if len + 1 < ENDSTR then
        begin
          Insert(typedChar, theString, ENDSTR);
          ShowCell(aRecord, TRUE);
        end;
      otherwise
       if len + 1 < ENDSTR then
        begin
          DrawChar(typedChar);
          Insert(typedChar, theString, ENDSTR);
        end; {if}
    end; {case}
  end; {with}
end; {AddChar}

procedure DoEditing (var aRecord : stringRec;
          var event : EVENTRECORD);
begin
 repeat
  repeat
  until GetNextEvent(mDownMask + keyDownMask, event);
  if event.what <> MOUSEDOWN then
   AddChar(Chr(event.message mod 256), aRecord)
 until event.what = MOUSEDOWN;
 GlobalToLocal(event.where);
end;

procedure ChangeTheBox (var aRecord : stringRec);
 var
```

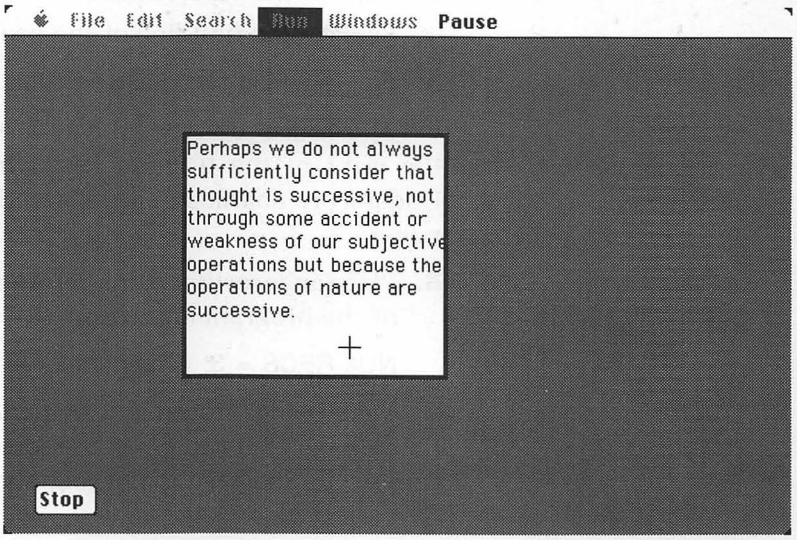
```
   event : EVENTRECORD;
 begin
  PenMode(patXOr);
  with aRecord do
   begin
    repeat
     repeat
     until GetNextEvent(mDownMask, event);
     GlobalToLocal(event.where);
     itsRect.topLeft := event.where;
     repeat
      GetMouse(itsRect.right, itsRect.bottom);
      FrameRect(itsRect);
      FrameRect(itsRect);
     until not Button;
    until not EmptyRect(itsRect);
    PenNormal;
    SetPt(wPos, itsRect.left + SPACE, itsRect.top + LINEHEIGHT);
   end;
 end;

begin  {main}
 HideAll;
 SetUpDisplay(drawingRect);
{Preceding line expands the drawing window to full screen.}
 ChangeTheBox(aRecord);
 DrawScreen(aRecord, buttons);
 choice := NOTHING;
 repeat
  case choice of
   AFIELD :
    begin
     ShowCell(aRecord, TRUE);
     DoEditing(aRecord, event);
     ShowCell(aRecord, FALSE);
    end;
   NOTHING :
    begin
     repeat
     until GetNextEvent(mDownMask, event);
     GlobalToLocal(event.where);
    end
  end;
```

```
    Picked(choice, aRecord, event.where, buttons);
   until choice = stopIt;
   SetRect(drawingRect, 293, 124, 508, 339);
   SetDrawingRect(drawingRect); {Restores drawing window.}
  end.
```

Figure 5-2

Choose Save from the File menu, and type "Field Editor" as the program name.

Run the program. When the program begins, it waits for you to draw a box on the screen. Do that the way you have before: hold the mouse button down and move the mouse down and to the right. The ChangeBox procedure rejects any "boxes" that don't enclose any space by using the EmptyRect routine. You can create an empty box accidentally by clicking the mouse button, instead of holding the button down and moving the mouse, or by moving the mouse to the left or up, instead of right and down. If you do that, ChangeBox loops back and waits for you to draw another box. Figure 5-3 shows a sample of what you can produce with this program.



```
Perhaps we do not always
sufficiently consider that
thought is successive, not
through some accident or
weakness of our subjective
operations but because the
operations of nature are
successive.
```

Figure 5-3

Type in the box, and notice how the typing that would otherwise show up outside the borders of the box is hidden by clipping. Notice that, as long as the ClipRect is set correctly, your program does not have to worry about whether or not it is drawing outside the "window." QuickDraw does that for you, automatically. It will even cut letters in pieces, so only a fraction of the letter is visible.

Click in the background and in the box again, to see how the box is selected and deselected. Notice that typing has no effect when the box is not selected.

When you are done, click in the Stop box.

Field Editor is easy to convert so that it can handle as many fields as you can fit into your Mac's memory.

Developing a Multiple Field Editor

Follow these steps to convert your program to the *Multiple Field Editor:*

1. Choose Save As from the file menu.
2. Type "Multiple Field Editor" as the new file name.
3. Click in the Save button.
4. Replace the first line of the program with:

```
program Multiple_Field_Editor;
```

Make sure you typed the underscores(_)—that is the key to the right of the 0 (zero) key, with the Shift key depressed. Underscore characters are allowed as separators in any Pascal name.

5. Place an insertion point before the "CR=13;"in the third line of the program and type:

```
NUMRECS = 3; {The number of separate records in an entry.}
```

Each field is stored as a *stringRec*-type record. NUMRECS is the number that can be on the screen at one time—in this case, it is set to 3. There is nothing about this program that limits

NUMRECS to 3. The only limit is the amount of memory in your Mac. If you, like me, have a 128K Macintosh, you won't be able to have much more than three records. If you have a 512K Mac, you can have many more. NUMRECS records are grouped together in an "entry." The final version of this program will store entries in a disk file, so that you can have a huge number of entries stored away.

6. In the **type** declaration part of the program, place an insertion point after the word "stop" and before the colon (:) in the line:

   ```
   butType = record
      stop : RECT;
   ```

   and type:

   ```
   ,new, box
   ```

   That line should now be:

   ```
   stop, new, box : RECT;
   ```

7. Add the words:

   ```
   CHANGEBOX, NEWENTRY,
   ```

   to the choiceType declaration. That line should now be:

   ```
   choiceType  =  (STOPIT,AFIELD,CHANGEBOX,NEWEN
                   TRY,NOTHING);
   ```

8. Place an insertion point *before* the first appearance of **var** (just after the last declaration in the **type** part) and type:

   ```
   entryType = array[1..NUMRECS] of stringRec;
   ```

   The constant NUMRECS is used whenever the number of records in an entry is needed. This way, all you need to do to change the number of records in an entry is to change the value of NUMRECS.

9. Place an insertion point before the first declaration in the **var** part, "choice : choiceType," and type:

   ```
   anEntry : entryType;
   thereIsAnEntry : BOOLEAN;
   which, n : INTEGER;
   ```

10. Change the following parameter definition in the parameter list of procedure DrawScreen:

```
aRecord: stringRec;
```

to:

```
anEntry: entryType; thereIsAnEntry: BOOLEAN;
```

11. Place an insertion point before the **begin** statement of procedure DrawScreen and type:

```
var
n: INTEGER;
```

12. In procedure DrawScreen, place an insertion point after the line that calls DrawButton and type:

```
DrawButton(buttons.new, 280,'New');
DrawButton(buttons.box,250,'Box');
```

13. Remove the following line from procedure DrawScreen:

```
ShowCell(aRecord, FALSE);
```

Replace it with:

```
if thereIsAnEntry then
for n:= 1 to NUMRECS do
ShowCell(anEntry[n], FALSE);
```

14. In the parameter list of procedure Picked, replace the words:

```
aRecord : stringRec;
```

with:

```
anEntry: entryType;
var which : INTEGER; thereIsAnEntry : BOOLEAN;
```

The parameter list of Picked now should be:

```
procedure Picked (var choice : choiceType;
                 anEntry : EntryType;
                 var which : INTEGER;
                 thereIsAnEntry : BOOLEAN;
                 mouse : POINT;
                 buttons : butType);
```

15. Place an insertion point before the words "ButtonPressed" and type:

```
var
n : INTEGER;
```

16. In procedure Picked, replace the lines:

```
if PtInRect(mouse, aRecord.itsRect) then
choice := aField
```

with:

```
if thereIsAnEntry then
for n := 1 to NUMRECS do
if PtInRect(mouse, anEntry[n].itsRect) then
begin
choice := AFIELD;
which:= n;
end;
```

17. After typing the above lines, without moving the insertion point, type:

```
if ButtonPressed(buttons.new, mouse) then
choice := NEWENTRY
else if ButtonPressed(buttons.box, mouse) thenchoice:=
CHANGEBOX
```

Procedure Picked should now be as shown in Figure 5-4.

18. In the main program, remove the lines:

```
ChangeTheBox(aRecord);
DrawScreen(aRecord, buttons);
```

and replace them with:

```
    thereIsAnEntry := FALSE;
which := 1;
DrawScreen(anEntry, thereIsAnEntry, buttons);
```

19. Place an insertion point after AFIELD: and type:

```
if thereIsAnEntry then
```

20. In the following lines in the main program:

```
ShowCell(aRecord, TRUE);
DoEditing(aRecord, event);
ShowCell(aRecord, FALSE);
```

replace the variable name *aRecord* with:

```
anEntry[which]
```

```
procedure Picked (var choice : choiceType;
          anEntry : entryType;
          var which : INTEGER;
          thereIsAnEntry : BOOLEAN;
          mouse : POINT;
          buttons : butType);
 var
  n : INTEGER;

 function ButtonPressed (whichButton : RECT;
           mouse : POINT) : BOOLEAN;
 begin
  ButtonPressed := FALSE;
  if PtInRect(mouse, whichButton) then
   begin
    InvertRoundRect(whichButton, 6, 6); {QuickDraw procedure.}
    ButtonPressed := TRUE;
   end;
 end;

begin {Picked}
 choice := NOTHING;
 if thereIsAnEntry then
  for n := 1 to NUMRECS do
   if PtInRect(mouse, anEntry[n].itsRect) then
    begin
     choice := AFIELD;
     which := n
    end;
 if ButtonPressed(buttons.new, mouse) then
  choice := NEWENTRY
 else if ButtonPressed(buttons.box, mouse) then
  choice := CHANGEBOX
 else if ButtonPressed(buttons.stop, mouse) then
  choice := STOPIT
end;
```

Figure 5-4

21. Place an insertion point before "NOTHING:" and type:

```
NEWENTRY:
begin
for n := 1 to NUMRECS do
begin
anEntry[n].theString:=''; {Two single quote marks.}
if not thereIsAnEntry then
begin
ChangeTheBox(anEntry[n]);
ShowCell(anEntry[n], FALSE);
end;
end;
event.where := anEntry[1].wPos;
thereIsAnEntry:= TRUE;
DrawScreen(anEntry, thereIsAnEntry, buttons);
end;
CHANGEBOX:
begin
if thereIsAnEntry then
begin
ChangeTheBox(anEntry[which]);
event.where := anEntry[which].itsRect.topLeft;
end
else
SetPt(event.where, 999,999);
DrawScreen(anEntry, thereIsAnEntry, buttons);
end;
```

22. Change the line that calls procedure Picked to:

```
Picked(choice, anEntry, which, thereIsAnEntry,
event.where, buttons);
```

23. Choose Save from the File menu.

```
program Multiple_Field_Editor;
 const
  NUMRECS = 3; {The number of separate records in an entry.}
  CR = 13;  {Character code for <RETURN>.}
  BS = 8;   {Character code for backspace.}
  LINEHEIGHT = 15; {The distance between lines in pixels.}
  ENDSTR = 256;    {Guaranteed to be the past end of the string.}
  SPACE = 3;
  BUTHEIGHT = 20;
  BUTLEFT = 20;
  BUTWIDTH = 40;
 type
  stringRec = record
    wPos : POINT;
    theString : string;
    itsRect : RECT;
   end;
  butType = record
    stop, new, box : RECT;
   end;
  choiceType = (STOPIT, AFIELD, CHANGEBOX, NEWENTRY, NOTHING);
  entryType = array[1..NUMRECS] of stringRec;
 var
  anEntry : entryType;
  thereIsAnEntry : BOOLEAN;
  which, n : INTEGER;
  choice : choiceType;
  aRecord : stringRec;
  buttons : butType;
  drawingRect : RECT;
  event : EVENTRECORD;

 procedure DrawButton (var butName : RECT;
           vertBase : INTEGER;
           theLabel : string);
 begin
  SetRect(butName, BUTLEFT, vertBase, BUTLEFT + BUTWIDTH, vertBase + BUTHEIGHT);
  FillRoundRect(butName, 6, 6, white);
  FrameRoundRect(butName, 6, 6);
  MoveTo(BUTLEFT + SPACE, vertBase + 14);
  TextFont(0);
  WriteDraw(theLabel);
  TextFont(1);
```

```
end;

procedure ShowCell (aRecord : stringRec;
        highlight : BOOLEAN);
 var
  n, currentLine : INTEGER;
begin
 with aRecord do
  begin
   ClipRect(itsRect);
   if highlight then
    PenSize(3, 3);
   FillRect(itsRect, white);
   FrameRect(itsRect);
   PenNormal;
   if highlight then
    InsetRect(itsRect, 3, 3);
   ClipRect(itsRect);
   MoveTo(wPos.h, wPos.v);
   currentLine := 0;
   for n := 1 to Length(theString) do
    if theString[n] = Chr(CR) then
     begin
      currentLine := currentLine + 1;
      MoveTo(wPos.h, wPos.v + currentLine * LINEHEIGHT);
     end
    else
     DrawChar(theString[n]);  {Pre-defined.}
  end;
 if not highlight then
  ClipRect(drawingRect);
end;

procedure DrawScreen (anEntry : entryType;
        thereIsAnEntry : BOOLEAN;
        var buttons : butType);
 var
  n : INTEGER;
begin
 FillRect(drawingRect, gray);
 DrawButton(buttons.stop, 310, 'Stop');
 DrawButton(buttons.new, 280, 'New');
 DrawButton(buttons.box, 250, 'Box');
```

```
  if thereIsAnEntry then
   for n := 1 to NUMRECS do
    ShowCell(anEntry[n], FALSE);
end;

procedure SetUpDisplay (var drawingRect : RECT);
begin
 SetRect(drawingRect, 0, 0, 532, 358);
 SetDrawingRect(drawingRect);
 ShowDrawing;
 FillRect(drawingRect, gray);
end;

procedure Picked (var choice : choiceType;
               anEntry : entryType;
               var which : INTEGER;
               thereIsAnEntry : BOOLEAN;
               mouse : POINT;
               buttons : butType);
 var
  n : INTEGER;

 function ButtonPressed (whichButton : RECT;
                 mouse : POINT) : BOOLEAN;
 begin
  ButtonPressed := FALSE;
  if PtInRect(mouse, whichButton) then
   begin
    InvertRoundRect(whichButton, 6, 6);  {QuickDraw procedure.}
    ButtonPressed := TRUE;
   end;
 end;

begin {Picked}
 choice := NOTHING;
 if thereIsAnEntry then
  for n := 1 to NUMRECS do
   if PtInRect(mouse, anEntry[n].itsRect) then
    begin
     choice := AFIELD;
     which := n
    end;
 if ButtonPressed(buttons.new, mouse) then
```

```
  choice := NEWENTRY
 else if ButtonPressed(buttons.box, mouse) then
  choice := CHANGEBOX
 else if ButtonPressed(buttons.stop, mouse) then
  choice := STOPIT
end;

procedure AddChar (typedChar : CHAR;
        var aRecord : stringRec);
 var
  len : INTEGER;
  lastChar : CHAR;
begin
 with aRecord do
  begin
   len := Length(theString);
   if len > 0 then
    lastChar := theString[len];
   case Ord(typedChar) of
    BS :
     if theString <> '' then  {Note: two single quote marks.}
      if lastChar = Chr(CR) then
       begin
        Delete(theString, len, 1);
        ShowCell(aRecord, TRUE);
       end
      else
       begin
        Move(-CharWidth(lastChar), 0);
        TextMode(srcXOr);
        DrawChar(lastChar);
        TextMode(srcOr);
        Move(-CharWidth(lastChar), 0);
        Delete(theString, len, 1)
       end;
    CR :
     if len + 1 < ENDSTR then
      begin
       Insert(typedChar, theString, ENDSTR);
       ShowCell(aRecord, TRUE);
      end;
    otherwise
     if len + 1 < ENDSTR then
```

```
      begin
       DrawChar(typedChar);
       Insert(typedChar, theString, ENDSTR);
      end; {if}
   end; {case}
  end; {with}
end; {AddChar}

procedure DoEditing (var aRecord : stringRec;
          var event : EVENTRECORD);
begin
 repeat
  repeat
  until GetNextEvent(mDownMask + keyDownMask, event);
  if event.what <> MOUSEDOWN then
   AddChar(Chr(event.message mod 256), aRecord)
 until event.what = MOUSEDOWN;
 GlobalToLocal(event.where);
end;

procedure ChangeTheBox (var aRecord : stringRec);
 var
  event : EVENTRECORD;
begin
 PenMode(patXOr);
 with aRecord do
  begin
   repeat
    repeat
    until GetNextEvent(mDownMask, event);
    GlobalToLocal(event.where);
    itsRect.topLeft := event.where;
    repeat
     GetMouse(itsRect.right, itsRect.bottom);
     FrameRect(itsRect);
     FrameRect(itsRect);
    until not Button;
   until not EmptyRect(itsRect);
   PenNormal;
   SetPt(wPos, itsRect.left + SPACE, itsRect.top + LINEHEIGHT);
  end;
end;
```

```
begin {main}
 HideAll;
 SetUpDisplay(drawingRect);
{Preceding line expands the drawing window to full screen.}
 thereIsAnEntry := FALSE;
 which := 1;
 DrawScreen(anEntry, thereIsAnEntry, buttons);
 choice := NOTHING;
 repeat
  case choice of
   AFIELD :
    if thereIsAnEntry then
     begin
      ShowCell(anEntry[which], TRUE);
      DoEditing(anEntry[which], event);
      ShowCell(anEntry[which], FALSE);
     end;
   NEWENTRY :
    begin
     for n := 1 to NUMRECS do
      begin
       anEntry[n].theString := ''; {Two single quote marks.}
       if not thereIsAnEntry then
        begin
         ChangeTheBox(anEntry[n]);
         ShowCell(anEntry[n], FALSE);
        end;
      end;
     event.where := anEntry[1].wPos;
     thereIsAnEntry := TRUE;
     DrawScreen(anEntry, thereIsAnEntry, buttons);
    end;
   CHANGEBOX :
    begin
     if thereIsAnEntry then
      begin
       ChangeTheBox(anEntry[which]);
       event.where := anEntry[which].itsRect.topLeft;
      end
     else
      SetPt(event.where, 999, 999);
     DrawScreen(anEntry, thereIsAnEntry, buttons);
    end;
```

```
      NOTHING :
       begin
        repeat
        until GetNextEvent(mDownMask, event);
        GlobalToLocal(event.where);
       end
     end;
     Picked(choice, anEntry, which, thereIsAnEntry, event.where, buttons);
    until choice = stopIt;
    SetRect(drawingRect, 293, 124, 508, 339);
    SetDrawingRect(drawingRect); {Restores drawing window.}
   end.
```

Figure 5-5

The *Multiple Field Editor* should now look like Figure 5-4.

Click in the Close box at the top left corner of the programming window to hide the window. (That frees up some memory space. This program otherwise runs out of room before reaching the HideAll call.)

Run the program.

Unlike the *Field Editor* program, *Multiple Field Editor* begins by waiting for a mouse button press to tell it what to do. The only meaningful action is pressing the New button. Then, the program waits for you to draw the required number of boxes, in this case, three. If you click the mouse in one of the boxes, that box is marked as "selected", and you can type in it. The Box button invokes the ChangeBox procedure, which then allows you to change the size and location of the most recently selected box. When you pick New, the boxes you drew before are redisplayed, with the text cleared out of them. When you click in the background, any highlighted box has the highlighting removed, and typing does nothing.

Try changing the number of boxes to some larger number—but be aware that, if you run out of memory, you may be forced to leave Pascal, so make sure you've saved the most recent version of your program.

Notes

It is a good idea to save any program in Macintosh Pascal before running it. The program may have an error that leaves you unable to save it later.

One frustrating thing about the *Multiple Field Editor* is that it just throws away the boxes you create and the text you type. Fortunately, it is pretty easy to save those entries on disk. The next chapter shows you how to do that.

Do More

1. Most word processors have a feature called "word-wrap." When a line is filled, the next word automatically moves to the next line. That is usually done by moving the first full word that can't all fit on the line.

   Try adding that feature. (Hint: You can draw the character, then find the pen position with GetMouse, and then use PtInRect to find if the pen is in the cell.) If the word won't fit in the current editing box, go back in the string to the last space and replace it with a CR. Then use ShowCell to redisplay the text.
2. Allow the user to change the font by pressing the Command key along with the < and > characters. (See the Do More section of Chapter 4 to see how you can test for the Command key.)
3. Can you think of ways to modify the program so that cells can contain more than 255 characters?

QUICK SUMMARY

This chapter develops a "field editor," which allows you to edit text in boxes on the screen. You can use the modules from this chapter for any programs that need to get text in a flexible way. The following routines, statements, and concepts are introduced.

Clipping	is when graphic information outside a given area is automatically blocked from being drawn. You can see clipping in operation any time you use a window on the Macintosh.
Data base	is a set of related information.
Enumerated type	is a programmer-defined type that can only take on certain the values listed by the programmer.
forward	is a reserved word that lets you call a procedure or function before you define it. You give the formal parameter list for the subprogram, followed by the word **forward**, but without giving the body of the subprogram. You can then define the body later. This is most often used when two subprograms call each other.
Global coordinates	in Macintosh Pascal, refers to coordinates expressed in terms of the upper left corner of the screen. The point (0,0) in global coordinates is always the upper left corner of the screen, and never moves.
GlobalToLocal	is a QuickDraw procedure that converts global coordinates to local coordinates. In Macintosh Pascal, this generally means converting from coordinates where the upper left corner of the screen is (0,0) to coordinates where the upper left corner of the Drawing window is (0,0). Note that the point doesn't move—the values used to refer to the point change.
HideAll	is a predefined procedure that hides all of Macintosh Pascal's windows.
InvertRoundRect	is a QuickDraw procedure that inverts a round-cornered rectangle.
Local coordinates	refers to coordinates expressed in terms of the current coordinate system. The position of (0,0) can move. In Macintosh Pascal, local coordinates usually are values where the upper left corner of the Drawing window is (0,0).
Local subprogram	is a subprogram that is defined within another subprogram, and can therefore only be called from within that subprogram.

LocalToGlobal	is a QuickDraw procedure that converts local coordinates to global coordinates. In Macintosh Pascal, this generally means converting from coordinates where the upper left corner of the Drawing window is (0,0) to coordinates where the upper left corner of the screen is (0,0). Note that the point doesn't move—the values used to refer to the point change.
Ordinal type	is a type whose possible values are of a limited, ordered set.
Side effect	is when a procedure or function changes the value of a variable that is not in the procedure or function's parameter list, and is not local to the procedure. (A subprogram's local variables are defined within the subprogram, and cannot be referenced outside the subprogram.) Side effects are considered bad programming style because they can create errors that are difficult to trace. It is much easier to find errors if every subprogram that modifies a variable declares that it does so by having that variable as one of its parameters.
with	is a reserved word used for a statement that lets you operate on fields of a record without writing the record's name every time. Every variable name within the **with** statement is checked to see if it is a field of the record.

CHAPTER

6

Files: A Piece of Cake

Data is stored on disk in **files**. Every program you write is stored in a file; Macintosh Pascal itself is stored in a file.

Disk files on any computer consist, on their most basic level, of a sequence of bytes. Pascal, though, imposes a few more restrictions. In Pascal, a file consists of a sequence of **components**, each of which is of the same data type.

A file can be filled with components of any data type. For example, you could fill a file with all INTEGER components, or all CHAR components, or all entryType components.

Every component has a sequence number. The first component in the file is numbered 0 (zero). You cannot read part of a component; you must read an entire component at a time. You can use the Read and Write predefined procedures to read from and write to files.

File Variables

A file is represented in Pascal by a variable, called a **file variable**. You define a type for the file variable in the same way you define a type for any variable. For example:

```
fileType = file of entryType;
```

creates a new file type which has components of the type entryType. Once you've created your file type, you can create a variable of the file type. For example:

```
fVar : fileType;
```

defines a file variable for a file filled with entryType components.

There is also a special predefined file type, called TEXT. TEXT in Macintosh Pascal is equivalent to **file of** CHAR. You define a TEXT file variable like this:

```
myFile: TEXT;
```

Macintosh Pascal program files are of type TEXT, as are MacWrite files.

File variables are unusual variables. Like all variables, the value of a file variable is initially undefined. You give a file variable an initial value by opening a file, including the file variable in the call that opens the file.

You then can use the file variable in calls to Read and/or Write, to get components from the file or put components into the file. The file variable always points to the current component of the file. It is automatically moved to the next component when you read or write to the file.

Every file can only have one variable at a time, and only one current component. For that reason, you must pass file variables as **var** parameters, and you cannot assign the value of a file variable to another variable, even one of the same type.

The file variable is actually a kind of "window" on the file, through which you can see one component at a time, as illustrated in Figure 6-1. You can get that component directly by using a caret (^) following the file variable's name. For example:

```
anEntry := fVar^;
```

File Procedures

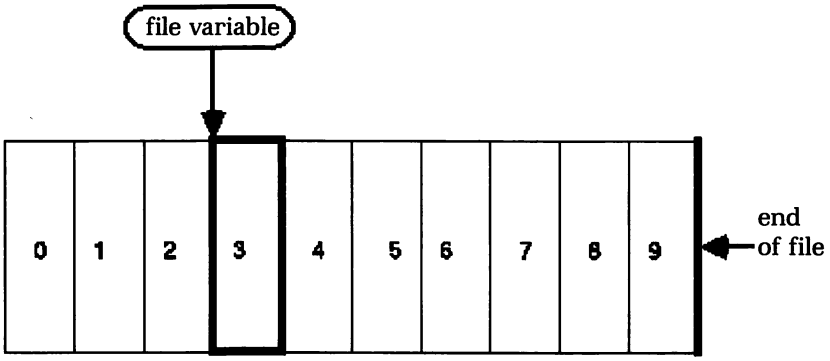


Figure 6-1 File Variables

You can move the file variable to the next component with the predefined **Get** procedure:

```
Get(fVar);
```

The predefined Read procedure combines the action of the above two statements:

```
Read(fVar, anEntry);
```

That call places the current component in *anEntry* and moves the file variable so that it looks at the next component. In either case, the file itself is not changed; the component is merely copied to the variable *anEntry*.

If there are no more components, however, the value of *fVar* is undefined. It is therefore an error to try to call Read on the file, or to try to assign the value of fVar^ to another variable. The predefined **EOF** function returns TRUE if you have reached the end of the file. If is often used like this:

```
If not EOF(fVar) then
Read(fVar, anEntry);
```

You can put new components in the file in a similar way. For example:

```
fVar^{= anEntry;
Put(fVar);
```

which is equivalent to:

```
Write(fVar, anEntry);
```

The **Put** and Write procedures move the file variable to the next component, if there is one, or to the end of the file, if there are no more components. There is always room to put a new component in a file; the file is automatically expanded to make room for it. If EOF is TRUE (that is, you are at the end of the file) the new component is added onto the end of the file. Otherwise, the new component replaces the current component.

You can also find out the component number of the current component with the FilePos function, and move to a particular component with the Seek procedure.

When using Read and Write with files of type TEXT, you can give any number of variables, or, in the case of Write, constants. For example, assume that *fText* is a file variable of type TEXT. The following call is legal:

```
Write(fText, aNumber, aLetter, 'T.S.Eliot','Ezra Pound');
```

Notice that this writes twenty-one components to the file, assuming that *aNumber* and *aLetter* each hold a single character. Figure 6-2 illustrates what the file might contain after this call, assuming that *aNumber* contains a '2' and *aLetter* contains a 'B'. Because this is a Write call, *fText* is positioned past the last character written, at the end of the file. Also, notice that the file only contains one space—the one between Ezra and Pound. Spaces are not automatically written into the file. You have to include them yourself.

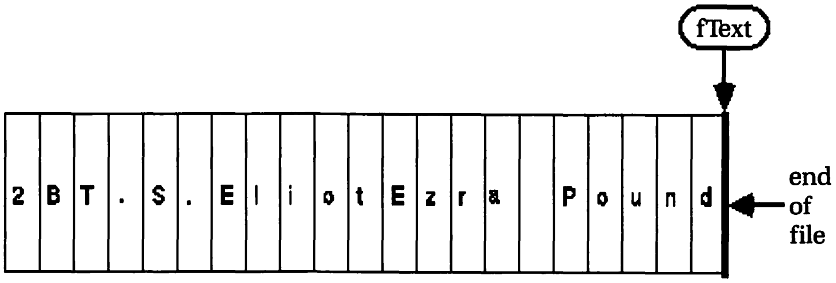


Figure 6-2 File After Write

There are also two procedures and a function that can be used only with files of type TEXT. They are:

- The WriteLn procedure, which acts exactly like Write, except that it puts an end-of-line character in the file after writing all the data contained in the WriteLn call. An end-of-line character is a non-printing character like the Backspace character or the Return character, except that it indicates the end of a line.
- The ReadLn procedure, which reads from the current component to the next end-of-line character, or the end of the file.
- The EOLN function, which returns TRUE when the file variable is at an end-of-line character.

Reading or writing with a file of type TEXT is just like reading or writing the Text window. In fact, the Text window is a file of type TEXT to Pascal.

There are two distinct classes of files in Macintosh Pascal: **normal files** and **anonymous files**. Normal files have names, and appear in the disk directory. They have icons associated with them, so you can copy them to other disks, or do anything else you can do with Macintosh files. Anonymous files do not have names, and are only temporary. When your program ends, they are destroyed, and any data in them is lost.

You cannot delete normal files in Macintosh Pascal. The only way to get rid of one is to dump its icon into the Trash. You also cannot delete individual components. You can, though, completely erase a normal file, and remove all of its components.

You begin using a file by opening it. There are three predefined procedures that open files. They are:

- **Reset**. After you open a file with Reset, you can read from a file, but you can't write to it. You can use the Read or Get procedures.
- **Rewrite**. After you open a file with Rewrite, its previous contents are erased, and you can only write to the file. You cannot read from it. You can use the Write or Put procedures.
- **Open** After you call Open, you can read from or write to a file. You can use the Read, Write, Put, or Get procedures.

You can also call Reset or Rewrite for a file that is already open. If the file was originally opened with Reset or Rewrite, the effect is the same as if the file is first closed and then newly opened with the Reset or Rewrite procedure you just called. If the file was originally opened with Open, Reset brings you back the beginning of the file, and allows you to continue writing or reading components. Rewrite erases the entire file, but as long as you originally opened the file with Open, you can still read components from the file after you've written them.

You can also call the Close procedure, which closes the file. After that, the file variable is again undefined. You can then call Reset, Open, or Rewrite, which all act as if the file had never been opened.

You can call Open or Rewrite for a file that does not yet exist. The file is automatically created.

You must supply a file name when you open a normal file. (You do not give a name when you open an anonymous file.)

There are two routines to get file names from the user:

- OldFileName displays the same dialog box you get when you choose Open from the Macintosh Pascal File menu. It displays all the files on the disk, and lets you choose one, also letting you eject the disk and put in a new one. The name you choose is passed back to the program, and can then be used in a call to Open, Rewrite, or Reset.
- NewFileName displays the same dialog box you get when you choose Save As from the file menu. It allows the user to type in a file name, and returns that name to your program for use in Open, Rewrite, or Reset.

Modifying the Multiple Field Editor

The following steps alter the *Multiple Field Editor* so that it saves each entry in a file.

The program begins by getting a file name from the user. It first displays the OldFileName dialog box, showing the files on your disk.

Warning

The results of opening a file that does not have components of the expected type are uncertain; Macintosh Pascal will open the file, and will interpret the bytes in the file as if they formed components of the file variable's type. If the components are not actually of that type, the results could make Macintosh Pascal crash. If you tried to write to the file, you could destroy the file.

If the user selects the Cancel button, the NewFileName dialog box is displayed, and the user can type in a new file name.

Whether it is an old or new file, the program opens it. The program then tries to get the first component in the file. If there aren't any components, the program waits for a mouse button press. When you press the New button, a new entry is created.

If the file existed before and has components, the first component is read and displayed. Clicking in the background, so that Picked returns a choiceType of NOTHING, causes the next component to be read. When the end of the file is reached, the file is reset so that it loops around and displays the first record again.

As an additional feature, there is a Clean button, which "cleans up" the file by removing blank entries.

That covers the basic methodology of these changes. The specifics are explained along with the changes. Follow these steps:

1. Add the word CLEANUP to the definition of choiceType so that definition looks like:

```
choiceType = (STOPIT, NEWENTRY, AFIELD, CLEANUP,
CHANGEBOX, NOTHING);
```

In this case, the order of the choiceType constants is unimportant. In general, though, the order of enumerated type constants can be significant.

2. Place an insertion point before the first **var**. Type:

```
fileType = file of entryType;
```

3. Add the name "clean" to the list of RECTs defined in the butType record type at the beginning of the program, so that the list looks like:

```
stop, new, clean, box : RECT;
```

The order is unimportant.

4. Place an insertion point before procedure DrawButton. Type:

```
fVar: fileType;
aNewEntry: BOOLEAN;
procedure GetFileName (var fVar : fileType);
var
fileName : string;
begin
repeat
fileName := OldFileName('Choose Cancel if for a new file.');
if fileName = '' then {Two single quote marks}
fileName := NewFileName('Give a new file name.');
until fileName < > ''; {Two single quote marks}
Open(fVar, fileName);
end;
```

The text strings contained in the calls to the predefined functions OldFileName and NewFileName are displayed on the screen in the corresponding dialog boxes, to tell the user what is wanted.

5. Place the insertion point in procedure DrawScreen, before the **if**. Type:

```
DrawButton(buttons.clean, 220, 'Clean');
```

6. In procedure Picked, place an insertion point before any one of the **else** words and type:

```
else if ButtonPressed(buttons.clean, mouse) then
choice:= CLEANUP
```

7. Place an insertion point before **begin** {main} and type:

```
procedure SaveEntry (thereIsAnEntry, aNewEntry :
  BOOLEAN;
anEntry : entryType; var fVar : fileType);
begin
if thereIsAnEntry then
begin
if aNewEntry then
Seek(fVar, MAXINT)
else
Seek(fVar, FilePos(fVar) - 1);
Write(fVar, anEntry);
end;
end;
```

This procedure is called to save the current entry before a new entry is displayed on the screen. The predefined Seek procedure moves the file variable to a specific component. Note that if you write to a component that already exists, the existing one is destroyed, and replaced by the one you just wrote. If this is a new entry, the only place you can put it without destroying another entry is at the end of the file. You can always find the end of the file by Seek-ing a component that is beyond the end. MAXINT, a predefined constant that holds the largest INTEGER value, is 32767. Chances are that Seek-ing MAXINT puts you at the end of the file.

If the entry is not new, it should be placed back where it came from, so that the old copy is replaced by the new copy. Every time you read from or write to a file, the file variable is moved so it points to the next component. To put a component back where it came from, you need to move the file variable back one. The predefined procedure FilePos finds the component number of the component at which the file variable is pointing. Putting all that together, the line:

```
Seek(fVar, FilePos(fVar) - 1);
```

moves the file variable *fVar* back one.

8. Leave the insertion point where it is (before **begin** {main}) and type:

```
procedure GetNextEntry (var anEntry : entryType;
var fVar : fileType;
var thereIsAnEntry : BOOLEAN);
var
n : INTEGER;
begin
if EOF(fVar) then
Reset(fVar);
if EOF(fVar) then
thereIsAnEntry := FALSE
else
begin
thereIsAnEntry := TRUE;
Read(fVar, anEntry);
end;
end;
```

This procedure is called whenever you click the mouse button in the background, which results in "choice:=NOTHING". It gets the next entry in the file, if there is one.

The predefined function EOF is TRUE when the file variable is past the last component in the file. If EOF is TRUE, Reset is used to move the file variable to the beginning of the file, component 0. If EOF is *still* TRUE, that means the file is empty, and there are no components. The variable *thereIsAnEntry* is set to FALSE to communicate that fact back to the main program. Otherwise, the next component is read.

Note that there is no need to explicitly move the file variable to the next component. Whenever you Read from the file or Write to the file, the file variable is moved so it points to the next component.

9. Leave the insertion point where it is (before **begin** {main}) and type:

```
procedure CleanUpFile (var fVar : fileType);
var
tempFile : fileType;
transferEntry : entryType;
function BlankEntry (anEntry : entryType) : BOOLEAN;
var
blank : BOOLEAN;
n : INTEGER;
begin
blank := TRUE;
for n := 1 to NUMRECS do
blank := (anEntry[n].theString = '') and blank;
BlankEntry := blank;
end;
begin
Rewrite(tempFile);
Reset(fVar);
while not EOF(fVar) do
begin
Read(fVar, transferEntry);
if not BlankEntry(transferEntry) then
Write(tempFile, transferEntry);
end;
Reset(tempFile);
Rewrite(fVar);
while not EOF(tempFile) do
begin
Read(tempFile, transferEntry);
Write(fVar, transferEntry);
end;
Reset(fVar);
end;
```

This procedure removes blank entries from the file. An important reason for including this subprogram is that, if you are allowing the user to store data permanently in a file, you should provide some way to delete file entries. In this case, if the strings in an entry are all blank, it is assumed that the user wants that entry removed.

An anonymous file is used as an intermediary for cleaning up the original file. An anonymous file, you may remember, is not permanent, but exists only while it is used. The entries in the original file are each examined. As long as at least one of the strings in an entry contains something, the entry is written to the anonymous file. If any entry contains only blank strings, it is not copied. When the end of the file is reached, Rewrite is called for the original file. Rewrite erases the file, and sets *fVar* to component 0, the beginning of the file. The last part of CleanUp copies the contents of the anonymous file back to the permanent file. Reset, called several times in CleanUp, brings the file variable back to the beginning of the file. It has the same effect as:

```
Seek(fVar,0);
```

10. Place an insertion point after the line:

```
HideAll;
```

and type:

```
GetFileName(fVar);
```

11. Place an insertion point after the lines:

```
NEWENTRY :
begin
```

and type:

```
SaveEntry(thereIsAnEntry, aNewEntry, anEntry,
  fVar);
```

12. In the main program, place an insertion point after the line:

```
event.where := anEntry[1].wPos;
```

and type:

```
aNewEntry := TRUE;
```

13. In the main program, place an insertion point before the line "NOTHING :" and type:

```
CLEANUP:
begin
SaveEntry(thereIsAnEntry, aNewEntry, anEntry, fVar);
CleanUpFile(fVar);
GetNextEntry(anEntry, fVar, thereIsAnEntry);
DrawScreen(anEntry, thereIsAnEntry, buttons);
event.where := anEntry[1].wPos;
end;
```

14. In the main program, place an insertion point after the lines:

```
NOTHING :
begin
```

and type:

```
SaveEntry(thereIsAnEntry, aNewEntry, anEntry, fVar);
GetNextEntry(anEntry, fVar, thereIsAnEntry);
aNewEntry := FALSE;
DrawScreen(anEntry, thereIsAnEntry, buttons);
```

Notice that, because *thereIsAnEntry* is passed to DrawScreen, the program displays the cells only if an entry was found by GetNextEntry.

15. Place an insertion point after the line:

```
until choice = STOPIT;
```

and type:

```
SaveEntry(thereIsAnEntry, aNewEntry, anEntry,
  fVar);Close(fVar);
```

16. Choose Save from the File menu.
The program should now look like Figure 6-3.

```
program Multiple_Field_Editor;
 const
  NUMRECS = 3; {The number of separate records in an entry.}
  CR = 13;  {Character code for <RETURN>.}
  BS = 8;   {Character code for backspace.}
  LINEHEIGHT = 15; {The distance between lines in pixels.}
  ENDSTR = 256;   {Guaranteed to be the past end of the string.}
  SPACE = 4;
  BUTHEIGHT = 20;
  BUTLEFT = 20;
  BUTWIDTH = 40;
type
 stringRec = record
   wPos : POINT;
   theString : string;
   itsRect : RECT;
  end;
 butType = record
   stop, new, clean, box : RECT;
  end;
 choiceType = (STOPIT, NEWENTRY, AFIELD, CLEANUP, CHANGEBOX, NOTHING);
 entryType = array[1..NUMRECS] of stringRec;
 fileType = file of entryType;
var
 anEntry : entryType;
 thereIsAnEntry : BOOLEAN;
 which, n : INTEGER;
 choice : choiceType;
 aRecord : stringRec;
 buttons : butType;
 drawingRect : RECT;
 event : EVENTRECORD;
 fVar : fileType;
 aNewEntry : BOOLEAN;
procedure GetFileName (var fVar : fileType);
 var
  fileName : string;
begin
 repeat
  fileName := OldFileName('Choose Cancel if for a new file.');
  if fileName = '' then
   fileName := NewFileName('Give a new file name.');
 until fileName <> '';
 Open(fVar, fileName);
end;

procedure DrawButton (var butName : RECT;
          vertBase : INTEGER;
          theLabel : string);
begin
 SetRect(butName, BUTLEFT, vertBase, BUTLEFT + BUTWIDTH, vertBase + BUTHEIGHT);
 FillRoundRect(butName, 6, 6, white);
 FrameRoundRect(butName, 6, 6);
 MoveTo(BUTLEFT + SPACE, vertBase + 14);
 TextFont(0);
 WriteDraw(theLabel);
 TextFont(1);
end;

procedure ShowCell (aRecord : stringRec;
          highlight : BOOLEAN);
 var
  n, currentLine : INTEGER;
begin
 with aRecord do
  begin
   ClipRect(itsRect);
   if highlight then
    PenSize(3, 3);
   FillRect(itsRect, white);
   FrameRect(itsRect);
   PenNormal;
   if highlight then
    InsetRect(itsRect, 3, 3);
   ClipRect(itsRect);
   MoveTo(wPos.h, wPos.v);
   currentLine := 0;
   for n := 1 to Length(theString) do
    if theString[n] = Chr(CR) then
     begin
      currentLine := currentLine + 1;
      MoveTo(wPos.h, wPos.v + currentLine * LINEHEIGHT);
     end
    else
     DrawChar(theString[n]);  {Pre-defined.}
  end;
 if not highlight then
  ClipRect(drawingRect);
end;

procedure DrawScreen (anEntry : entryType;
         thereIsAnEntry : BOOLEAN;
         var buttons : butType);
 var
  n : INTEGER;
begin
 FillRect(drawingRect, gray);
 DrawButton(buttons.stop, 310, 'Stop');
 DrawButton(buttons.new, 280, 'New');
 DrawButton(buttons.box, 250, 'Box');
 DrawButton(buttons.clean, 220, 'Clean');
 if thereIsAnEntry then
  for n := 1 to NUMRECS do
   ShowCell(anEntry[n], FALSE);
end;

procedure SetUpDisplay (var drawingRect : RECT);
begin
 SetRect(drawingRect, 0, 0, 532, 358);
 SetDrawingRect(drawingRect);
 ShowDrawing;
 FillRect(drawingRect, gray);
end;

procedure Picked (var choice : choiceType;
         anEntry : entryType;
         var which : INTEGER;
         thereIsAnEntry : BOOLEAN;
         mouse : POINT;
         buttons : butType);
 var
  n : INTEGER;

 function ButtonPressed (whichButton : RECT;
          mouse : POINT) : BOOLEAN;
 begin
  ButtonPressed := FALSE;
  if PtInRect(mouse, whichButton) then
```

```
        begin
        InvertRoundRect(whichButton, 6, 6);  {QuickDraw procedure.}
        ButtonPressed := TRUE;
      end;
  end;

begin {Picked}
  choice := NOTHING;
  if thereIsAnEntry then
    for n := 1 to NUMRECS do
      if PtInRect(mouse, anEntry[n].itsRect) then
        begin
          choice := AFIELD;
          which := n
        end;
  if ButtonPressed(buttons.new, mouse) then
    choice := NEWENTRY
  else if ButtonPressed(buttons.box, mouse) then
    choice := CHANGEBOX
  else if ButtonPressed(buttons.clean, mouse) then
    choice := CLEANUP
  else if ButtonPressed(buttons.stop, mouse) then
    choice := STOPIT
end;

procedure AddChar (typedChar : CHAR;
            var aRecord : stringRec);
  var
    len : INTEGER;
    lastChar : CHAR;
begin
  with aRecord do
    begin
      len := Length(theString);
      if len > 0 then
        lastChar := theString[len];
      case Ord(typedChar) of
        BS :
          if theString <> '' then  {Note: two single quote marks.}
            if lastChar = Chr(CR) then
              begin
                Delete(theString, len, 1);
                ShowCell(aRecord, TRUE);
              end
      else
       begin
        Move(-CharWidth(lastChar), 0);
        TextMode(srcXOr);
        DrawChar(lastChar);
        TextMode(srcOr);
        Move(-CharWidth(lastChar), 0);
        Delete(theString, len, 1)
       end;
    CR :
     if len + 1 < ENDSTR then
      begin
       Insert(typedChar, theString, ENDSTR);
       ShowCell(aRecord, TRUE);
      end;
    otherwise
     if len + 1 < ENDSTR then
      begin
       DrawChar(typedChar);
       Insert(typedChar, theString, ENDSTR);
      end; {if}
   end; {case}
  end; {with}
end; {AddChar}

procedure DoEditing (var aRecord : stringRec;
        var event : EVENTRECORD);
begin
 repeat
  repeat
  until GetNextEvent(mDownMask + keyDownMask, event);
  if event.what <> MOUSEDOWN then
   AddChar(Chr(event.message mod 256), aRecord)
 until event.what = MOUSEDOWN;
 GlobalToLocal(event.where);
end;

procedure ChangeTheBox (var aRecord : stringRec);
 var
  event : EVENTRECORD;
begin
 PenMode(patXOr);
 with aRecord do
```

```
  begin
   repeat
    repeat
    until GetNextEvent(mDownMask, event);
    GlobalToLocal(event.where);
    itsRect.topLeft := event.where;
    repeat
     GetMouse(itsRect.right, itsRect.bottom);
     FrameRect(itsRect);
     FrameRect(itsRect);
    until not Button;
   until not EmptyRect(itsRect);
   PenNormal;
   SetPt(wPos, itsRect.left + SPACE, itsRect.top + LINEHEIGHT);
  end;
end;

procedure SaveEntry (thereIsAnEntry, aNewEntry : BOOLEAN;
          anEntry : entryType;
          var fVar : fileType);
begin
 if thereIsAnEntry then
  begin
   if aNewEntry then
    Seek(fVar, MAXINT)
   else
    Seek(fVar, FilePos(fVar) - 1);
   Write(fVar, anEntry);
  end;
end;

procedure GetNextEntry (var anEntry : entryType;
          var fVar : fileType;
          var thereIsAnEntry : BOOLEAN);
 var
  n : INTEGER;
begin
 if EOF(fVar) then
  Reset(fVar);
 if EOF(fVar) then
  thereIsAnEntry := FALSE
 else
  begin
```

```
      thereIsAnEntry := TRUE;
      Read(fVar, anEntry);
    end;
end;

procedure CleanUpFile (var fVar : fileType);
  var
    tempFile : fileType;
    transferEntry : entryType;

  function BlankEntry (anEntry : entryType) : BOOLEAN;
    var
      blank : BOOLEAN;
      n : INTEGER;
  begin
    blank := TRUE;
    for n := 1 to NUMRECS do
      blank := (anEntry[n].theString = '') and blank;
    BlankEntry := blank;
  end;

begin
  Rewrite(tempFile);
  Reset(fVar);
  while not EOF(fVar) do
    begin
      Read(fVar, transferEntry);
      if not BlankEntry(transferEntry) then
        Write(tempFile, transferEntry);
    end;
  Reset(tempFile);
  Rewrite(fVar);
  while not EOF(tempFile) do
    begin
      Read(tempFile, transferEntry);
      Write(fVar, transferEntry);
    end;
  Reset(fVar);
end;

begin  {main}
  HideAll;
  GetFileName(fVar);
```

```
 SetUpDisplay(drawingRect);
{Preceding line expands the drawing window to full screen.}
 thereIsAnEntry := FALSE;
 which := 1;
 DrawScreen(anEntry, thereIsAnEntry, buttons);
 choice := NOTHING;
 repeat
  case choice of
   AFIELD :
    if thereIsAnEntry then
     begin
      ShowCell(anEntry[which], TRUE);
      DoEditing(anEntry[which], event);
      ShowCell(anEntry[which], FALSE);
     end;
   NEWENTRY :
    begin
     SaveEntry(thereIsAnEntry, aNewEntry, anEntry, fVar);
     for n := 1 to NUMRECS do
      begin
       anEntry[n].theString := '';  {Two single quote marks.}
       if not thereIsAnEntry then
        begin
         ChangeTheBox(anEntry[n]);
         ShowCell(anEntry[n], FALSE);
        end;
      end;
     event.where := anEntry[1].wPos;
     aNewEntry := TRUE;
     thereIsAnEntry := TRUE;
     DrawScreen(anEntry, thereIsAnEntry, buttons);
    end;
   CHANGEBOX :
    begin
     if thereIsAnEntry then
      begin
       ChangeTheBox(anEntry[which]);
       event.where := anEntry[which].itsRect.topLeft;
      end
     else
      SetPt(event.where, 999, 999);
     DrawScreen(anEntry, thereIsAnEntry, buttons);
    end;
```

```
      CLEANUP :
        begin
          SaveEntry(thereIsAnEntry, aNewEntry, anEntry, fVar);
          CleanUpFile(fVar);
          GetNextEntry(anEntry, fVar, thereIsAnEntry);
          DrawScreen(anEntry, thereIsAnEntry, buttons);
          event.where := anEntry[1].wPos;
        end;
      NOTHING :
        begin
          SaveEntry(thereIsAnEntry, aNewEntry, anEntry, fVar);
          GetNextEntry(anEntry, fVar, thereIsAnEntry);
          aNewEntry := FALSE;
          DrawScreen(anEntry, thereIsAnEntry, buttons);
          repeat
          until GetNextEvent(mDownMask, event);
          GlobalToLocal(event.where);
        end
    end;
    Picked(choice, anEntry, which, thereIsAnEntry, event.where, buttons);
  until choice = stopIt;
  SaveEntry(thereIsAnEntry, aNewEntry, anEntry, fVar);
  Close(fVar);
  SetRect(drawingRect, 293, 124, 508, 339);
  SetDrawingRect(drawingRect);  {Restores drawing window.}
end.
```

Figure 6-3

Before you run the program, check the value of NUMRECS, at the beginning of the program. It should be 3. A 128K Mac will not be able to run the program with more records in an entry. You can always use a smaller number. You can also use a larger number if you have a 512K Mac, but use 3 for the moment.

Also, click in the Close box in the upper left corner of the program window. This program is so big that Macintosh Pascal runs out of room before it gets to the HideAll statement at the beginning of the program.

Run the program. The first time you run it, you have no files on your disk of the right type, so you should press the Cancel button when you are asked to pick an existing file. You are then asked for a new file name. Type in some name that you'll associate with this program.

Like the previous version, this program begins by waiting for you to choose a button. Choose New. Draw three boxes. Type something in at least one of the boxes. Choose New again. Notice that the new record has the same box sizes and positions as the last record. You may have heard a quick hum from your disk drive when you clicked on New the second time. That was the sound of the Mac storing the first entry. Type something in one of the second entry's fields. Click on New a third time, but don't type anything in any of that entry's fields. Now click in the Clean button.

The disk drive will hum for a few seconds, and then the first record in the file will be displayed.

If you click in the background, with the mouse on the gray part of the window, the next record is displayed. If you click once more, the first record is again displayed.

The blank record has been deleted.

Select one of the fields. Click on the Box button. The Mac is now waiting for you to specify a new box. Draw one in the usual way: by holding the mouse button down and moving it down and to the right. When you let go of the mouse button the screen is redisplayed with the selected field shown with its new size.The new size will be stored away, and that record will always appear with that new size.

You now have a simple data base system working on your Mac. You can save as many records as will fit on a disk, and you can customize the appearance of the screen as you desire.

The last section in this chapter shows you how to make a module that sorts your records in order. The next chapter shows you how to use this program to create a telephone book, and then use the Mac's sound generator to produce telephone dialing tones.

If you want to set up your telephone book now, make sure you only have one telephone number to a box. The dialing program will dial all numbers in a single box. If you have two numbers in a box, the dialing program will try to dial both.

Warning

If you create a large data file, you may run out of space on your disk. You may want to move some of the programs on your Pascal disk to another disk before creating a large data base.

The next section creates the framework that will be used for the sorting and dialing programs.

Creating the Basic Framework

Although the program you just created allows you to make your own data base, it doesn't really do very much. The problem is that the Mac's memory limits how much can be done by a single program.

The solution is to create a set of compatible "modules" that can process the data base. This section shows you how to strip down the *Multiple Field Editor* program so that it provides a framework for all the modules. The framework will be able to open a file and read records from it.

The next section shows how to create a sorting module and how to add it to the framework. Chapter 7 shows how to create a module that dials a telephone using numbers stored in the data base. Chapter 8 shows how to print the contents of the data base.

Notes

If you have a Macintosh with 512K or more of memory, you can combine these modules together into a single program that does everything.

First, make sure you have saved the last version of the *Multiple Field Editor.*

You should have the *Multiple Field Editor* program displayed in the Macintosh Pascal programming window.

1. Choose Save As... from the File menu.
2. Type "Database Framework" as the new file name, and click in the Save button.
3. Change the program name in the first line of the program to:

   ```
   Database_Framework
   ```

4. Remove the following from the list of RECTs defined in the definition of butType in the first **type** section:

   ```
   ,new, clean, box
   ```

   That line should now be:

   ```
   stop : RECT;
   ```

5. Remove the following from the definition of choiceType:

   ```
   NEWENTRY,
   ```

   Also remove the following from choiceType:

   ```
   CLEANUP, CHANGEBOX,
   ```

   The definition of choiceType should now be:

   ```
   choiceType = (STOPIT, AFIELD, NOTHING);
   ```

6. Remove the following definition from the variables defined in the first **var** section:

   ```
   aNewEntry: BOOLEAN;
   ```

7. In procedure GetFileName, remove the line within the parentheses in the call to OldFileName:

   ```
   'Choose Cancel if for a new file.'
   ```

   and replace it with:

   ```
   'Choose a file.'
   ```

8. Remove the following lines from procedure GetFileName:

   ```
   if fileName = '' then
   fileName := NewFileName('Give a new file name.');
   ```

9. Remove the following from lines from procedure DrawScreen:

```
DrawButton(buttons.new, 280, 'New');
DrawButton(buttons.box, 250, 'Box');
DrawButton(buttons.clean, 220, 'Clean');
```

10. Delete the entire procedure AddChar.
11. Delete the entire procedure DoEditing.
12. Delete the entire procedure ChangeTheBox.
13. Delete the entire procedure SaveEntry.
14. Delete the entire procedure CleanUpFile, including the function BlankEntry.
15. Remove the following from procedure Picked:

```
if ButtonPressed(buttons.new, mouse) then
choice := NEWENTRY
else if ButtonPressed(buttons.box, mouse) then
choice := CHANGEBOX;
else if ButtonPressed(buttons.clean, mouse)      then
choice := CLEANUPelse
```

16. Remove the following case constants and all of their associated statements from the **case** statement in the main program:

```
NEWENTRY:
CHANGEBOX:
CLEANUP:
```

(AFIELD and NOTHING should be the only case constants remaining.)

17. Replace the following line in the compound statement following the case constant AFIELD:

```
DoEditing(anEntry[which], event);
```

with:

```
repeat
until GetNextEvent(mDownMask, event);
GlobalToLocal(event.where);
```

18. Remove the following line from the two places it appears in the main program, where it follows the NOTHING case constant, and as the fourth line from the end:

SaveEntry(thereIsAnEntry, aNewEntry, anEntry, fVar);

19. Remove the line in the section following the case constant NOTHING that refers to *aNewEntry*.

20. Choose Save from the File menu.

The program should now be as shown in Figure 6-4.

```
program Database_Framework;
 const
  NUMRECS = 3;  {The number of separate records in an entry.}
  CR = 13;  {Character code for <RETURN>.}
  BS = 8;     {Character code for backspace.}
  LINEHEIGHT = 15;  {The distance between lines in pixels.}
  ENDSTR = 256;     {Guaranteed to be the past end of the string.}
  SPACE = 3;
  BUTHEIGHT = 20;
  BUTLEFT = 20;
  BUTWIDTH = 40;
 type
  stringRec = record
     wPos : POINT;
     theString : string;
     itsRect : RECT;
   end;
  butType = record
     stop : RECT;
   end;
  choiceType = (STOPIT, AFIELD, NOTHING);
  entryType = array[1..NUMRECS] of stringRec;
  fileType = file of entryType;
 var
  anEntry : entryType;
  thereIsAnEntry : BOOLEAN;
  which, n : INTEGER;
  choice : choiceType;
  aRecord : stringRec;
  buttons : butType;
  drawingRect : RECT;
  event : EVENTRECORD;
  fVar : fileType;
```

```
procedure GetFileName (var fVar : fileType);
 var
  fileName : string;
begin
 repeat
  fileName := OldFileName('Choose a file.');
 until fileName <> '';
 Open(fVar, fileName);
end;
procedure DrawButton (var butName : RECT;
          vertBase : INTEGER;
          theLabel : string);
begin
 SetRect(butName, BUTLEFT, vertBase, BUTLEFT + BUTWIDTH, vertBase + BUTHEIGHT);
 FillRoundRect(butName, 6, 6, white);
 FrameRoundRect(butName, 6, 6);
 MoveTo(BUTLEFT + SPACE, vertBase + 14);
 TextFont(0);
 WriteDraw(theLabel);
 TextFont(1);
end;

procedure ShowCell (aRecord : stringRec;
          highlight : BOOLEAN);
 var
  n, currentLine : INTEGER;
begin
 with aRecord do
  begin
   ClipRect(itsRect);
   if highlight then
    PenSize(3, 3);
   FillRect(itsRect, white);
   FrameRect(itsRect);
   PenNormal;
   if highlight then
    InsetRect(itsRect, 3, 3);
   ClipRect(itsRect);
   MoveTo(wPos.h, wPos.v);
   currentLine := 0;
   for n := 1 to Length(theString) do
    if theString[n] = Chr(CR) then
     begin
      currentLine := currentLine + 1;
      MoveTo(wPos.h, wPos.v + currentLine * LINEHEIGHT);
     end
     else
      DrawChar(theString[n]);  {Pre-defined.}
   end;
 if not highlight then
  ClipRect(drawingRect);
  end;

  procedure DrawScreen (anEntry : entryType;
            thereIsAnEntry : BOOLEAN;
            var buttons : butType);
   var
    n : INTEGER;
  begin
   FillRect(drawingRect, gray);
   DrawButton(buttons.stop, 310, 'Stop');
   if thereIsAnEntry then
    for n := 1 to NUMRECS do
     ShowCell(anEntry[n], FALSE);
  end;

  procedure SetUpDisplay (var drawingRect : RECT);
  begin
   SetRect(drawingRect, 0, 0, 532, 358);
   SetDrawingRect(drawingRect);
   ShowDrawing;
   FillRect(drawingRect, gray);
  end;

  procedure Picked (var choice : choiceType;
            anEntry : entryType;
            var which : INTEGER;
            thereIsAnEntry : BOOLEAN;
            mouse : POINT;
            buttons : butType);
   var
    n : INTEGER;

   function ButtonPressed (whichButton : RECT;
             mouse : POINT) : BOOLEAN;
   begin
    ButtonPressed := FALSE;
    if PtInRect(mouse, whichButton) then
     begin
      InvertRoundRect(whichButton, 6, 6);  {QuickDraw procedure.}
      ButtonPressed := TRUE;
     end;
   end;
```

```
begin {Picked}
 choice := NOTHING;
 if thereIsAnEntry then
  for n := 1 to NUMRECS do
   if PtInRect(mouse, anEntry[n].itsRect) then
    begin
     choice := AFIELD;
     which := n
    end;
 if ButtonPressed(buttons.stop, mouse) then
  choice := STOPIT
end;

procedure GetNextEntry (var anEntry : entryType;
         var fVar : fileType;
         var thereIsAnEntry : BOOLEAN);
 var
  n : INTEGER;
begin
 if EOF(fVar) then
  Reset(fVar);
 if EOF(fVar) then
  thereIsAnEntry := FALSE
 else
  begin
   thereIsAnEntry := TRUE;
   Read(fVar, anEntry);
  end;
end;

begin {main}
 HideAll;
 GetFileName(fVar);
 SetUpDisplay(drawingRect);
{Preceding line expands the drawing window to full screen.}
 thereIsAnEntry := FALSE;
 which := 1;
 DrawScreen(anEntry, thereIsAnEntry, buttons);
 choice := NOTHING;
 repeat
  case choice of
   AFIELD :
    if thereIsAnEntry then
```

```
          begin
            ShowCell(anEntry[which], TRUE);
            repeat
            until GetNextEvent(mDownMask, event);
            GlobalToLocal(event.where);
            ShowCell(anEntry[which], FALSE);
          end;
      NOTHING :
        begin
          GetNextEntry(anEntry, fVar, thereIsAnEntry);
          DrawScreen(anEntry, thereIsAnEntry, buttons);
          repeat
          until GetNextEvent(mDownMask, event);
          GlobalToLocal(event.where);
        end
    end;
    Picked(choice, anEntry, which, thereIsAnEntry, event.where, buttons);
  until choice = stopIt;
  Close(fVar);
  SetRect(drawingRect, 293, 124, 508, 339);
  SetDrawingRect(drawingRect); {Restores drawing window.}
end.
```

Figure 6-4

You can run this program, if you want, but it won't do anything except display the contents of files created with the *Multiple Field Editor*.

Sorting

Sorting is the process of putting a set of items into some particular order. When you alphabetize a list, for example, you sort the list in alphabetical order.

This section adds a very simple sorting mechanism to *Database Framework*. The sorting added here allows you to sort the records based on any one field in the entries.

To add sorting to the framework:

1. Make sure that the *Database Framework* program is displayed in the programming window.
2. Choose Save As... from the File menu.

3. When you are asked for a new file name, type:

   ```
   Sort Program
   ```

   Click in the Save button.

4. Change the name of the program in the first line to:

   ```
   Sort_Program
   ```

5. Add the following to the list of button RECTs defined in the definition of butType in the **type** part of the program:

   ```
   sort,
   ```

   That line should now be:

   ```
   sort, stop : RECT;
   ```

   The order is unimportant.

6. Add the following to the definition of choiceType:

   ```
   SORTTHEM,
   ```

   That line should now be:

   ```
   choiceType = (STOPIT, AFIELD, SORTTHEM, NOTHING);
   ```

   The order is unimportant in this case.

7. Add the following line to procedure DrawScreen, after the call to DrawButton:

   ```
   DrawButton(buttons.sort, 280, 'Sort');
   ```

8. Add the following to procedure Picked immediately before the **end** of the procedure:

   ```
   else if ButtonPressed(buttons.sort, mouse) then
   choice := SORTTHEM;
   ```

   If you added a semicolon after the word "STOPIT" in the preceding line, remove it now.

9. Place an insertion point before **begin** {main} and type:

```
procedure Sort (var fVar : fileType;
which : INTEGER);
var
n, j : INTEGER;
numEntries : LONGINT;
firstEntry, secondEntry : entryType;
procedure Exchange (var fVar : fileType;
firstEntry, secondEntry : entryType);
begin
Seek(fVar, filePos(fVar) - 2);
Write(fVar, secondEntry, firstEntry);
end;
begin
See(fVar, MAXLONGINT);
numEntries := filePos(fVar) - 1;
for n := numEntries downto 1 do
for j := 0 to n - 1 do
begin
Seek(fVar, j);
Read(fVar, firstEntry);
Read(fVar, secondEntry);
if firstEntry[which].theString > secondEntry[which].theString
then
Exchange(fVar, firstEntry, secondEntry);
end;
Reset(fVar);
end;
```

10. Add the following to the **case** statement in the main program before AFIELD:

```
SORTTHEM:
begin
Sort(fVar, which);
GetNextEntry(anEntry, fVar, thereIsAnEntry);
DrawScreen(anEntry, thereIsAnEntry, buttons);
event.where := anEntry[which].wPos;
end;
```

11. Choose Save from the File menu.

The program should now be as shown in Figure 6-5.

```
program Sort_Program;
 const
  NUMRECS = 3;  {The number of separate records in an entry.}
  CR = 13;  {Character code for <RETURN>.}
  BS = 8;     {Character code for backspace.}
  LINEHEIGHT = 15;  {The distance between lines in pixels.}
  ENDSTR = 256;     {Guaranteed to be the past end of the string.}
  SPACE = 3;
  BUTHEIGHT = 20;
  BUTLEFT = 20;
  BUTWIDTH = 40;
 type
  stringRec = record
     wPos : POINT;
     theString : string;
     itsRect : RECT;
    end;
  butType = record
     sort, stop : RECT;
    end;
  choiceType = (STOPIT, AFIELD, SORTTHEM, NOTHING);
  entryType = array[1..NUMRECS] of stringRec;
  fileType = file of entryType;
 var
  anEntry : entryType;
  thereIsAnEntry : BOOLEAN;
  which, n : INTEGER;
  choice : choiceType;
  aRecord : stringRec;
  buttons : butType;
  drawingRect : RECT;
  event : EVENTRECORD;
  fVar : fileType;

 procedure GetFileName (var fVar : fileType);
  var
   fileName : string;
 begin
  repeat
   fileName := OldFileName('Choose a file.');
  until fileName <> '';
  Open(fVar, fileName);
 end;
```

```
procedure DrawButton (var butName : RECT;
          vertBase : INTEGER;
          theLabel : string);
begin
 SetRect(butName, BUTLEFT, vertBase, BUTLEFT + BUTWIDTH, vertBase + BUTHEIGHT);
 FillRoundRect(butName, 6, 6, white);
 FrameRoundRect(butName, 6, 6);
 MoveTo(BUTLEFT + SPACE, vertBase + 14);
 TextFont(0);
 WriteDraw(theLabel);
 TextFont(1);
end;

procedure ShowCell (aRecord : stringRec;
          highlight : BOOLEAN);
 var
  n, currentLine : INTEGER;
begin
 with aRecord do
  begin
   ClipRect(itsRect);
   if highlight then
    PenSize(3, 3);
   FillRect(itsRect, white);
   FrameRect(itsRect);
   PenNormal;
   if highlight then
    InsetRect(itsRect, 3, 3);
   ClipRect(itsRect);
   MoveTo(wPos.h, wPos.v);
   currentLine := 0;
   for n := 1 to Length(theString) do
    if theString[n] = Chr(CR) then
     begin
      currentLine := currentLine + 1;
      MoveTo(wPos.h, wPos.v + currentLine * LINEHEIGHT);
     end
    else
     DrawChar(theString[n]);  {Pre-defined.}
  end;
 if not highlight then
  ClipRect(drawingRect);
```

```
end;

procedure DrawScreen (anEntry : entryType;
           thereIsAnEntry : BOOLEAN;
           var buttons : butType);
 var
  n : INTEGER;
begin
 FillRect(drawingRect, gray);
 DrawButton(buttons.stop, 310, 'Stop');
 DrawButton(buttons.sort, 280, 'Sort');
 if thereIsAnEntry then
  for n := 1 to NUMRECS do
   ShowCell(anEntry[n], FALSE);
end;

procedure SetUpDisplay (var drawingRect : RECT);
begin
 SetRect(drawingRect, 0, 0, 532, 358);
 SetDrawingRect(drawingRect);
 ShowDrawing;
 FillRect(drawingRect, gray);
end;

procedure Picked (var choice : choiceType;
           anEntry : entryType;
           var which : INTEGER;
           thereIsAnEntry : BOOLEAN;
           mouse : POINT;
           buttons : butType);
 var
  n : INTEGER;

 function ButtonPressed (whichButton : RECT;
            mouse : POINT) : BOOLEAN;
 begin
  ButtonPressed := FALSE;
  if PtInRect(mouse, whichButton) then
   begin
    InvertRoundRect(whichButton, 6, 6);  {QuickDraw procedure.}
    ButtonPressed := TRUE;
   end;
 end;
```

```
begin {Picked}
 choice := NOTHING;
 if thereIsAnEntry then
  for n := 1 to NUMRECS do
   if PtInRect(mouse, anEntry[n].itsRect) then
    begin
     choice := AFIELD;
     which := n
    end;
 if ButtonPressed(buttons.stop, mouse) then
  choice := STOPIT
 else if ButtonPressed(buttons.sort, mouse) then
  choice := SORTTHEM;
end;

procedure GetNextEntry (var anEntry : entryType;
        var fVar : fileType;
        var thereIsAnEntry : BOOLEAN);
 var
  n : INTEGER;
begin
 if EOF(fVar) then
  Reset(fVar);
 if EOF(fVar) then
  thereIsAnEntry := FALSE
 else
  begin
   thereIsAnEntry := TRUE;
   Read(fVar, anEntry);
  end;
end;

procedure Sort (var fVar : fileType;
        which : INTEGER);
 var
  n, j : INTEGER;
  numEntries : LONGINT;
  firstEntry, secondEntry : entryType;
 procedure Exchange (var fVar : fileType;
         firstEntry, secondEntry : entryType);
 begin
  Seek(fVar, filePos(fVar) - 2);
```

```
    Write(fVar, secondEntry, firstEntry);
   end;
  begin
   Seek(fVar, MAXLONGINT);
   numEntries := filePos(fVar) - 1;
   for n := numEntries downto 1 do
    for j := 0 to n - 1 do
     begin
      Seek(fVar, j);
      Read(fVar, firstEntry);
      Read(fVar, secondEntry);
      if FirstEntry[which].theString > secondEntry[which].theString then
       Exchange(fVar, firstEntry, secondEntry);
     end;
   Reset(fVar);
  end;
begin {main}
 HideAll;
 GetFileName(fVar);
 SetUpDisplay(drawingRect);
{Preceding line expands the drawing window to full screen.}
 thereIsAnEntry := FALSE;
 which := 1;
 DrawScreen(anEntry, thereIsAnEntry, buttons);
 choice := NOTHING;
 repeat
  case choice of
   SORTTHEM :
    begin
     Sort(fVar, which);
     GetNextEntry(anEntry, fVar, thereIsAnEntry);
     DrawScreen(anEntry, thereIsAnEntry, buttons);
     event.where := anEntry[which].wPos;
    end;
   AFIELD :
    if thereIsAnEntry then
     begin
      ShowCell(anEntry[which], TRUE);
      repeat
      until GetNextEvent(mDownMask, event);
      GlobalToLocal(event.where);
      ShowCell(anEntry[which], FALSE);
     end;
```

```
      NOTHING :
        begin
          GetNextEntry(anEntry, fVar, thereIsAnEntry);
          DrawScreen(anEntry, thereIsAnEntry, buttons);
          repeat
          until GetNextEvent(mDownMask, event);
          GlobalToLocal(event.where);
        end
    end;
    Picked(choice, anEntry, which, thereIsAnEntry, event.where, buttons);
  until choice = stopIt;
  Close(fVar);
  SetRect(drawingRect, 293, 124, 508, 339);
  SetDrawingRect(drawingRect); {Restores drawing window.}
end.
```

Figure 6-5

Sort implements a bubble sort. A bubble sort is the simplest kind of sorting method. It has that name because components rise to the top like bubbles in a glass of sparkling water. You can use it to sort just about anything: the basic point is to compare two components, and exchange them if they are in the wrong order.

downto is an alternate keyword for use in a **for** statement. When **downTo** is used, the loop variable's value moves down instead of up.

Click in the Close box in the upper left corner of the programming window to hide the window.

Run this program. Note that you must choose a file that was created with the *Multiple Field Editor* when the *Sort* program asks you for a file name. If you don't have a file of the right type, choose any file, and when the buttons appear, click in the Stop button. When the program stops, close the *Sort* program, open *Multiple Field Editor* and run it. (Don't forget to click in the Close box of the programming window to hide that window.) Create some entries for the file. Then choose Stop, close *Multiple Field Editor*, open the *Sort* program, and run that program.

When you have a file displayed, click on one of the fields, and choose Sort.

Warning

The bubble sort method used in this program can be quite slow with more than a few records. When you run a sort, be patient.

Check to see that the records are now in sorted order. Click in another field, and click Sort again. The records are now sorted according to the field you just choose.

Do More

When doing the changes suggested in this section, if you have a 128K Mac, be careful to save your changes before trying to run your program. Otherwise you may lose your work.

1. There is a problem with the sort algorithm given in this chapter: the strings are sorted in character code order, and not in alphabetical order. In character code order, all lower case letters are greater than any upper case letters. For example, 'aaa' appears *after* 'ZZZ'. One way to get around that limitation is to convert everything to upper or lower case before sorting. You can convert a lower case letter to upper case by subtracting 32 from the Ord of the character. (That trick works because of the way the character codes are arranged.) Before doing that, you must check each character to see if it is a lower case letter. Change the Sort procedure to do that.
2. Write the Sort procedure described in Item 1 above, but do it so that the characters in the file are kept in their original state, although they are converted for the purpose of the sort.

3. You can use the *Multiple Field Editor* to store any kind of data. Use it to create an appointment calender that automatically tells you when you have an appointment. You can get the current date and time from the Macintosh with the GetTime procedure. GetTime returns a record of type DateTimeRec (a predefined data type). The definition of a DateTimeRec is:

```
record
   Year, Month, Day, Hour, Minute, Second,
   DayOfWeek:INTEGER
end;
```

To give the alarm, use the SysBeep procedure, which is discussed further in the next chapter. You give SysBeep an integer value which determines the length of the beep in .022 second increments. For example:

```
SysBeep(45)
```

produces a tone about a second long.

Hints: This program is fairly difficult. The simplest way is to have only one alarm active at a time. Write a routine that checks the current record for an entry marked with a certain character, such as an @. When an @ is found, the text following it should be of the form MM/DD HH:MM. (That is, month/day hour:minute.)

Programming this will be much easier if you restrict your date and time entries to a given format and mark it in this way. Write the routine that converts the text into conventional month, day, hour, and minute format as a separate procedure. Then, you can go back later and make it capable of handling less standard entries, such as one with the month written out (for example, November or December).

To convert text to integers, use the Ord function (which can find the character code of a character) and subtract 48 from the result. For example:

```
(Ord('3')-48)
```

produces the INTEGER value 3.

Convert the characters one at a time, and add the pairs together, taking care to multiply the character in the ten's position by ten. For example:

```
Day := (Ord(firstChar)-48)*10+(Ord(secondChar)-48)
```

You can put a call to the routine that checks the time in the **repeat/until** loop that waits for a mouse event. For example:

```
repeat
CheckAlarm(anEntry);
until GetNextEvent(mDownMask, event);
```

Once you have this working, modify it so it checks all entries in the file in turn. That way you can have several alarms active at a time. In the best situation, CheckAlarm checks a single entry, then returns to see if there is a mouse event, and then, the next time CheckAlarm is called, the next entry is checked. Otherwise, CheckAlarm would have control for long periods, preventing the program user from stopping the program.

QUICK SUMMARY

This chapter shows how to use disk files, so you can save information permanently. It includes a sorting module, to sort file records into character code order. The following routines and concepts were introduced.

Anonymous	file is a temporary file that is opened with no name. It ceases to exist when the program stops or when the file is closed.
Bubble sort	is a common method of sorting where components rise to their places in the list like bubbles in a glass of water. It is easy to implement and fairly simple, but takes a long time to sort large lists.

Character code order	is the sorting order defined by the character code used in the Macintosh. Every character has a numerical code assigned to it that is used internally to represent the character. The Ord function returns the character code of a character. This is not the same as alphabetical order because upper and lower case characters are sorted separately.
Close	is a predefined procedure that closes a file.
File variable	is a variable that is used to identify a file in Pascal. When a file is open, the file's file variable points to the current component of the file, or the the end of the file.
downTo	is a reserved word that can be used in a **for** statement in place of **to** so that **for** counts down instead of up.
EOF	is a predefined Boolean function that tells if the file variable points to the end of the file.
EOLN	is a predefined Boolean function that tells if the file variable points at an end-of-line character.
FilePos	is a predefined procedure whose return value is the component number of the component to which the file variable points.
Get	is a predefined procedure that gets the next component of the file. It does not actually read a component into a variable—it just moves the file variable so it points at the next component. You can then access the component by using fileVar±, where fileVar is name of the file variable. It is an error if the file was at the end of file before the Get call. See Read.
MAXINT	is a predefined constant that holds the largest INTEGER value, 32767.
NewFileName	is a predefined function that lets the user enter a new file name and choose Open or Cancel. The file name entered, if any, is the return value of this function. The file is not actually opened by this function, even if the user chooses Open.
Normal file	is a file with a name and a desktop document icon. See Anonymous file.
OldFileName	is a predefined function that displays the standard file dialog box that lets you choose from files on any disk and choose Open or Cancel. Only files of type TEXT (which include Macintosh Pascal files and files created by MacWrite and saved with the text-only option) are shown. The name of the file chosen is the return value of this function. The file is not actually opened by this function, even if the user chooses Open.

Open	is a predefined procedure that opens a file for reading and/or writing.
Put	is a predefined procedure that places a new component into a file and moves the file variable so it points at the next component (or at the end of the file, if there is no next component). If the file was at the end of file before the Put call the new component is appended to the file; otherwise the current component is replaced. See Write.
Read	is a predefined procedure that can read from a file. Read(fileVar, aVar) is equivalent to: aVar:=fileVar±; Get(fileVar).
ReadLn	is a predefined procedure like Read, except it can only be used with a file of type TEXT and it reads to the next end-of-line character or until the end of the file, whichever comes first.
Reset	is a predefined procedure that opens a file for reading only, if the file was closed. If the file was already open, Reset moves the file variable to the beginning of the file. If the file was already open when you called Reset and it was opened with Open, you can still read from or write to the file.
Rewrite	is a predefined procedure that erases the contents (if any) of a file and opens it for writing. If the file was already open, its contents are erased, but, if the file was opened with Open, you can still read from or write to the file.
Seek	is a predefined procedure that moves the file variable to a specified component of the file. You can only use this with files opened with Open.Sorting is the process of putting a list of items into a particular order.
TEXT	when used to define a file variable, defines a special type of file. In Macintosh Pascal, a TEXT file is the same as a file of CHAR, but, in other versions of Pascal a file of type TEXT is a unique type. There are certain file handling routines that can only be used with a TEXT file.
Write	is a predefined procedure that places a component into the file at the current file position. If the file variable points to the end of the file, the new components are appended to the file. If the file variable points elsewhere, the current component is replaced with the new component. Write(fileVar, aVar) is equivalent to: fileVar±:=aVar; Put(fileVar).
Writeln	is a predefined procedure that is like Write, except it can only be used with a TEXT file and it adds an end-of-line character after it completes writing its parameters into the file.

CHAPTER

7

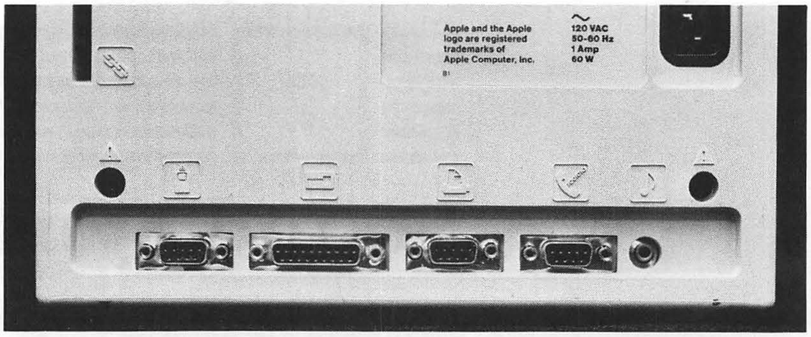
Sound Synthesis

One thing that separates the Mac from most computers is that it can make complicated sounds, with up to four different tones at a time. In fact, a Mac makes a pretty good music synthesizer, and can even be programmed to speak. (You can buy speech software for your Mac. Speech involves very complex sounds, which are outside the scope of this book.)

This chapter shows you how to get your Mac to make the sounds produced by a tone-dialing phone. Once you know how to produce those sounds, you can use similar methods to make whatever sounds you want. You are only limited by your imagination and your musical knowledge.

This chapter develops a module that can be combined with the editing program produced by the last three chapters. You will be able to use the editing program to create a telephone book, and use the new module to dial your phone.

You may want to add an extension speaker to your Mac. There is a socket on the back, near the On/off Switch on the bottom row, with an image of a musical note above it.


Apple and the Apple logo are registered trademarks of Apple Computer, Inc.
120 VAC
50-60 Hz
1 Amp
60 W

You can plug a speaker into that socket. It takes a miniature monaural plug of the size used for the tiny walk-about cassette/radio systems. You can get one at an electronics store. An external speaker will produce a louder, clearer sound than the speaker inside the Mac's case. You need to add a speaker if you want to dial your phone with the program in this chapter. The Mac's internal speaker is not loud enough to operate a telephone.

You also need to have a telephone line that can handle pulse-tone dialing. If you do not know if your line can, ask your local telephone company.

The next part of this chapter gives some theoretical background for understanding how the Mac's sound generator works. It is not necessary to understand all of what is in this section. It is here because you may, at some time, want to go back and figure out where the numbers used in this chapter come from. If you don't care to go too deeply into it, you can skip to the following section, which is about the practical side of sound generation.

A Little Theory

Sound is made up of waves moving through the air, in much the same way as waves move through water. You may have seen a graph of a wave before. Figure 7-1 shows one. A complete waveform like this is one **cycle** of the wave. A sound consists of a waveform like this one repeated many times a second.

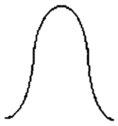


Figure 7-1 A Sine Wave

The pitch of the sound—how high or low it is—is determined by the numbers of cycles per second, which is the **frequency** of the waves. If there are more waves each second, the sound is higher pitched (Figure 7-2); if there are fewer waves each second, the sound is lower pitched (Figure 7-3).

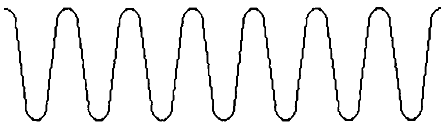


Figure 7-2 Higher-Pitched Wave

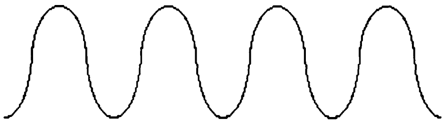


Figure 7-3 Lower-Pitched Wave

The quality of the sound is also changed by the shape of the wave. A wave can start at a low amplitude, rise slowly to its highest point and then fall slowly back to a low amplitude. These may be **sine waves**, which sound fairly smooth and flute-like (Figure 7-1). Other sounds may rise abruptly to full amplitude, and then fall just as abruptly to zero before the next cycle comes along. Those are called **square waves**, which sound rather abrupt and mechanical, like the sound of an organ (Figure 7-4). The shape of the wave determines the character of the sound.

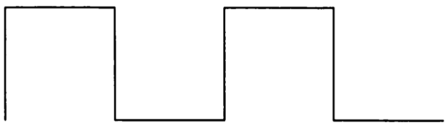


Figure 7-4 A Square Wave

Dialing tones consist of two separate frequencies at a time. When the switching computer in the telephone company's office detects the correct combination of tones, it interprets the tones as dialing commands. If the frequencies are not correct, or do not occur in the correct combinations, the switching computer ignores them.

The Mac has three built-in sound "synthesizers." The simplest is the **square-wave synthesizer**. The square-wave synthesizer always produces sounds with square waves—so that the sound is fairly mechanical and sharp. Only one square wave can be produced at a time—but you can change the frequency as often as you want.

You use the Mac's **four-tone synthesizer** to produce up to four different sounds at a time. Each of the four tones can have its own waveform and frequency. You generally define a single complete wave for each sound. The number of times the wave is repeated each second is the frequency of the sound.

Alternately, you can define a more complicated waveform. When you need that much control over the shape of the waveform, you use the **free-form synthesizer**. The free-form synthesizer can only produce one tone at a time, but the tone can change as much as you like.

With the free-form and four-tone synthesizers, you define a waveform by defining an array of values. The array contains 256 elements for the four-tone synthesizer, and 3000 elements for the free-form synthesizer. You then give a **rate** that determines how quickly the Mac goes through the waveform. Therefore, the frequency of the sound produced depends both on the rate and on the shape of the waveform.

With the free-form synthesizer, although the rate is fixed for the entire 3000 bytes of the wave, you can vary the shape of the wave so that the frequency changes. With the square-wave synthesizer, the shape of the wave is defined for you, but you can change the rate, and therefore the frequency, of the sound, as often as sixty times a second.

The sound generator uses a bit of memory left over at the end of each line of the screen. The sound is generated every time the Mac finishes scanning a video line, at intervals of 44.93 - μ5-secs. A -μ5-sec, pronounced micro-second, or mu-second, is a millionth of a second. That means that the Mac generates the sound over 20,000 times a second.

When you use the four-tone or free-form synthesizers, you define a waveform and a rate.

The scanning interval, 44.93 ·μ5-sec, relates the waveform, the rate, and the frequency. The rate is the number of values of the waveform that are played each time the Mac finishes displaying a video line.

The rate uses a data type you may not have seen before: a **fixed-point number**. A fixed-point number takes up four bytes in memory. The upper two bytes contain the integer part of the number. The lower two bytes contain the fractional part of the number. For example, if the number is 1234.5678, the upper two bytes contain 1234 while the lower two contain 5678.

There is a set of routines, described in Chapter 10 in the Macintosh Pascal manual, for manipulating fixed-point numbers. FixRatio, which is used in this chapter, returns the dividend of its two parameters, both of which must be INTEGERs.

When using the four-tone synthesizer, FixRatio can be used to divide the frequency by 87, to obtain the rate. The divisor 87 is derived from the fact that the wave contains 256 elements and the Mac scans the wave every 44.93 - μ5-secs. Every time the wave is scanned, the number of bytes of the wave indicated by the rate is loaded into the four-tone generator. (There can be a separate rate and waveform for each of the four tones. All tone generators are loaded simultaneously.)

The frequency is in cycles per second. There are one million ·μ5-secs in a second. The Mac therefore scans the wave 22256.84398 times a second. As there are 256 bytes in the wave, a number of bytes from the waveform equal to the rate is loaded into the synthesizer 86.9407968 times a second, which is close enough to 87 for our purposes. That means that, to produce a sound of a given frequency, you must give a rate which is the frequency divided by 87.

The whole number 87 is used in place of 86.9407968 because both factors in FixRatio must be INTEGER values.

The Practical Side

Notes

Before going on to the programs in this chapter, you need to make some extra room on your Pascal disk, if you haven't already done so. Move every program except for the last version of the *Multiple Field Editor* (the one with filing), the Sort program, and the *Database Framework* to another disk. (There should be plenty of room on the disk you used for information and programs removed from the release disk.)

There are three separate procedures that make sounds on the Mac:

- **StartSound**. You use StartSound if you want to use the square-wave, four-tone, or free-form synthesizers. You pass it different values depending on which synthesizer you use.
- **SysBeep**. You use the SysBeep procedure when you need the tone the Mac makes when it first turns on. With Sysbeep, you can only control the length of the sound. The waveform and frequency are fixed.
- **Note**. You use the Note procedure to produce a simple square-wave tone of a given frequency, volume, and duration.

A Simple Program

The following program is a short introduction to sound on the Mac. Get a new programming window and type:

```
program Notes;
const
BASE = 11;
var
event : EVENTRECORD;
key, quit : INTEGER;
begin
quit := (Ord('~') - 32);
repeat
repeat
until GetNextEvent(keyDownMask, event);
key := (event.message mod 256) - 32;
if key > 0 then
Note(key * BASE, 255, 5);
until key = quit;
end.
```

Choose Save As... from the File menu, and type "Notes" as the program name. Click in the Save button.

Run the program. Whenever you type a key, a note is played. To stop the program, type the ~ key, which is the key at the upper left corner of your keyboard. (You have to hold the Shift key down to get the ~ character.)

The predefined Note procedure is the heart of this program. The first parameter to Note is a **frequency**, in **cycles per second**. (A cycle per second is also called a **hertz**.) The second parameter is the **volume**, and can be a number from 0 to 255. The third parameter is a **duration**, in 60ths of a second. The Note procedure always produces square waves, and can only produce one frequency at a time. Try changing the duration. The tones produced are always multiples of the BASE value.

This program has a very simple method for converting the key values to notes: it uses a multiple of a value derived from the character code for the character produced by the key. A disadvantage of this method is that the keys on the keyboard are not in character code order, so the notes don't rise in a regular fashion as you move across the keyboard. You may want to rewrite this program, using a **case** statement to define the frequencies produced by each key, so that you can play the Mac's keyboard as if it were an instrument.

Save the program as *Notes,* and get a new programming window before going on to the next section.

Dialing a Telephone

The rest of this chapter shows how to use the four-tone synthesizer to dial a telephone.

Get a new programming window and type:

```
program Dialer;
var
dTones : FTSYNTHREC;
begin {main}
SetUpSound(dTones);
Dial(dTones, '555-1234');
Dispose(dTones.sndRec^. sound1Wave);
Dispose(dTones.sndRec);
end.
```

Choose Save As... from the file menu. Type "Dialer" as the program name, and click in the Save button.

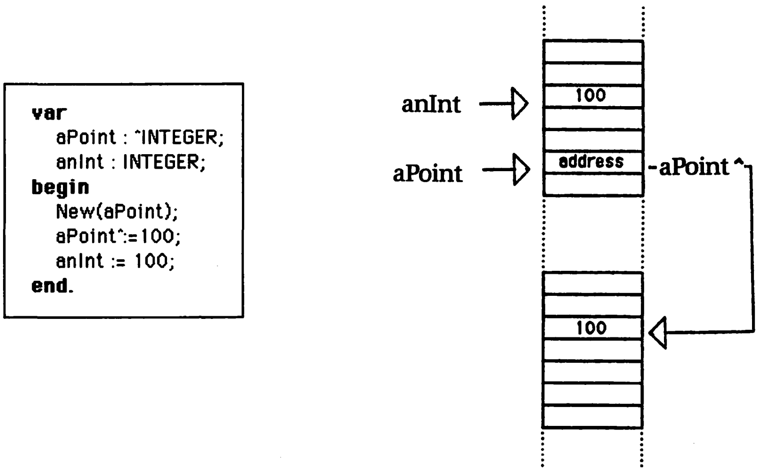
The number given in the call to Dial, which is a procedure defined within the next few pages, is the one that is dialed. You can change 555-1234 to any number you wish.

The variable *dTones* is the heart of the program. It is a FTSYNTHREC (short for four-tone synthesizer record), which is a special type defined just for the four-tone sythesizer. Two parts of the FTSYNTHREC are special variables called **pointers**.

To understand pointers, you have to know something about how a computer's memory works. Memory on any computer is numbered so that you can refer to any byte by its unique number. (A byte is the smallest part of memory you can directly refer to.) These numbers are called **addresses**.

When you define a variable, Pascal assigns, or **allocates,** some space in memory for the variable. When you use the variable's name, Pascal looks up the address of that variable's space in memory, and does whatever is required to the data at that address. This happens automatically, so you can consider a variable's name and its address to be the same thing for most purposes. The value of an ordinary variable is the value contained in the variable's address space.

A pointer is a special type of variable whose **value** is an address in memory. The pointer is said to **point to,** or **reference,** that address. You refer to the data contained at the location to which the pointer points by giving the caret symbol "^ ", after the name of the pointer. The difference is illustrated in Figure 7-5.



```
var
   aPoint : ^INTEGER;
   anInt : INTEGER;
begin
   New(aPoint);
   aPoint^:=100;
   anInt := 100;
end.
```

Figure 7-5

Pointers and Normal Variables

For example, suppose you have a pointer variable called *aPointer*. If you want to change the value at the part of memory to which *aPointer* points, you could use a statement like this:

```
aPointer^:= 509;
```

If you want to change the value of *aPointer* itself, so that it points to a different piece of memory, you would not use the caret (). For example, suppose you had another pointer called *otherPointer*. To make *aPointer* point at the same piece of memory as *otherPointer*, you would use this statement:

```
    aPointer := otherPointer;
```

Notice that no carets are used. On the other hand, if you want to copy the value that is in *otherPointer's* space to *aPointer's* space:

```
    aPointer^:= otherPointer^;
```

Every pointer is defined to point to a particular data type. For example:

```
    aPointer :^INTEGER;
```

defines *aPointer* to be a pointer to an INTEGER. You can define a pointer to any data type in the same way.

When you first begin your program, your pointer variables, like all variables, are undefined. To use a pointer, you must first give it an address to point to. If you want that to be a new piece of memory, as you often do, you call the predefined New procedure to allocate a piece of memory for the pointer's use. When you execute this statement:

```
    New(aPointer);
```

space is allocated for an INTEGER (assuming *aPointer* is an INTEGER pointer, as defined above), and the address of that space is placed in *aPointer*.

When you are finished with a pointer, you call the predefined Dispose procedure to free the space used by the pointer. If you don't dispose of the pointer, the pointer's space is locked up until yourestart Macintosh Pascal, and your program may eventually run out of space. The statement:

```
Dispose(aPointer);
```

frees or **deallocates** the space pointed to by *aPointer*. In the program, the statement:

```
Dispose(dTones.sndRec^.sound1Wave);
```

frees the space pointed to by the pointer *sound1Wave*. The pointer *sound1Wave* is itself stored in a part of memory reached by the pointer *sndRec*. The statement:

```
Dispose(dTones.sndRec);
```

frees the space pointed to by *sndRec*. You must free the space pointed to by *sound1Wave* before you free the space pointed to by *sndRec*. If you called Dispose for *sndRec* first, you wouldn't be able to dispose of the space pointed to by *sound1Wave,* because, if you destroy *sndRec* first, there is no way to reach *sound1Wave,* and no way to release its space.

Place an insertion point before **begin** {main} and type:

```
procedure SetUpSound (var dTones : FTSYNTHREC);
var
n : INTEGER;
begin
NEW(dTones.sndRec);
NEW(dTones.sndRec^.sound1Wave);
for n := 0 to 125 do
dTones.sndRec^.sound1Wave^[n] := 255;
for n := 126 to 255 do
dTones.sndRec^.sound1Wave^[n] := 0;
dTones.sndRec^.sound2Wave := dTones.sndRec^.sound1Wave;
dTones.sndRec^.sound1Phase := 0;
dTones.sndRec^.sound2Phase := 0;
dTones.mode := FTMODE;
end;
```

An FTSYNTHREC has two fields: *mode* and *sndRec*. The *mode* tells the Mac which synthesizer you are using. FTMODE is a predefined constant that indicates you want to use the four-tone synthesizer.

The statements:

```
for n := 0 to 125 do
dTones.sndRec^.sound1Wave^[n] := 255;
for n := 126 to 255 do
dTones.sndRec^.sound1Wave^[n] := 0;
```

place a square wave in the part of memory pointed to by *sound1Wave*. A four-tone synthesizer wave is an array of 256 values. These values are byte-sized values. A byte is the smallest part of memory you can directly refer to. An INTEGER takes up two bytes in memory. A single byte can take any whole-number value from 0 to 255. The value of each byte in the wave array determines the amplitude of the wave at that point. The beginning of this array has the maximum amplitude available , while the second half of the array has the minimum value available, resulting in a square wave. By placing different values in the wave's elements, you can shape the wave any way you like.

The pointer *sound2Wave* is assigned the value of *sound1Wave*.That means that *sound2Wave* points to the same part of memory as *sound1Wave*, and therefore *sound1Wave*^ and *sound2Wave*^ have the same value.

The **phase** values determine where in the wave the sound begins. You generally want a phase of 0 (zero), which means that sound is generated beginning at the beginning of the wave you've defined.

When you are using the four-tone synthesizer, the *sndRec* contains a wave, rate, and phase for each of the four available tones. This program uses only two of those tones, but you can use the other two tones in the same way.

The rate is defined in the MakeSound procedure, which is presented shortly.

First, place an insertion point before **begin** {main} and type:

```
procedure Dial (dTones : FTSYNTHREC;
number : string);
var
freq1, freq2, n ,oldLevel: INTEGER;
begin {Dial}
GetSoundVol(oldLevel); {Gets current volume setting.}
SetSoundVol(7); {Sets volume to highest.}
for n := 1 to Length(number) do
if number[n] in ['0' .. '9', '*','#']
then
begin
GetFreqs(number[n], freq1, freq2);
MakeSound(dTones, freq1, freq2);
end;
SetSoundVol(oldLevel); {Resets volume.}end;
```

This procedure creates the sound. The main part of it receives the number from the main program, and checks each character in it to see if it is a character that can be in a telephone number. The statement fragment:

```
if number[n] in ['0' .. '9', '*','#']
```

checks if the character in the number string is a telephone character. The operator **in** returns TRUE if the first operand (in this case, number[n]) is in the set that follows.

A set is any group of values of a single ordinal type. You can define a set by listing values inside square brackets explicitly, or with a range as is done here. (You can also define a set-type in the **type** declaration part of a program.) The range part of the set:

```
'0'..'9'
```

is equivalent to listing all the values in that range as part of the set.

Getting back to the **if** statement, if number[*n*] is not found **in** the set of values, in returns FALSE. The effect of that fragment is to check number[**n**], and see if it is a dialable character. If it is, the character is dialed. Otherwise, the **for** statement loops around, and the next character is checked, until there are no more characters in the number.

Finally, GetSoundVol finds the current setting of the volume control, while SetSoundVol changes that setting. The volume level is a value from zero to seven. The volume is set to seven before dialing because a telephone requires a rather loud tone to work. Changing the level with the SetSoundVol procedure has exactly the same effect as setting the volume with the volume control in the Mac's control panel. The old level is restored because it is impolite to change something in the control panel and not put it back the way it was.

The following procedure gets the correct frequencies for the given number. Place an insertion point before **begin** {dial} and type:

```
procedure GetFreqs (number : CHAR;
var freq1, freq2 : INTEGER);
begin
case number of
'1', '2', '3' :
freq1 := 697;
'4', '5', '6' :
freq1 := 770;
'7', '8', '9' :
freq1 := 852;
'*', '0', '#' :
freq1 := 941
end; {first case}
case number of
'1', '4', '7', '*' :
freq2 := 1209;
'2', '5', '8', '0' :
freq2 := 1336;
'3', '6', '9', '#' :
freq2 := 1477
end; {second case}
end;
```

Tone telephones produce two frequencies each time a key is pressed. The frequencies are shown in Figure 7-6. This procedure uses two **case** statements to find the frequencies for each key.

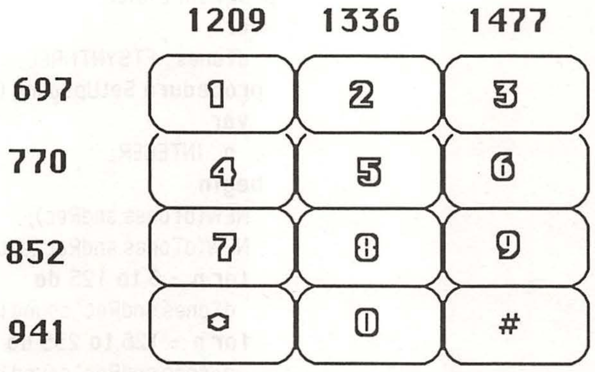


Figure 7-6 Matrix of Telephone Frequencies

The following procedure does the work of this program. Leave the insertion point where it is, before **begin** {dial}, and type:

```
Procedure MakeSound (dTones : FTSYNTHREC;
freq1, freq2 : INTEGER);
begin
with dTones.sndRec^ do begin
duration := 3; {length of sound}
sound1Rate := FixRatio(freq1, 87);
sound2Rate := FixRatio(freq2, 87);
end;
StartSound(@dTones, sizeOf(dTones), pointer(-1));
end;
```

Choose Save from the File menu.

The program should look like Figure 7-7.

```
program Dialer;
 var
  dTones : FTSYNTHREC;
 procedure SetUpSound (var dTones : FTSynthRec);
  var
   n : INTEGER;
 begin
  NEW(dTones.sndRec);
  NEW(dTones.sndRec^.sound1Wave);
  for n := 0 to 125 do
   dTones.sndRec^.sound1Wave^[n] := 255;
  for n := 126 to 255 do
   dTones.sndRec^.sound1Wave^[n] := 0;
  dTones.sndRec^.sound2Wave := dTones.sndRec^.sound1Wave;
  dTones.sndRec^.sound1Phase := 0;
  dTones.sndRec^.sound2Phase := 0;
  dTones.mode := FTmode;
 end;
 procedure Dial (dTones : FTSynthRec;
          number : string);
  var
   freq1, freq2, n, oldLevel : INTEGER;
  procedure GetFreqs (number : CHAR;
          var freq1, freq2 : INTEGER);
  begin
   case number of
    '1', '2', '3' :
     freq1 := 697;
    '4', '5', '6' :
     freq1 := 770;
    '7', '8', '9' :
     freq1 := 852;
    '*', '0', '#' :
     freq1 := 941
   end; {first case}
   case number of
    '1', '4', '7', '*' :
     freq2 := 1209;
    '2', '5', '8', '0' :
     freq2 := 1336;
    '3', '6', '9', '#' :
     freq2 := 1477
   end; {second case}
  end;
  procedure MakeSound (dTones : FTSynthRec;
          freq1, freq2 : INTEGER);
  begin
   with dTones.sndRec^ do
    begin
     duration := 3; {length of sound}
     sound1Rate := FixRatio(freq1, 87);
     sound2Rate := FixRatio(freq2, 87);
    end;
   StartSound(@dTones, sizeOf(dTones), pointer(-1));
  end;
 begin {Dial}
  GetSoundVol(oldLevel); {Gets current volume setting.}
  SetSoundVol(7); {Sets volume to highest.}
  for n := 1 to Length(number) do
   if number[n] in ['0'..'9', '*', '#'] then
    begin
     GetFreqs(number[n], freq1, freq2);
     MakeSound(dTones, freq1, freq2);
    end;
  SetSoundVol(oldLevel); {Resets volume.}
 end;
begin {main}
 SetUpSound(dTones);
 Dial(dTones, '555-1234');
 Dispose(dTones.sndRec^.sound1Wave);
 Dispose(dTones.sndRec);
end.
```

Figure 7-7

The duration is the length of the sound, in 60ths of a second. A duration of 3 should work, but if this program won't dial your phone when you use it later, you can try increasing this value to give longer tones.

The FixRatio function and the calculation of the rate are explained in the theory section in this chapter.

Look again at the call to StartSound:

StartSound(@dTones, SizeOf(dTones), pointer(-1));

StartSound requires the following three parameters, in this order:

1. A pointer to a synthesizer record.
2. The size, in bytes, of the synthesizer record.
3. A pointer to a procedure that is called after StartSound.

All of the procedures you've used so far go off and do their work, and then, when they are finished, return to your program.

StartSound starts the sound going, but doesn't have to wait for the sound to finish before returning to your program. There are three possibilities for what you give for this parameter. You can give the address of a procedure that you want called after StartSound starts the sound going. Doing so would, however, alter the flow of your program, and is not recommended. You can give the **nil** reserved word. **nil** is a special type of pointer value which points to nothing. When StartSound finds a value of **nil** as its third parameter, it returns to your program as soon as the sound begins. The third possibility is to give Pointer(-1), which causes StartSound to wait until the sound is finished before returning to your program. This is used in this program, and is explained below.

One problem is initially apparent: although the first parameter to StartSound must be a pointer to the synthesizer record, the variable *dTones*, which contains the synthesizer record, is not a pointer. The @ operator fixes that problem. The @ operator (pronounced “”at”) returns the address of the variable that follows it. The value of a pointer is an address, so giving @dTones is equivalent to giving a pointer to *dTones*. The value of the expression @dTones is the address of the storage space assigned to *dTones.*

The predefined SizeOf function returns the size of a variable, so SizeOf*(dTones)* produces the second required parameter.

Pointer(−1) is given as the third parameter. The predefined Pointer function converts a number to an address. When StartSound finds an address of −1, it doesn't return until it is finished generating the sound.Run the program. If you have an external speaker, a telephone near your Mac, and a telephone line that can handle tone dialing, you can use this program to dial your telephone. Put any number you like in place of 555-1234.

The procedures in this program can be combined with the *Database Framework* to produce the *Dialing* program. With this new program, you can dial numbers stored in a database produced by the *Multiple Field Editor.* Follow these steps:

1. Place an insertion point immediately before procedure SetUpSound.
2. Scroll the screen until you can see **begin** {main}.
3. Place the mouse pointer before **begin** {main}, hold down the Shift key, and press the mouse button. All three procedures in the program should now be selected.
4. Choose Copy from the Edit menu.
5. Open the Scrapbook by choosing Scrapbook from the Apple menu (the one on the leftmost edge of the menu bar).
6. Choose Paste from the Edit menu. The three procedures should now appear in the Scrapbook. (You will only be able to see the beginning of the first procedure.
7. Choose Close from the File menu to get rid of the *Dialer* program. If Macintosh Pascal asks you if you want to save the changes to your program, click in the Save button.
8. Choose Open... from the File menu.
9. Choose *Database* Framework from the file dialog box, and click in the Open button. (You probably will only be able to see part of the program's name:

   Database_Fra...)
10. Choose Save As... from the File menu. When you are asked for a new file name, type "Dialing Program", and click in the Save button.
11. Change the program name in the first line of the program to:

    Dialing_Program
12. Open the Scrapbook by choosing the Scrapbook command from the Apple menu. Unless you've added something to the Scrapbook, the Dial procedure should show in the Scrapbook's window. (If you have added something else to the Scrapbook, click in the Scrapbook's scroll bar until procedure SetUpSound is visible.) Choose Copy from the

File menu. (It is not necessary to explicitly select something from the Scrapbook to copy it. When the Scrapbook is displayed, the page that shows is always automatically selected.)

13. Click in the Close box in the upper left corner of the Scrapbook.
14. Place an insertion point in the program before **begin** {main}. Choose Paste from the Edit menu. The dialing procedures should now appear in your program.
15. In the type section of the program, insert the following into the list of button RECTs in butType:

    ```
    dial,
    ```

    That line should now be:

    ```
    dial, stop : RECT;
    ```

    The order is unimportant.
16. In the type section of the program, insert the following into the definition of choiceType:

    ```
    , DIALPHONE
    ```

    The definition of choiceType should now be:

    ```
    choiceType = (STOPIT, AFIELD, DIALPHONE, NOTHING);
    ```

    The order is unimportant in this case.
17. Add the following to the list of variables defined in the first **var** section in the program:

    ```
        dTones: FTSYNTHREC;
    ```

18. Add the following lines to procedure DrawScreen, immediately before the **if** statement:

    ```
    DrawButton(buttons.dial, 280, 'Dial');
    ```

19. Add the following to procedure Picked immediately before the **end** of the procedure:

    ```
    else if ButtonPressed(buttons.dial, event) then
    choice := DIALPHONE;
    ```

    (If you added a semicolon after the word STOPIT in that procedure, you must remove it.)

20. In the main program, add the following line after the call to SetUpDisplay:

```
SetUpSound(dTones);
```

21. In the main program, add the following to the case statement before AFIELD:

```
DIALPHONE:
begin
Dial(dTones, anEntry[which].theString);
DrawScreen(anEntry, thereIsAnEntry, buttons);
event.where := anEntry[which].wPos;
end;
```

22. Choose Save from the File menu.

The program should now look like Figure 7-8.

```
program Dialing_Program;
 const
  NUMRECS = 3;  {The number of separate records in an entry.}
  CR = 13;  {Character code for <RETURN>.}
  BS = 8;    {Character code for backspace.}
  LINEHEIGHT = 15;  {The distance between lines in pixels.}
  ENDSTR = 256;     {Guaranteed to be the past end of the string.}
  SPACE = 3;
  BUTHEIGHT = 20;
  BUTLEFT = 20;
  BUTWIDTH = 40;
 type
  stringRec = record
     wPos : POINT;
     theString : string;
     itsRect : RECT;
   end;
  butType = record
     dial, stop : RECT;
   end;
  choiceType = (STOPIT, AFIELD, DIALPHONE, NOTHING);
  entryType = array[1..NUMRECS] of stringRec;
  fileType = file of entryType;
 var
  dTones : FTSYNTHREC;
  anEntry : entryType;
  thereIsAnEntry : BOOLEAN;
  which, n : INTEGER;
  choice : choiceType;
  aRecord : stringRec;
  buttons : butType;
  drawingRect : RECT;
  event : EVENTRECORD;
  fVar : fileType;

procedure GetFileName (var fVar : fileType);
  var
    fileName : string;
begin
  repeat
    fileName := OldFileName('Choose a file.');
  until fileName <> '';
  Open(fVar, fileName);

end;

procedure DrawButton (var butName : RECT;
              vertBase : INTEGER;
              theLabel : string);
begin
  SetRect(butName, BUTLEFT, vertBase, BUTLEFT + BUTWIDTH, vertBase + BUTHEIGHT);
  FillRoundRect(butName, 6, 6, white);
  FrameRoundRect(butName, 6, 6);
  MoveTo(BUTLEFT + SPACE, vertBase + 14);
  TextFont(0);
  WriteDraw(theLabel);
  TextFont(1);
end;

procedure ShowCell (aRecord : stringRec;
              highlight : BOOLEAN);
  var
    n, currentLine : INTEGER;
begin
  with aRecord do
    begin
      ClipRect(itsRect);
      if highlight then
        PenSize(3, 3);
      FillRect(itsRect, white);
      FrameRect(itsRect);
      PenNormal;
      if highlight then
        InsetRect(itsRect, 3, 3);
      ClipRect(itsRect);
    MoveTo(wPos.h, wPos.v);
    currentLine := 0;
    for n := 1 to Length(theString) do
     if theString[n] = Chr(CR) then
      begin
       currentLine := currentLine + 1;
       MoveTo(wPos.h, wPos.v + currentLine * LINEHEIGHT);
      end
     else
      DrawChar(theString[n]);  {Pre-defined.}
  end;
 if not highlight then
   ClipRect(drawingRect);
 end;

procedure DrawScreen (anEntry : entryType;
          thereIsAnEntry : BOOLEAN;
          var buttons : butType);
 var
  n : INTEGER;
begin
 FillRect(drawingRect, gray);
 DrawButton(buttons.stop, 310, 'Stop');
 DrawButton(buttons.dial, 280, 'Dial');
 if thereIsAnEntry then
  for n := 1 to NUMRECS do
   ShowCell(anEntry[n], FALSE);
end;

procedure SetUpDisplay (var drawingRect : RECT);
begin
 SetRect(drawingRect, 0, 0, 532, 358);
 SetDrawingRect(drawingRect);
 ShowDrawing;
 FillRect(drawingRect, gray);
end;

procedure Picked (var choice : choiceType;
          anEntry : entryType;
          var which : INTEGER;
          thereIsAnEntry : BOOLEAN;
          mouse : POINT;
          buttons : butType);
 var
  n : INTEGER;
 function ButtonPressed (whichButton : RECT;
          mouse : POINT) : BOOLEAN;
 begin
  ButtonPressed := FALSE;
  if PtInRect(mouse, whichButton) then
   begin
    InvertRoundRect(whichButton, 6, 6);  {QuickDraw procedure.}
    ButtonPressed := TRUE;
   end;
 end;

begin {Picked}
 choice := NOTHING;
 if thereIsAnEntry then
  for n := 1 to NUMRECS do
   if PtInRect(mouse, anEntry[n].itsRect) then
    begin
     choice := AFIELD;
     which := n
    end;
 if ButtonPressed(buttons.stop, mouse) then
  choice := STOPIT
 else if ButtonPressed(buttons.dial, mouse) then
  choice := DIALPHONE;
end;

procedure GetNextEntry (var anEntry : entryType;
         var fVar : fileType;
         var thereIsAnEntry : BOOLEAN);
 var
  n : INTEGER;
begin
 if EOF(fVar) then
  Reset(fVar);
 if EOF(fVar) then
  thereIsAnEntry := FALSE
 else
  begin
   thereIsAnEntry := TRUE;
   Read(fVar, anEntry);
  end;
end;

procedure SetUpSound (var dTones : FTSynthRec);
 var
  n : INTEGER;
```

```
begin
 NEW(dTones.sndRec);
 NEW(dTones.sndRec^.sound1Wave);
 for n := 0 to 125 do
  dTones.sndRec^.sound1Wave^[n] := 255;
 for n := 126 to 255 do
  dTones.sndRec^.sound1Wave^[n] := 0;
 dTones.sndRec^.sound2Wave := dTones.sndRec^.sound1Wave;
 dTones.sndRec^.sound1Phase := 0;
 dTones.sndRec^.sound2Phase := 0;
 dTones.mode := FTmode;
end;

procedure Dial (dTones : FTSynthRec;
          number : string);
 var
  freq1, freq2, n, oldLevel : INTEGER;

 procedure GetFreqs (number : CHAR;
           var freq1, freq2 : INTEGER);
 begin
  case number of
   '1', '2', '3' :
    freq1 := 697;
   '4', '5', '6' :
    freq1 := 770;
   '7', '8', '9' :
    freq1 := 852;
   '*', '0', '#' :
    freq1 := 941
  end; {first case}
  case number of
   '1', '4', '7', '*' :
    freq2 := 1209;
   '2', '5', '8', '0' :
    freq2 := 1336;
   '3', '6', '9', '#' :
    freq2 := 1477
  end; {second case}
 end;

 procedure MakeSound (dTones : FTSynthRec;
           freq1, freq2 : INTEGER);
 begin
  with dTones.sndRec^ do
```

```
begin
 duration := 3; {length of sound}
 sound1Rate := FixRatio(freq1, 87);
 sound2Rate := FixRatio(freq2, 87);
    end;
   StartSound(@dTones, sizeOf(dTones), pointer(-1));
  end;

 begin {Dial}
  GetSoundVol(oldLevel); {Gets current volume setting.}
  SetSoundVol(7); {Sets volume to highest.}
  for n := 1 to Length(number) do
   if number[n] in ['0'..'9', '*', '#'] then
    begin
     GetFreqs(number[n], freq1, freq2);
     MakeSound(dTones, freq1, freq2);
    end;
  SetSoundVol(oldLevel); {Resets volume.}
 end;

begin {main}
 HideAll;
 GetFileName(fVar);
 SetUpDisplay(drawingRect);
{Preceding line expands the drawing window to full screen.}
 SetUpSound(dTones);
 thereIsAnEntry := FALSE;
 which := 1;
 DrawScreen(anEntry, thereIsAnEntry, buttons);
 choice := NOTHING;
 repeat
  case choice of
   DIALPHONE :
    begin
     Dial(dTones, anEntry[which].theString);
     DrawScreen(anEntry, thereIsAnEntry, buttons);
     event.where := anEntry[which].wPos;
    end;
   AFIELD :
    if thereIsAnEntry then
     begin
      ShowCell(anEntry[which], TRUE);
      repeat
      until GetNextEvent(mDownMask, event);
      GlobalToLocal(event.where);
      ShowCell(anEntry[which], FALSE);
      end;
      NOTHING :
        begin
          GetNextEntry(anEntry, fVar, thereIsAnEntry);
          DrawScreen(anEntry, thereIsAnEntry, buttons);
          repeat
          until GetNextEvent(mDownMask, event);
          GlobalToLocal(event.where);
        end
     end;
     Picked(choice, anEntry, which, thereIsAnEntry, event.where, buttons);
    until choice = stopIt;
    Close(fVar);
    SetRect(drawingRect, 293, 124, 508, 339);
    SetDrawingRect(drawingRect); {Restores drawing window.}
  end.
```

Figure 7-8

If you've already used the *Multiple Field Editor* to save some telephone numbers in a file, you can skip the next group of steps. Otherwise:

1. Choose Close from the File menu.
2. Choose Open... from the File menu.
3. Select *Multiple Field Editor*, and click in the Open button.
4. Click in the Close box for the programming window to hide it.
5. Run the program.
6. Click in the Cancel button when you are asked to choose an existing file.
7. Type in "Telephone Book" as the new file name.
8. Draw some boxes you like, and type in a few records with telephone numbers. One restriction: all fields with telephone numbers should contain no other numbers. You can, though, have identifying words using characters that are not part of the set of dialing characters, such as:

   home: 555-1234

   or

   work: 555-1235

The important point is that all telephone characters (1, 2, 3, 4, 5, 6, 7, 8, 9, 0, #, *) in a single window should belong to the number. The dialing program dials all the numbers in a window at a time, so you can't have more than one telephone number in a single window.

When you are done entering numbers, click in the Stop button.

9. Choose Close from the File menu.

10. Choose Open from the File menu.

11. Select the *Dialing Program*, and click in the Open button.

You should now have the Dialing Program displayed, and you should have a data file with telephone numbers stored on your disk. Click in the Close box in the upper left corner of the programming window to hide the window. Run the program.

It asks you for a file name. If, by some chance, you don't have a file of the right type, you must choose any file, and then choose the Stop button. This program should not damage any files, as it never changes any data, but it will not work properly unless you give it a file created with a version of *Multiple Field Editor* that has the same number of fields.

You can now click in any field with a telephone number, in order to select it, and then click in the Dial button. The program produces tones for any numbers (or # or *) that appear in the selected field.

You cannot edit the records without running Multiple Field Editor. If you have a 512K Mac, you may want to add dialing capabilities to the full Multiple Field Editor program, rather than stripping out editing and sorting capabilities first.

Do More

1. Some long distance services require you to dial a phone number, pause until the service answers, dial a set of numbers, pause, and then dial the phone number you want to reach. Modify the program to be able to handle that. Do it by defining a new character which specifies a pause. For example, if "#" is the pause sign, the following might be such a sequence:

   555-1234–09876543–555-5678

   You can use the TickCount function to implement a pause. TickCount returns a LONGINT value giving the elapsed time since the system was last started, in 60ths of a second. You can have your pause procedure check the value of TickCount, and keep on checking the value until the value increases by a given amount. (Hint: Store the initial value of TickCount, and use a **repeat/until** statement.) You may need to experiment to decide how long the pause must be. Alternately, you can use the GetTime procedure to find the current time, and keep checking the time value until it increases by a given amount.

2. You can't detect the telephone's response to your call without special equipment, but suppose you had a way of detecting if the phone rings or if there is a busy signal. If you could do that, you might want the program to retry a busy call until the phone rings. Rewrite the program so that a mouse click in a certain interval after dialing indicates a busy signal, and have your program redial. Don't forget to have the program hang up first. (The Mac can't actually hang up, of course, so have it tell you to hang up.)

QUICK SUMMARY

This chapter shows how to use the sound synthesizers on the Macintosh. You first created a program that demonstrated some sounds, and then wrote a module that produced two simultaneous tones that could be used to dial a phone. You used the following new statements, routines, and concepts.

Address	is a location in a computer's memory.
Cycle	of a wave is one complete wave.
Dispose	is a predefined procedure that frees space pointed to by a pointer. This is the inverse of the New procedure.
Fixed-point number	in mathematics is a number where the decimal point is in a fixed position. In Macintosh Pascal, a fixed-point number is one where the whole number part and the fractional part are stored separately. You must use a special set of routines to do arithmetic with these numbers.
Four-tone synthesizer	is one of the three sound synthesizers available on the Macintosh. You can use this synthesizer to produce up to four tones at a time. (This synthesizer is used to produce the two simultaneous sounds needed for dialing a phone.)
Free-form synthesizer	is one of the three sound synthesizers available on the Macinotosh. You can use this to produce a complicated sound, such as speech. The free-form synthesizer can only produce one sound at a time.
Frequency	of a wave is the number of cycles per second, usually expressed in hertz or cycles per second.
GetSoundVol	is a predefined procedure that returns the current sound volume setting.
Hertz	is the number of cycles per second.
Microsecond	is a millionth of a second, orμb5-second.
New	is a predefined procedure that allocates space for a variable and returns the address of that variable in the given pointer.
nil	is a pointer-type value which points at nothing. It is used because you can test for it, while you cannot otherwise test a pointer that points at nothing.

Note	is a predefined procedure that produces a square-wave tone of a given frequency, volume, and duration.
Pitch	of a sound is how high or low the sound is.
Pointer	is a type of variable whose value is an address in memory. A pointer points to an identified variable, which can be a variable of any data type. You access values in the pointer's identified variable by giving a caret(^) after the name of the pointer.
Pointer	is a predefined function that converts an INTEGER or LONGINT value into a pointer-type value.
Rate	is the value given to the four-tone and free-form synthesizers that determines how quickly the Macintosh moves through the waveform you've defined. The shape of the waveform and the rate together determine the frequency of the wave, which determines the pitch.
SetSoundVol	is a predefined procedure which sets the volume of sound proceded by the Macintosh.
Sine wave	is a common type of wave, which sounds fairly smooth and flute-like.
Square wave	is a common type of wave, which sounds relatively abrupt and mechanical, like an organ.
Square wave synthesizer	is one of the three sound synthesizers built into the Macintosh. It is easy to use, but always produces square waves, and only produces one tone at a time.
StartSound	is a predefined procedure that starts sound when you use the square-wave, four-tone, or free-form synthesizers.
SysBeep	is a predefined procedure that produces the tone the Macintosh makes when it first turns on.
Waveform	is a set of values that determine the form of a wave. The waveform and the rate determine how the sound produced.

CHAPTER 8

Printing

Computers are wonderful for manipulating information, but when it comes to carrying something around, paper works much better. You can have as much information as you like typed into a computerized database, but if you can't print it out when you need to, it isn't going to be much help.

When you follow these steps, you will have a program that allows you to look at a data file and decide whether to print it. You can start printing at any entry, and all the entries are printed. A line is skipped between each entry.

1. Open the file *Database Framework.*
2. Choose Save As... from the File menu. Type "Printing Program" as the new file name, and click in the Save button.
3. Change the name of the program on the first line to:

   Printing_Program.

4. In the **type** section of the program, add the following to the list of RECTs in the definition of butType:

   ```
   print,
   ```

   That line should now be:

   ```
   print, stop: RECT;
   ```

   The order is unimportant.

5. Add the following to the definition of choiceType:

   ```
   PRINTFILE,
   ```

   The definition of choiceType should now be:

   ```
   choiceType = (STOPIT, AFIELD, PRINTFILE, NOTHING);
   ```

   The order is unimportant in this case.

6. In procedure DrawScreen, place an insertion point before the **if** and type:

   ```
   DrawButton(buttons.print, 280,'Print');
   ```

7. In procedure Picked, place an insertion point before the **end** and type:

   ```
   else if ButtonPressed(buttons.print, mouse) thenchoice :=
   PRINTFILE;
   ```

   If you put a semicolon after STOPIT, remove it.

8. Place an insertion point before **begin** {main} and type:

   ```
   procedure Print (var fVar : fileType;
   var anEntry : entryType);
   var
   printer : TEXT;
   stopHere : LONGINT;
   n : INTEGER;
   thereIsAnEntry : BOOLEAN;
   procedure PrintString (var printer : TEXT; theString : string);
   var
   toPrint : string;
   n : INTEGER;
   begin
   toPrint := ''; {Two single quote marks.}
   for n := 1 to length(theString) do
   if theString[n] <> chr(CR) then
   Insert(theString[n], toPrint, ENDSTR)
   else
   ```

```
begin
WriteLn(printer, toPrint);
toPrint := ''; {Two single quote marks.}
end;
if toPrint <> '' then
WriteLn(printer, toPrint);
end;
begin {Print}
ReWrite(printer, 'PRINTER:');
stopHere := filePos(fVar);
repeat
for n := 1 to NUMRECS do
PrintString(printer, anEntry[n].theString);
WriteLn(printer, ''); {To get blank line between entries.}
GetNextEntry(anEntry, fVar, thereIsAnEntry);
until (filePos(fVar) = stopHere) or (not thereIsAnEntry);
WriteLn(printer, chr(12));
Close(printer);
end;
```

In this procedure, the line:

```
WriteLn(printer, chr(12));
```

sends a **control code** to the printer. The character represented by the character code 12 is a non-printing character, that tells the printer to perform a form feed. The manual that comes with your printer details the control codes available. You can do all sorts of fancy printing using control codes, including printing any graphics that can be drawn on the Macintosh screen.

9. In the **case** statement in the main program, place an insertion point before the word "AFIELD" and type:

```
PRINTFILE :
begin
Print(fVar, anEntry);
DrawScreen(anEntry, thereIsAnEntry,buttons);
event.where := anEntry[which].wpos;
end;
```

10. Choose Save from the File menu.

The program should like Figure 8-1.

Using this program, you should be able to print any file created with the *Multiple Field Editor*. It starts printing with the record that is currently displayed on the screen.As with the *Dialing* program, if you change the number of records in an entry (NUMRECS) in the *Multiple Field Editor*, you should change the value of NUMRECS in the printing program before trying to open files that use that program.

This program will not allow you to stop without first picking a file. If you realize after you run the program that you don't have a file of the right type, choose any file, and then immediately choose Stop.

```
program Printing_Program;
 const
  NUMRECS = 3;  {The number of separate records in an entry.}
  CR = 13;  {Character code for <RETURN>.}
  BS = 8;    {Character code for backspace.}
  LINEHEIGHT = 15;  {The distance between lines in pixels.}
  ENDSTR = 256;     {Guaranteed to be the past end of the string.}
  SPACE = 3;
  BUTHEIGHT = 20;
  BUTLEFT = 20;
  BUTWIDTH = 40;
 type
  stringRec = record
     wPos : POINT;
     theString : string;
     itsRect : RECT;
   end;
  butType = record
     print, stop : RECT;
   end;
  choiceType = (STOPIT, AFIELD, PRINTFILE, NOTHING);
  entryType = array[1..NUMRECS] of stringRec;
  fileType = file of entryType;
```

```
var
 anEntry : entryType;
 thereIsAnEntry : BOOLEAN;
 which, n : INTEGER;
 choice : choiceType;
 aRecord : stringRec;
 buttons : butType;
 drawingRect : RECT;
 event : EVENTRECORD;
 fVar : fileType;

procedure GetFileName (var fVar : fileType);
 var
  fileName : string;
begin
 repeat
  fileName := OldFileName('Choose a file.');
 until fileName <> '';
 Open(fVar, fileName);
end;

procedure DrawButton (var butName : RECT;
          vertBase : INTEGER;
          theLabel : string);
begin
 SetRect(butName, BUTLEFT, vertBase, BUTLEFT + BUTWIDTH, vertBase + BUTHEIGHT);
 FillRoundRect(butName, 6, 6, white);
 FrameRoundRect(butName, 6, 6);
 MoveTo(BUTLEFT + SPACE, vertBase + 14);
 TextFont(0);
 WriteDraw(theLabel);
 TextFont(1);
end;

procedure ShowCell (aRecord : stringRec;
          highlight : BOOLEAN);
 var
  n, currentLine : INTEGER;
begin
 with aRecord do
  begin
   ClipRect(itsRect);
   if highlight then
    PenSize(3, 3);
   FillRect(itsRect, white);
  FrameRect(itsRect);
  PenNormal;
  if highlight then
   InsetRect(itsRect, 3, 3);
  ClipRect(itsRect);
  MoveTo(wPos.h, wPos.v);
  currentLine := 0;
  for n := 1 to Length(theString) do
   if theString[n] = Chr(CR) then
    begin
     currentLine := currentLine + 1;
     MoveTo(wPos.h, wPos.v + currentLine * LINEHEIGHT);
    end
   else
    DrawChar(theString[n]);  {Pre-defined.}
 end;
if not highlight then
 ClipRect(drawingRect);
end;

procedure DrawScreen (anEntry : entryType;
         thereIsAnEntry : BOOLEAN;
         var buttons : butType);
 var
  n : INTEGER;
begin
 FillRect(drawingRect, gray);
 DrawButton(buttons.stop, 310, 'Stop');
 DrawButton(buttons.print, 280, 'Print');
 if thereIsAnEntry then
  for n := 1 to NUMRECS do
   ShowCell(anEntry[n], FALSE);
end;

procedure SetUpDisplay (var drawingRect : RECT);
begin
 SetRect(drawingRect, 0, 0, 532, 358);
 SetDrawingRect(drawingRect);
 ShowDrawing;
 FillRect(drawingRect, gray);
end;
```

```
procedure Picked (var choice : choiceType;
        anEntry : entryType;
        var which : INTEGER;
        thereIsAnEntry : BOOLEAN;
        mouse : POINT;
        buttons : butType);
 var
  n : INTEGER;

 function ButtonPressed (whichButton : RECT;
         mouse : POINT) : BOOLEAN;
 begin
  ButtonPressed := FALSE;
  if PtInRect(mouse, whichButton) then
   begin
    InvertRoundRect(whichButton, 6, 6);  {QuickDraw procedure.}
    ButtonPressed := TRUE;
   end;
 end;

begin {Picked}
 choice := NOTHING;
 if thereIsAnEntry then
  for n := 1 to NUMRECS do
   if PtInRect(mouse, anEntry[n].itsRect) then
    begin
     choice := AFIELD;
     which := n
    end;
 if ButtonPressed(buttons.stop, mouse) then
  choice := STOPIT
 else if ButtonPressed(buttons.print, mouse) then
  choice := PRINTFILE;
end;

procedure GetNextEntry (var anEntry : entryType;
        var fVar : fileType;
        var thereIsAnEntry : BOOLEAN);
 var
  n : INTEGER;
begin
 if EOF(fVar) then
  Reset(fVar);
 if EOF(fVar) then
  thereIsAnEntry := FALSE
 else
```

```
    begin
     thereIsAnEntry := TRUE;
     Read(fVar, anEntry);
    end;
end;

procedure Print (var fVar : fileType;
          var anEntry : entryType);
 var
   printer : TEXT;
   stopHere : LONGINT;
   n : INTEGER;
   thereIsAnEntry : BOOLEAN;

 procedure PrintString (var printer : TEXT;
           theString : string);
  var
    toPrint : string;
    n : INTEGER;
 begin
  toPrint := ''; {Two single quote marks.}
  for n := 1 to length(theString) do
    if theString[n] <> chr(CR) then
      Insert(theString[n], toPrint, ENDSTR)
    else
     begin
       WriteLn(printer, toPrint);
       toPrint := ''; {Two single quote marks.}
     end;
  if toPrint <> '' then
    WriteLn(printer, toPrint);
 end;

begin {Print}
 ReWrite(printer, 'PRINTER:');
 stopHere := filePos(fVar);
 repeat
   for n := 1 to NUMRECS do
     PrintString(printer, anEntry[n].theString);
   WriteLn(printer, ''); {To get blank line between entries.}
   GetNextEntry(anEntry, fVar, thereIsAnEntry);
 until (filePos(fVar) = stopHere) or (not thereIsAnEntry);
 WriteLn(printer, chr(12));
 Close(printer);
end;
```

```
    begin {main}
     HideAll;
     GetFileName(fVar);
     SetUpDisplay(drawingRect);
    {Preceding line expands the drawing window to full screen.}
     thereIsAnEntry := FALSE;
     which := 1;
     DrawScreen(anEntry, thereIsAnEntry, buttons);
     choice := NOTHING;
     repeat
      case choice of
       PRINTFILE :
        begin
      Print(fVar, anEntry);
      DrawScreen(anEntry, thereIsAnEntry, buttons);
      event.where := anEntry[which].wpos;
     end;
    AFIELD :
     if thereIsAnEntry then
      begin
       ShowCell(anEntry[which], TRUE);
       repeat
       until GetNextEvent(mDownMask, event);
       GlobalToLocal(event.where);
       ShowCell(anEntry[which], FALSE);
      end;
    NOTHING :
     begin
      GetNextEntry(anEntry, fVar, thereIsAnEntry);
      DrawScreen(anEntry, thereIsAnEntry, buttons);
      repeat
      until GetNextEvent(mDownMask, event);
      GlobalToLocal(event.where);
     end
   end;
   Picked(choice, anEntry, which, thereIsAnEntry, event.where, buttons);
  until choice = stopIt;
  Close(fVar);
  SetRect(drawingRect, 293, 124, 508, 339);
  SetDrawingRect(drawingRect); {Restores drawing window.}
 end.
```

CHAPTER

9

Advanced QuickDraw

Advanced QuickDrawThe QuickDraw graphics package is a large part of what makes the Mac the visually exciting and interactive computer that it is. QuickDraw draws simple and complex shapes, handles text, and also lets you find out information about points and shapes—such as whether a point is inside or outside some area.

Macintosh Pascal divides QuickDraw into two parts, QuickDraw1 and QuickDraw2. All of the QuickDraw routines used so far in this book have been in QuickDraw1. The routines in QuickDraw1 are, with a few exceptions, straightforward. QuickDraw2 contains routines that are also basically simple and straightforward, although they are occasionally sufficiently clever that it is a bit hard to understand how to use them. This chapter shows how to use the main routines in the five basic areas in QuickDraw2. These areas are:

- **Polygons**, which are closed figures with as many sides as you want.

- **Pictures**, which are sequences of QuickDraw drawing calls saved and played back later.
- **Regions**, which are closed figures that can be of any shape. Regions can consist of several separate, unconnected areas, all of which together make a single region.
- **Cursor**, which is the figure that appears on the screen to show the position of the mouse. You can define your own cursor shapes and use them in your programs.
- **Bitmaps**, which are special variables that represent groups of pixels. A bitmap may be displayed on the screen, or it may be stored in memory. You can copy sections of the screen into a bitmap, copy bitmaps to the screen, and change the shape or contents of bitmaps.

This chapter has a section on each of these areas, each with a sample program showing you how to use that part of QuickDraw.

All of QuickDraw, including some parts that are not included in this book, is described in an appendix of the Macintosh Pascal manual. At the end of that appendix is a copy of the code that defines the QuickDraw routines. That code is called the QuickDraw **interface**. There is a number 1 or 2 next to each routine in the left column of the interface. A "1" indicates that the routine is contained in QuickDraw1 and "2" indicates that it is in QuickDraw 2. You don't have to do anything special to use the QuickDraw1 routines. To use the QuickDraw2 routines, insert the following line in your program immediately after the **program** statement:

```
uses QuickDraw2;
```

This statement orders Macintosh Pascal to load QuickDraw2 into memory when it runs your program. Unfortunately, QuickDraw2 takes up a considerable amount of space, so do not use it unless you really need it.

Polygons

A polygon is any multi-sided shape. In QuickDraw, you can use any group of line drawing statements to define a polygon. Polygons are always closed shapes, though, which means that the lines that make up a polygon must come together, so that there is no part of the polygon that doesn't enclose something. Figure 9-1 shows some possible QuickDraw polygons. Notice that they can be complicated shapes. The lines that make up the polygon can cross, however, no lines that make up a polygon can just end in space. In fact, if you try to create a polygon that is not a closed space, QuickDraw closes up the polygon for you, which may result in a shape different from what you intended. Notice that QuickDraw polygons are not exactly the same as polygons you can draw in MacPaint and MacDraw. Those polygons can have ""sides" that fail to enclose space.

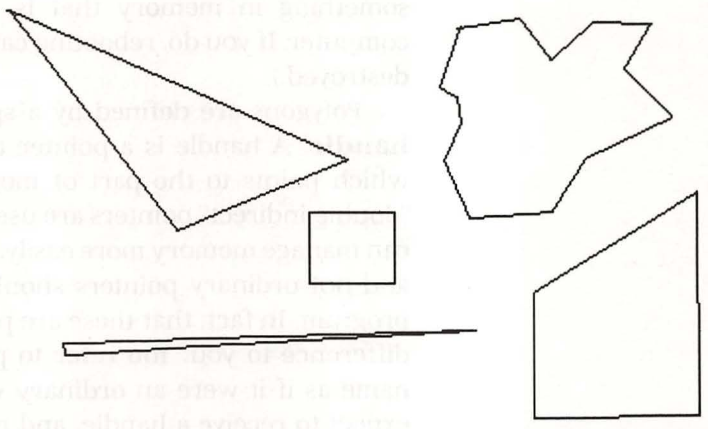


Figure 9-1 Sample Polygons

Once you have a polygon, you can do most of the things with it that you can do with a rectangle. You can frame the polygon or fill it with different colors, you can invert the polygon, and you can change the size of the polygon.

The main difference between a polygon and a rectangle is that a polygon can have any number of sides. For that reason, QuickDraw must be able to store the definition of the polygon in a different way from the way it stores the definition of a rectangle. Ordinary variables, such as RECTs, are of a fixed size. Once you define the type of a variable, the amount of memory that that variable can occupy is fixed, and, therefore, the amount of information that that variable can hold is fixed. With a polygon, though, you may not know how many sides the polygon will have, and you cannot, therefore, know how much space will be needed to store the information that defines the polygon.

The way around that limitation is to use pointers to memory. A pointer gets around Pascal's otherwise strict requirement that you define the size of variables before your program starts running. Because a pointer points directly at a place in memory, Pascal does not necessarily know how big a chunk of memory you are using. (That freedom can create lots of problems if you are not *very* careful, because you can accidentally destroy something in memory that is vital to the operation of the computer. If you do, rebooting can usually restore whatever was destroyed.)

Polygons are defined by a special kind of pointer called a **handle**. A handle is a pointer that points to another pointer, which points to the part of memory holding the data. These "double-indirect" pointers are used so that the Macintosh system can manage memory more easily. The fact that these are handles and not ordinary pointers should make no difference to your program. In fact, that these are pointers at all should make little difference to you. You refer to polygons by using the handle's name as if it were an ordinary variable. The polygon routines expect to receive a handle, and not direct data.

The other sections of this chapter also use handles.

When you define a polygon, you call a function to create the handle for your polygon, then you call line drawing routines until you have defined the entire polygon, and then you call a procedure that tells QuickDraw you have defined the entire polygon.

Here is a procedure that defines a polygon. You always define polygons like this. If you want to define your own polygon, just place your own line drawing calls between the OpenPoly and ClosePoly calls.

Get a new programming screen and type:

```
procedure CreateShape (var polygon : polyHandle);
begin
polygon := OpenPoly;
DrawLine(0, 60, 160, 60);
LineTo(13, 160);
LineTo(80, 0);
LineTo(133, 160);
LineTo(0, 60);
ClosePoly;
end;
```

You can use LineTo, Line, or DrawLine to define polygon sides. Rectangle, arc, or oval drawing routines have no effect on your polygon definition.

The call to OpenPoly hides the pen, so that the drawing does not show on the screen until the ClosePoly call is executed. If you want the drawing to show while you are defining the polygon, you can call ShowPen.

To use the polygon, first place an insertion point before the beginning of the routine you just entered and type:

```
program Polygons;
uses
QuickDraw2;
var
polygon : polyHandle;
box1, box2 : RECT;
```

Now, place an insertion point past the **end;** and type:

```
begin {main}
SetRect(box1, 0, 0, 160, 160);
box2 := box1;
ShowDrawing;
CreateShape(polygon);
FramePoly(polygon);
repeat
InsetRect(box2, 1, 1);
MapPoly(polygon, box1, box2);
EraseRect(box1);
FramePoly(polygon);
until Button;
KillPoly(polygon);
end.
```

Notice the call to MapPoly. That routine changes the shape of the polygon so that it fits in *box2*. You use MapPoly to move polygons around. In that case, the second RECT would enclose the space where you want the polygon to apear. MapPoly changes the definition of the polygon so that it always appears in the new place. The first RECT, *box1*, is used for reference.

Also, notice the call to KillPoly. That routine frees the part of memory taken up by the polygon. You should always free up the space you use when you create a handle or a pointer. After you do so, the handle is invalid, and you get an error if you try to use it.

Run this program. It keeps on running until you hold the mouse button down long enough for the program to reach the Button routine.

Look in the QuickDraw interface at the end of the QuickDraw appendix of the Macintosh Pascal manual to see a list of all the polygon routines.

Pictures

A QuickDraw picture is a "recording" of a group of QuickDraw drawing calls. A picture is much like a polygon in that way. However, with a picture, you can use any routine that draws on the screen, and the shape does not have to be closed. A picture, in fact, can be as complicated or as simple as you like, anywhere from a single straight line to a diagram of the New York City bus system, and beyond those limits.

Like polygons, pictures are stored in memory space reached by handles. You define pictures in much the same way you define polygons. The most important difference is that when you define a polygon, the calls that make up the polygon's definition also define where the polygon appears. With pictures, you give a RECT when you define the picture. That RECT defines the original size of the picture. When you draw the picture, you give another RECT, which defines the size, shape, and location the picture will have when it appears on the screen. The RECT you give when you define the picture does not have to enclose the entire picture. It is merely for reference—it defines the scale of the picture. When you ask QuickDraw to draw the picture, the RECT you give when you define the picture is scaled to fit the RECT you give in the DrawPicture call, and the picture is scaled by the same amount.

The following program uses two pictures to simulate a square panel spinning through space. The panel has two sides, each of which is defined as a picture. First, one picture is displayed. The destination RECT is moved across the screen and squeezed as if the box were moving through space and turning. When the panel is flattened until it is apparently edge-on, the destination RECT encloses no space. At that point, the pictures are switched, and the back side of the panel is drawn. The destination RECT gradually grows to full size, and then shrinks again.

Here is the routine that defines the pictures. Don't type this.

```
procedure DefinePics (var pic1, pic2 : PICHANDLE; var box :
RECT);
begin
  SetRect(box, 10, 10, 10 + 2 *
HALFWIDTH,10+2*HALFWIDTH);
  pic1 := OpenPicture(box);
  with box do
DrawLine(left, top, right, bottom);
  FrameRect(box);
  ClosePicture;
  pic2 := OpenPicture(box);
  PaintRect(box);
  FrameRect(box);
  ClosePicture;
end;
```

You can insert any drawing calls between the OpenPicture and ClosePicture calls. You should *never* call OpenPicture a second time before calling ClosePicture. When you are done with a picture, you should always use the predefined KillPicture procedure to free the memory space used to store the picture.

Get a new programming window and type:

```
program Twirl_Box;
uses
QuickDraw2;
constHALFWIDTH = 10;
var
box, drawingRect : RECT;
```

```
velX, velY : INTEGER;
pic1, pic2 : PICHANDLE;
procedure DrawBox (pic1, pic2 : PICHANDLE;switch:
BOOLEAN);
begin
if switch then
DrawPicture(pic1, box)
else
DrawPicture(pic2, box)
end;
procedure Bounce (var velX, velY : INTEGER);
begin
 if (drawingRect.top > box.top) or (drawingRect.bottom <
 box.bottom) then
 velY := -velY;
 if (drawingRect.left > box.left) or (drawingRect.right <
  box.right) then
  velX := -velX;
end;
procedure SetUp (var drawingRect : RECT;
var velX, velY : INTEGER);
begin
velX := 1;velY := 1;
ShowDrawing;
GetDrawingRect(drawingRect);
GlobalToLocal(drawingRect.topLeft);
GlobalToLocal(drawingRect.botRight);
drawingRect.right := drawingRect.right - 16; {The -16 is for the
width of the scroll bar.}
drawingRect.bottom := drawingRect.bottom - 16;
PenMode(patXOr);
end;
procedure DefinePics (var pic1, pic2 : PICHANDLE;
var box : RECT);
begin
```

You define regions in much the same way as you define polygons and pictures: you call OpenRgn, make some drawing calls, and then call CloseRgn. There are some differences: you create a region handle first, before calling OpenRgn, and you give the handle when you close the region.

Another important difference is that you can change the shape of an existing region. You can do that by adding two regions together, subtracting one region from another, or finding the intersection of two regions. You can also expand or contract a region. When you define a picture or a polygon, that picture or polygon has a definition that remains the same until you kill the picture or polygon.

The following program creates a region that consists of a few separate shapes. The program then allows you to move the shape around by using the mouse. Here is the part of the program that defines the region. Don't type this.

```
region := NewRgn;
OpenRgn;
SetRect(tempRect, 20, 20, 50, 50);
FrameOval(tempRect);
SetRect(tempRect, 100, 70, 120, 80);
FrameRect(tempRect);
SetRect(tempRect, 80, 20, 110, 50);
FrameOval(tempRect);
Line(100, 100);
CloseRgn(region);
```

The variable *region* is defined as a RGNHANDLE. You can use line drawing calls in defining a region. However, the calls must result in a closed space in order to be added to the region. As with other handles, you should call DisposeRgn to free the part of memory that stores the region's definition.

Get a new programming screen and type:

```
program regions;
uses
QuickDraw2;
var
region : RgnHandle;
mouse : POINT;
procedure CreateShape (var region : RgnHandle);
var
tempRect : Rect;
begin
region := NewRgn;
OpenRgn;
SetRect(tempRect, 20, 20, 50, 50);
FrameOval(tempRect);
SetRect(tempRect, 100, 70, 120, 80);
FrameRect(tempRect);
```

```
SetRect(tempRect, 80, 20, 110, 50);
FrameOval(tempRect);
Line(100, 100);
CloseRgn(region);
end;
procedure MoveRegion (var region : RgnHandle; mouse :
POINT);
var
lastMouse : POINT;
begin
EraseRgn(region);
SetRect(box, 10, 10, 10 + 2 *
HALFWIDTH,10+2*HALFWIDTH);
pic1 := OpenPicture(box);
with box do
DrawLine(left, top, right, bottom);
FrameRect(box);
ClosePicture;
pic2 := OpenPicture(box);
PaintRect(box);
FrameRect(box);
ClosePicture;
end;
procedure Display (pic1, pic2 : PICHANDLE;
velX, velY : INTEGER; var box, drawingRect : RECT);
var
switch : BOOLEAN;
n, changer : INTEGER;
begin
changer := 1;
repeat
for n := 1 to HALFWIDTH do
begin
InsetRect(box, changer, 0);
OffsetRect(box, velX, velY);
DrawBox(pic1, pic2, switch);
Bounce(velX, velY);
DrawBox(pic1, pic2, switch);
end;
if EmptyRect(box) then
switch := not switch;
changer := -changer;
until Button;
end;
```

```
begin {main}
SetUp(drawingRect, velX, velY);
DefinePics(pic1, pic2, box);
Display(pic1, pic2, velX, velY, box, drawingRect);
KillPicture(pic1);
KillPicture(pic2);
end.
```

Regions

Regions, like pictures and polygons, are arbitrarily large descriptions of shapes. Like polygons, and unlike pictures, regions must be closed; there can be no part of a region that does not enclose something. Regions, in fact, are all border. Like polygons, you can frame, paint, or fill regions. You can also combine regions together, find out things about regions (such as whether a given point or rectangle is within the region) and more regions around. Regions, generally, define areas of the screen. In fact, the QuickDraw appendix says that a region divides the screen into two parts: the part that is in the region and the part that is not in the region. One thing to keep in mind, though, is that a region does not have to be contiguous: it can consist of several pieces spread over different parts of the screen. Figure 9-2 shows some possible regions. The most important point about regions is that they can contain any number of parts of the screen, and the parts can have any shape, as long as the shape is two-dimensional (that is, as long as the shape encloses some space).

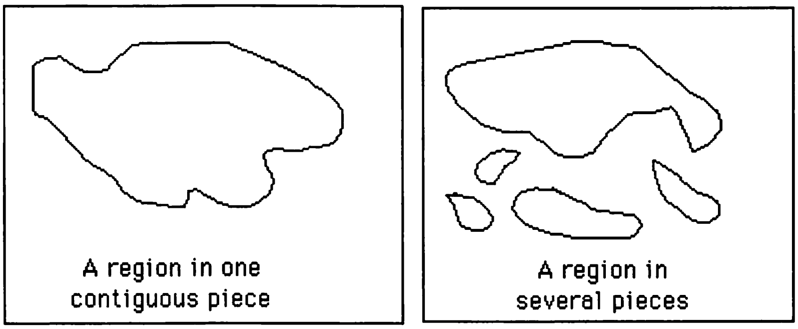


Figure 9-2 Possible Regions

```
PenMode(patXOr);
repeatlastMouse := mouse;
GetMouse(mouse.h, mouse.v);
SubPt(mouse, lastMouse);
OffSetRgn(region, -lastMouse.h, -lastMouse.v);
FrameRgn(region);
FrameRgn(region);
until not Button;
PenNormal;
PaintRgn(region);
end;
begin {main}
ShowDrawing;
CreateShape(region);
PaintRgn(region);
repeat
begin
repeat
until Button;
GetMouse(mouse.h, mouse.v);
if PtInRgn(mouse, region) then
MoveRegion(region, mouse);
end;
until FALSE;
DisposeRgn(region);
end.
```

Run the program. You can move the region by placing the mouse pointer somewhere in some part of the region, and pressing the mouse button.

You must use the Pause menu to stop this program.

Cursors

The little picture on the screen that moves in concert with the mouse is properly called a pointer. QuickDraw calls it a cursor to avoid confusion with the data type pointer, which has nothing to do with the little picture that appears on the screen.

You probably have noticed that the cursor has different shapes at different times. Sometimes it is an arrow, sometimes it is a cross, sometimes it is an I-beam. Those are predefined shapes the system uses. You can define your own cursors, as long as they form a 16-by-16 grid.

Pixels are defined by **bits**. A bit is the lowest level of information on a computer. A bit can have a value of 1 or 0. You can also think of that as On or Off. (The numbers 1 and 0 and the notions of On and Off are so entangled in the computer world that when the designers of the Macintosh wanted a way of labeling the On/off switch on the back of the Macintosh that wouldn't have to be translated into other languages, they chose 1 and 0.)

When a bit that corresponds with a pixel on the screen has a value of 0, the pixel is white. When the bit has a value of 1, the pixel is black. To define a cursor, you need to give the computer a bunch of 1's and 0's. Take a piece of graph paper, count out a 16-by-16 grid, and decide which boxes should be black and which should be white. The black boxes require a bit value of 1, and the white boxes require a value of 0.

Here is the complication: You cannot deal directly with bits in Pascal.

Although computers ultimately deal with nothing but bits (1's and 0's), those bits are always grouped together and translated into a more compact form, hexadecimal numbers, also called base 16.

Ordinary numbers, the ones you learned to count in, are in the decimal system, or base 10. That means that there are 10 digits: 0,1,2,3,4,5,6,7,8, and 9. Notice that the number 10, for which the system is named, is not a digit, but a combination of two digits.

Bits are in the binary numbering system, or base 2. Base 2 has only two digits: 0 and 1. The biggest problem with base 2, aside from the fact that people have trouble dealing with it, is that numbers take up a lot of space. The decimal number 100, for example is 1100100 in base 2. Decimal 1000 is 1111101000 in base 2.

The hexadecimal numbering system, base 16, is used mostly because it is more compact than base 2, and it is relatively easy to convert numbers from base 2 to base 16 and back. There are 16 hexadecimal digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

It isn't hard to convert small decimal numbers to hexadecimal: decimal 3 is a 3 in hexadecimal, and 9 is a 9 in both systems. A decimal 10 is an A in hexadecimal, a decimal 11 is a B, decimal 12 is C, decimal 13 is D, decimal 14 is E, and decimal 15 is E in hexadecimal. After that, you may find it more confusing.

Decimal 16 is 10 in hexadecimal. In every numbering system, 10 is the number that occurs when you run out of single digits. 10 in base 2 is equivalent to a 2 in base 10. What do you think the value of 10 in base 50 is in base 10? That's right, the value is a decimal 50.

When you define a cursor, you must give the cursor's definition (the data that says which pixels are On and which are Off) as a string of hexadecimal digits, so you need to know how to convert the binary digits you got from your graph paper into hexadecimal digits.

It is relatively easy to convert from binary to hexadecimal because each hexadecimal digit corresponds to four binary digits. Take the following four digits:

1111

The way you convert from one system to another is to find out the value of each digit that appears in the number you are trying to convert, change that value to the new numbering system, and add all of those together.

To make this easier, think back to your early days of learning addition. Each place in a number has a name. In the decimal system, the place on the right end is the ones' place, the next over is the tens' place, the next over is the hundreds' place, and so on. Look at this decimal number:

1111

Think of that as one thousand, one hundred, and eleven. To arrive at that value, you unconsciously multiplied the digit at that place by the base value of the place, and then added the result together.

```
(1*1)
+ (1* 10)
+ (1* 100)
+ (1* 1000)
```

Take another number:

5785

To read the value of that, you unconsciously add:

(5*1)
+ (8* 10)
+ (7* 100)
+ (5* 1000)

You do exactly the same thing with a number in a different numbering system, except that the value of the places change. To convert the number 1111 from binary to decimal add:

(1*1)
+ (1* 2)
+ (1* 4)
+ (1* 8)

The value of binary 1111 is therefore decimal 15. Notice that 15 is the largest single digit that can be expressed in base 16: F. 1111 is the largest binary number that can fit in four spaces, and corresponds to the largest hexadecimal number that can fit in a single space.

Suppose you had a string of binary digits like this:

100101111001101101111011111000000110001

To convert that to the equivalent list of hexadecimal digits, first break it down into groups of four, like this:

1001 0111 1001 1011 0111 1011 1111 0000 0011 0001

Now, you can convert each group of four into the equivalent hexadecimal digit by using the method just shown. Try it yourself before looking at the answer. Remember that the hexadecimal digits A, B, C, D, E, and F are equivalent to the decimal numbers from 10 to 15.

Here is the answer:

```
  9    7    E    B    7    9    F    0    3    1
1001 0111 1101 1011 0111 1011 1111 0000 0011 0001
```

Figure 9-3 has a chart showing all the conversions for all of the 16 possible combinations of four binary digits. You can use that chart to convert any picture from pixels to hexadecimal digits.

binary	hex	decimal
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Figure 9-3 Binary, Hexadecimal, Decimal

Once you have your picture converted from pixels to a string of hexadecimal digits, you must get those digits into the Mac's memory in some way. You can't just put them into a string variable, because Macintosh Pascal then treats them like characters, and makes a conversion before storing them in its internal representation. You can't assign them to a numeric variable, because Pascal again makes a conversion before storing them. Pascal does a wonderful job of separating you from the internal workings of the computer, to such an extent that it can get in your way at times. Fortunately, QuickDraw provides a way of shoving a string of hexadecimal values into memory directly, with no conversion, by using the StuffHex procedure. One danger with StuffHex is that it totally avoids Pascal's protection; you must be careful where you point that procedure, and you also must be careful you don't use it to write more than its target can hold.

Cursors are not defined with handles, but with an ordinary variable of a special, predefined type called a CURSOR. That variable can hold 64 hexadecimal digits, exactly enough to define the 16-by-16 grid that makes up a cursor. StuffHex places the string of hexadecimal digits at a place in memory. You can use the @ operator (which is not part of standard Pascal) to get the address of a variable. Here is the line that is used in the following program:

```
StuffHex(@face,'07800FC0303060186018CCCC8CC48004400
843883450186007C0038000000000');
```

The variable *face* is of type CURSOR. The string defines a small, bearded face. Notice that the rows are all ignored. QuickDraw knows to divide that string up into groups of four hexadecimal digits, each of which defines a line of the picture. There are 16 lines altogether.

A CURSOR is a record, with the following definition:

```
CURSOR = record
  data : array [0..15] of INTEGER;
  mask : array [0..15] of INTEGER;
  hotSpot : POINT
end;
```

The *data* is the list of pixel values you just defined. (You don't have to name the field *face.data* in the call to StuffHex because *data* is the first field in the record.)

The *mask* determines the way the cursor image appears on the screen. You can define a mask with as much detail as the cursor image itself. The easiest thing to do is to set each integer of the *mask* to 0, which sets all bits of the *mask* to 0. If you do that, drawing under the cursor image can be seen through the cursor. The I-beam cursor used in MacWrite has all *mask* bits set to 0. If you set all the bits in the *mask* to 1, the cursor is opaque. The arrow cursor you often see on the Macintosh screen has the part of the *mask* corresponding to the body of the arrow filled with 1's, along with a narrow border around the arrow, while the rest of the cursor *mask* is filled with 0's. That is why the cursor box is invisible, but a small border shows around the arrow when you move it over black backgrounds.

The *hotSpot* is the part of the cursor that determines the location returned when you ask for the position of the mouse. The arrow cursor, for example, has a *hotSpot* of (0,0), which is the upper left corner of the cursor, while the cross-bar cursor has a *hotSpot* of (8,8), which is the center of the cursor.

Once you define a cursor variable, you display it on the screen by calling SetCursor.

Here is the sample program. Get a new programming screen, and type:

```
program Cursor__Face;
var
face : CURSOR;
procedure SetUpCursor;
var
n : integer;
begin
StuffHex(@face,'07800FC0303060186018CCCC8CC48004400
843883450186007C0038000000000');
for n := 0 to 15 do
face.mask[n] := 0;
face.hotSpot.v := 16;
face.hotSpot.h := 0;
end;
begin
SetupCursor;
SetCursor(face);
repeat
until button;
end.
```

The program runs until you press the mouse button.

Bitmaps

The entire Macintosh screen is defined in the way you define a cursor: by strings of 1's and 0's. You can directly change those onscreen values, or you can create pictures with strings of values off the screen, and then display them on the screen later. Those strings of 1's and 0's that correspond to images that may or may not actually be displayed, are called **bit images**. A bit image is a string of bits that may be mapped (transferred and translated) to the screen.

At its most basic level, the Macintosh always deals with the screen in terms of bit images. All the other drawing is done using bit images. Bit images, therefore, are the most basic and most flexible way of dealing with the screen. On the other hand, they are also the most difficult.

When you create a bit image, you can use the StuffHex procedure you used in the section on cursors to put hexadecimal values into some part of memory. (If you don't know what hexadecimal values have to do with pictures, read the section on cursors in this chapter.) The hexadecimal values alone don't really contain enough information for the QuickDraw to turn those bits into a picture for the screen. You need to tell QuickDraw how long each row is, and how big the resulting picture should be. When a bit image is combined with those values, the row width and a RECT that defines the size of the picture's boundaries, it is a bitmap.

You use the QuickDraw type BITMAP to create variables to store bitmaps. A BITMAP is a record. Here is the definition:

```
BITMAP = record
  baseAddr: QDPTR; {QuickDraw pointer type.}
  rowBytes: INTEGER;
  bounds: RECT;
end;
```

You put the address of the string of hexadecimal characters in the *baseAddr* field. You put a number in *rowBytes* that indicates the number of bytes in each row of the final picture. You then place a RECT that defines the size of the picture. The bit image begins at *bounds.upperLeft*, and each *rowBytes* bytes of the hexadecimal string is aligned with the left edge.

One point you may find confusing is that the *bounds* field does not change the size of the picture. Each bit of the string defines one pixel. The *bounds* can cut the picture down, so that parts of it are ignored. There are two reasons to do this: one is that a bit image may be used for several bitmaps, and the bitmaps may use different *bounds* (or even different *rowBytes*) to change the resulting picture. The other reason is illustrated in the sample program for this section:

Each row of a bitmap *must* contain an even number of bytes. An INTEGER variable takes up two bytes, so you are always safe if you use INTEGER arrays for your strings. The proportions of the picture may work out so that you only need an odd number of bytes in each row. In that case, you must still use the extra byte in each row, but you can adjust the *bounds* so that that byte is ignored in assembling the final picture.

In addition, the address of the bit image *must* be even. That shouldn't be a problem as long as you assign @variableName as the *baseAddr*. If you assign a *baseAddr* in another way, be careful that the resulting address is always an even number.

Once you have a bitmap defined, you can use the QuickDraw procedure CopyBits to display it. CopyBits copies bits from one bitmap to another. The screen is just another bitmap. The current screen image is stored in a special variable maintained by the Macintosh system: *grafPort^.portBits.* You can access *portBits* directly, as if it were any other bitmap. You can use CopyBits to copy bits from *portBits* to your own bitmap, or to copy a section of the screen from one part of the screen to another.

The most common situation, though, is copying from your own bitmap to *portBits*. When you call CopyBits, you give a RECT in your bitmap and a RECT in *portBits*. The section of your bitmap specified by the first RECT is enlarged or shrunk, if necessary, to fit the RECT in *portBits*. If you give *yourBitmap.bounds* as the first RECT, the entire picture is transferred to the screen, and enlarged or shrunken, if necessary, to fit in the destination rectangle.

CopyBits also can take a transfer mode, of the type you've used before. The transfer mode determines the way the bitmap changes whatever is already on the screen. If you use a transfer mode of *srcCopy*, the bitmap completely replaces whatever was already in the destination rectangle.

In addition, CopyBits takes a handle for a *mask* region. Give **nil** for that field, unless you need to hide part of the bitmap.

Here is a program that demonstrates the use of a bitmap. It creates a small, 24-by-24 pixel image, and displays it on the screen. The method used is similar to the method used to create a new cursor. One difference is that a two-dimensional array is used to store the string of hexadecimal digits. This is done for purely human reasons: it is easier to get it right when you can think of the picture a row at a time, instead of as one long string of bytes.

In Pascal, a two-dimensional array can be an array of another array. Any array in Pascal has the elements of the array lined up in memory, one after another.

It takes up some extra space to use a two-dimensional array, and only gains you a little bit in comprehensibility, so, if you write a program that uses bitmaps, and you find that you are running out of space, you can load the entire string with one StuffHex call and change your array to a one-dimensional array.

Notice that each row contains two integers, and there are 24 rows. How does that result in a 24-by-24 picture? Each integer has two bytes, or 16 bits. The *bounds* field of the bitmap is set to a 24-by-24 rectangle, so the rightmost eight bits in each row are ignored. Notice that the *rowBytes* figure is 4, which is an even number, as required, even though the picture only requires three bytes (24 bits).

Get a new programming screen and type:

```
program Sherlock;
uses
QuickDraw2;
type
rows = array[1..2] of INTEGER;
imageType = array[1..24] of rows;
var
image : imageType;
map : BITMAP;
displayRect : RECT;
n : INTEGER;
procedure StuffImage (var image : imageType);
var
n : INTEGER;
begin
StuffHex(@image[1][1],
'01E0000003D8000005BE00000A7B0000');
StuffHex(@image[5][1],
'0DBF00001E0DF8001FFFF0001FF 8000');
StuffHex(@image[9][1],
'1753800016D800001ADC40003F3EE000');
StuffHex(@image[13][1],
'7BFC0000EFFF00008EFFC0000B1F6000');
StuffHex(@image[17][1],
'06CE6E0006CFFC0007D21C0003EBEC00');
StuffHex(@image[21][1],
'0FE7F0000FE3F00017C3F0001FC3F000');
end;
procedure PrepareMap (var image : imageType;
var map : BITMAP);
```

```
begin
map.baseAddr := @image;
map.rowBytes := 4;
SetRect(map.bounds, 0, 0, 24, 24);
end;
begin
ShowDrawing;
StuffImage(image);
PrepareMap(image, map);
SetRect(displayRect, 256, 171, 280, 195);
for n := 1 to 100 do
begin
CopyBits(map, thePort±.portBits, map.bounds, displayRect,
srcCopy, nil);
OffSetRect(displayRect, Random div 600, Random div 600);
end;
end.
```

Before you run the program, enlarge the Drawing window so it fills most of the window. Now run the program.

The main program displays the bitmap 100 times at randomly selected locations. The QuickDraw Random function returns a random number between −32767 and 32767. Dividing Random by 600 assures a random value from −5 to 5. The OffSetRect routine changes the definition of the RECT given by adding the two values given to the top, bottom, left, and right values of the RECT, so that the rectangle is shifted without its size changing.

The picture shown is a small head of Sherlock Holmes. I got the values for that picture by tracing the original using some transparent graph paper from an arts supply store. That process is referred to as **digitizing** a picture. You lay the graph paper over the original and decide which boxes should be black and which should be white. That is essentially what is done by a digitizing camera, such as the ones you can buy that can put pictures in MacPaint format.

The main problem with bitmaps is that they take up a lot of space in memory. The Macintosh screen takes up 21,888 bytes. You cannot store many large pictures in memory. Bitmaps are most useful for creating small images, such as game pieces.

You can also use the QuickDraw ScrollRect procedure to scroll bitmaps, in the way the windows are scrolled on the Macintosh. See the QuickDraw appendix of the *Macintosh Pascal Technical Appendix* for details of that procedure.

Patterns

When you draw with the pen, the current pen pattern is used (along with the current mode) to determine exactly which pixels are turned black and which are turned white.

You have seen how you can set the pattern to the predefined patterns: *black, gray, ltGray, dkGray,* and *white*.

You can also define your own patterns.

A pattern is an 8-by-8 bit image. You can use the predefined type PATTERN to store a pattern. You define the pattern the way you define cursors and bit images: with a string of hexadecimal digits. This program sets the pattern to a boomerang form:

```
program NewPattern;
var
pat : PATTERN;
x, y : INTEGER;
begin
StuffHex(@pat, 'E01C0E07070E1CE0');
GetMouse(x, y);
MoveTo(x, y);
PenPat(pat);
PenSize(8, 8);
repeat
until Button;
while Button do
begin
GetMouse(x, y);
LineTo(x, y);
end;
end.
```

Run the program. When you move the mouse pointer into the Drawing window and press the mouse button, you can draw lines with the new pattern. When you let the button up, the program stops. A sample of the program's output is given in Figure 9-4.

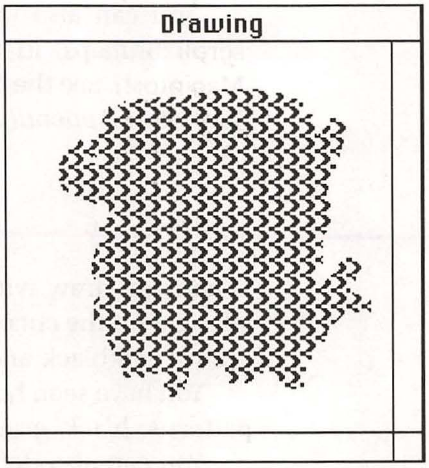


Figure 9-4 Sample Pattern

One point about the program: notice that the pen size is increased to show the pattern. The width of the line drawn is independent from the current pattern.

Notice that the patterns produced are fully lined up. Drawing with a pattern is like scraping off the white of the screen to reveal the pattern laid underneath, rather than painting a pattern that is laid down wherever the pen goes. In other words, the pattern is fixed to the Drawing window, not to the pen.

Do More

1. You can do most of the operations with regions that you can do with RECTs, including framing (FrameRgn), erasing (EraseRgn), testing (PtInRgn) and clipping (SetClip). Try modifying the Field Editor so the user can draw a field of any shape and edit within it. (If you have a 512K Macintosh, you can try to make these changes to the *Multiple Field Editor*, otherwise, use the simpler *Field Editor*.)

2. Create a digitized image for use as a game piece by using a bitmap. Try creating a similar image using a picture and display both with the same program. Which is faster?
3. Use a series of pictures or bitmaps or a routine using several pictures or bitmaps to simulate an explosion of some simple shape, like an oval or a polygon. If you use bitmaps and you have a 128K Macintosh, make the images small.
4. Create a simple game using the advanced QuickDraw functions. You could make a game using the exploding image you created from the previous suggestion, by creating a cursor that constantly changes shape, pointing in different directions. When the user presses the mouse button, have a dot move across the screen. When the dot hits the target, the game piece blows up. Better yet, make up your own game.

Hints for the suggested game:

Create the changing cursors so they are arrows, with the hotSpot at the tip of the arrow. Have a POINT value associated with each cursor that defines the movement of a projectile fired with that cursor. For example, if the cursor points towards the lower right, the POINT value is (1,1). When the user presses the mouse button, use GetMouse to find the mouse position (or use an event) and add the POINT value to it repeatedly. Store each cursor and its POINT value together in a record, and store all the records in an array. Use the Random function to choose cursors from the array.

Keep testing if the projectile is still on the screen by using PtInRect, and if it has hit the target by using PtInRgn. You can create a region for any shape by calling OpenRgn before drawing, and CloseRgn after. Also, call ShowPen if you want to actually draw on the screen after calling OpenRgn, because OpenRgn hides the pen.

QUICK SUMMARY

This chapter explores most of the capabilities of QuickDraw not covered in other chapters. Along the way, it explains binary and hexadecimal numbers. The following routines and concepts were introduced.

Binary number	is a number in base 2.
Bit	is the smallest piece of information that exists on a computer. A bit has a value of 1 or 0.
Bit image	is a collection of bits in memory that can be divided on even byte boundaries into rows that can be combined to form a picture. The screen is an example of a bit image that is automatically displayed by the Macintosh's hardware. Generally, you display a bit image by using bitmap routines.
Bitmap	is a type of variable that combines a bit image with a coordinate system. A bitmap tells QuickDraw exactly how to display the sequence of bits that are defined by a bit image.
ClosePicture	is a predefined procedure that stops recording of drawing calls for a picture. The definition of the picture is fixed after this call.
ClosePoly	is a predefined procedure that stops recording of line-drawing calls for a polygon. The definition of the polygon is fixed after this call, except the polygon may be scaled and moved with MapPoly.
CloseRgn	is a predefined procedure that stops recording of the borders of a region and saves the region using a given region handle. The region handle must have been created with NewRgn.
CopyBits	is a predefined procedure that copies bits from one bitmap to another. It is often used to display bitmaps on the screen.
Cursors	are the little images, usually called pointers, that indicate the current mouse position.
Decimal number	is a number in base 10, which is the commonly used numbering system.
Digitizing	is the process of converting information, often pictures, from continuously varying values to on/off values. Computers generally can only handle digital values. Continuously varying values are called analog values.
DisposeRgn	frees a region's handle.

DrawPicture	is a predefined procedure that displays a picture. It takes a picture handle and a RECT as arguments. The RECT is compared with the RECT given in the original call to OpenPicture to find the position and size of the displayed image.
EraseRgn	is a predefined procedure that erases everything in a given region.
FramePoly	is a predefined procedure that draws a polygon.
FrameRgn	is a predefined procedure that draws a region.
Handle	is a special kind of pointer that points to another pointer. Handles are used so that Macintosh can move variables around in memory more easily, thus allowing memory to be used more efficiently.
Hexadecimal number	is a number in base 16.
hotSpot	is the part of the definition of a cursor that defines the point considered the mouse position.
Interface	is what defines how you access a routine. Collections of routines, like QuickDraw, generally have an interface unit that gives the formal parameter lists for all of the routines. The routines are actually implemented in an implementation unit. This kind of division has many advantages—including allowing the interface to be in Pascal while the implementation is in more efficient assembler code.
KillPoly	is a predefined procedure that frees a polygon's handle.
KillPicture	is a predefined procedure that frees a picture's handle.
MapPoly	is a predefined procedure used to move polygons around. It moves and scales a polygon so it fits in a given box. It takes two RECTs, the first of which must be a RECT containing the polygon. The polygon is scaled and moved so it keeps a relatively similar position in the second RECT.
Mask	is the part of the definition of a cursor that defines how a cursor appears when it crosses images on the screen.
NewRgn	is a predefined function that returns a handle for a region. It *does not* begin recording drawing for a region. See OpenRgn.
OpenPicture	is a predefined function that returns a picture handle and starts recording drawing calls for the picture. It takes a RECT argument that defines the scale of the picture.
OpenPoly	is a predefined function that returns a handle for a polygon and begins recording line drawing calls for the polygon's borders.

OpenRgn	is a predefined procedure that begins recording drawing calls for the borders of a region. It does not take any arguments, and the region handle used for the region is not determined until CloseRgn is called.
PaintRgn	is a predefined procedure that fills a given region with the current pen pattern.
Pattern	is an 8-by-8 pattern of bits used in drawing on the screen. The pen repeatedly uses the pattern when drawing lines or shapes.
Pictures	are sequences of QuickDraw drawing calls which can be played back as a unit.
Polygons	are closed shapes with any number of sides.
PtInRgn	is a predefined Boolean function that tells if a point is in a given region.
QuickDraw2	is the part of QuickDraw that contains most of the advanced functions. To use these, you must include the lines: **uses** QuickDraw2 at the beginning of your program, immediately after the **program** statement.
Regions	are arbitrarily shaped, closed figures. A region consists only of a border edge, and can be made of any number of unconnected parts.
Scaling	is when an image is made larger or smaller, but the proportions are kept the same.
SetCursor	is a predefined procedure that changes the current cursor to the given cursor.
StuffHex	is a predefined procedure that places a string of hexadecimal numbers at a location in memory.

CHAPTER

10

Macintosh Math

Math on the Macintosh

When you get down to basics, the business of a computer is doing math. All the rest—word processing, sound generation, graphics, whatever—is built on a mathematic foundation. This chapter discusses math on the Macintosh.

It begins by explaining the significance of the different numeric data types. The chapter discusses integer- and real-type numbers, and the data types used to store those numbers.

The chapter then describes and what a **mathematical expression** is. Expressions are made up of **operators** and **operands**.

Operators are symbols that let you do addition, subtraction, multiplication, division, and a couple of other operations.

Operands can be values or names that represent values (variables and named constants) or **functions**, which are subprograms that may do complicated tasks, such as finding the trigonometic sine of a given angle.

The Importance of Data Types

Nearly all numbers used in this book are integers. An integer is a whole number value, such as 1,3,5, or 32776.

Many operations, however, result in values that are not whole numbers. In those cases, you must use one of a group of data types that can hold fractional values.

Those types are called real-types. A real-type might be 3.14959, 1.0000, .5, or 32766.99.

You can assign an integer-type (INTEGER and LONGINT) to a real-type variable. You cannot, however, do the reverse. Two of the Pascal standard functions described in this chapter, Trunc and Round, convert real values to integer values. For example, AReal := AnInt is legal, but AnInt := AReal is always illegal. AnInt := TRUNC(AReal) is legal, however.

Variables and Constants

Constants are either:

- Values that appear explicitly in the program, such as numbers.
- Named constants, which are defined in the **const** section of the program. These look like variables, but their values cannot be changed during the executation of the program.

Variables are named items which have values. Using the name of the variable is generally equivalent to giving a constant with the same value in that place.

The most important difference between variables and constants is that you can change the value of a variable, and you can't change the value of a constant. Both constants and variables can appear on the right side of the ":=" in a statement, but only variables can appear on the left side of the ":=".

Numbers

There are two basic groups of numerical values commonly used in programs: real-type numbers and integer-type numbers. Numeric variables acquire a type when they are defined; numeric constants also have types. A constant's type is the type of a variable that can hold it. In general, if the number has a decimal point, it is a real-type number, otherwise it is an integer-type number.

Integer-type numbers have whole-number values, as described before. Real-type variables can hold **real numbers**—numbers that may not be whole numbers. For example, 3.14 is a real number. The following two sections have discussions of INTEGERs, LONGINTs, and real-type numbers.

Precision

Mathematical operations often result in values that are not whole numbers. For example, 5 divided by 2 is 2.5. 100 divided by 3 is 33 and one third, or, approximately 33.33333. To express 33 and one third exactly as a decimal number, you would have to keep writing 3's forever. At some point, you need to decide that you have defined 100/3 precisely enough.

In Pascal, different data types allow you to specify a number with different levels of precision.

The precision of each data type is discussed along with the description of the data type later in this chapter. Precision means the number of separate digits that you can use in a number.

Integer Data Types

The two integer data types hold only whole number values. When you divide 100 by 3, for example, and put the result in an integer-type variable, the result is 33. The difference between the two integer data types is simply the magnitude of the largest number a variable of each type can hold.

INTEGER

INTEGER is a predefined data type that is used to represent whole-number numeric data. The value of an INTEGER can range from −MAXINT to MAXINT, where MAXINT is a predefined constant equal to 32767.

Notes

You cannot give commas in numbers in Pascal. The commas are given here for clarity.

The fact that you cannot have an INTEGER outside that range means that you can't, for example, have an INTEGER with a value of 1000000. The fact that all INTEGERs have whole-number values means that you can't, for example, have an integer with a value of 1/2.

The fact that INTEGERs can only take whole-number values also implies that INTEGER is an ordinal type, meaning that every value is part of an ordered, limited set.

There are two difficulties that result from INTEGER's limitations:

- INTEGERs outside the range -b1-32727 cannot exist.
- INTEGERs with fractional values cannot exist.

If you need to represent integer values outside the range ±MAXINT, use the LONGINT data type. (You can also use the COMPUTATIONAL data type, which is a special real-type which can only hold integer values.)

If you need to represent values that are not whole numbers, use one of the real data types.

LONGINT

LONGINT is an extension of the type INTEGER. An INTEGER takes up two bytes, or 16-bits, of space in memory. The value of MAXINT results from that space limitation. A LONGINT takes up four bytes, or 32 bits, of memory space. That results in the value for MAXLONGINT, the largest long integer, of 2147483647. A LONGINT can hold any value from −MAXLONGINT to +MAXLONGINT.

You can use LONGINTs almost anywhere you can use INTEGERs. You can even give a LONGINT in place of an INTEGER as a parameter to a subprogram that requires an INTEGER parameter—as long as the value of the LONGINT is in the range ±MAXINT.

LONGINT, like INTEGER, is an ordinal type. That means that the number of values is limited. You could list all the LONGINT values, if you wanted. (Although that would mean listing over four billion values.)

Real Data Types

In mathematics, the set of real numbers is the set of all numbers that can exist. That includes fractional values. These are examples of real numbers:

```
3.14159
0.0000000000000000000000
11812389872.95
100.0
```

There are a number of data types that are capable of representing a subset of real numbers.

There are an infinite number of real numbers. Suppose, for example, you divided 1 by 2, resulting in .5. You could then divide .5 by 2, resulting in .25. You could then divide .25 by 2, resulting in .125. You could keep on dividing by 2 forever, and, eventually, the string of numbers needed to specify the result would be so long that it wouldn't fit in this book, or in an entire library. A computer cannot represent values with that many digits. Every real data type has a limitation on the number of digits and on the magnitude of numbers that can be stored in variables of that type.

When you give a real constant in a program, you must have at least one digit to the left and one digit to the right of the decimal point.

Either or both of those digits can be 0. For example, 0.0 is a legal real number, but 0. and .0 are not.

Notes

You can also use integer numbers where real-type values are expected. However, if you use a decimal point, there must be at least one digit to the right *and* one to the left of the decimal point.

Real numbers are normally represented in Pascal in **floating-point notation**, also called **scientific notation**. Here is an example of floating-point notation:

```
1.2E3
```

The "E" means "10 to the power of." That expression can be read as "1.2 times 10 to the power of 3".

A "power" is a number multiplied by itself the power number of times. For example, 10 to the power of 3 is:

```
10 * 10 * 10
```

which equals 1000.

The value of 1.2E3 is 1200.0.

What makes floating-point notation convenient is that you can achieve the same result by moving the decimal point three places to the right.

```
1.2E3 = 12.0E2 = 120.0E1 = 1200.0
```

The number following the "e" is called the **exponent**, or **order of magnitude** of the floating-point number.

Floating-point notation is quite compact when dealing with large numbers. For example:

```
5.6E35
```

is equivalent to:

```
560,000,000,000,000,000,000,000,000,000,000,000.0
```

A "+" sign is often given before the exponent, to indicate it is a positive exponent. (If no sign is given, the exponent is positive, anyway.) For example:

```
5.6E+35
```

When the exponent is negative, the number has a magnitude less than 1. For example:

4.67E-3

is equal to:

0.00467

When a negative sign precedes the whole number, the number is negative. For example:

-1.0E-20

is equal to:

-0.00000000000000000001

There are two issues involved in the size of the numbers: **range** and **precision**.

Range is maximum size of the number following the "e".

Precision is determined by the number of digits that can be in the multiplier part of the number.

The real data types are REAL, DOUBLE, EXTENDED, and COMPUTATIONAL. The difference between REAL, DOUBLE, and EXTENDED types is the amount of storage space allocated for the number, and therefore the range and precision that can be achieved. COMPUTATIONAL is a special case, and is discussed below.

A REAL variable (the only one of these three defined in standard Pascal) has four bytes of storage. It can hold values from 1.5E−45 to 3.4E+38. It can have up to 7 or 8 digits.

A DOUBLE variable has eight bytes of storage. It can hold values from 5.0E10−324 to 1.7E+308. It can have up to 15 or 16 digits.

An EXTENDED variable has ten bytes of storage. It can hold values from 1.9E−4951 to 1.1E4932. It can have up to 19 or 20 digits.

As you can see, the real-types, especially DOUBLE and EXTENDED, can hold far larger and far smaller values than integer-types. They also take up much more room in memory.

COMPUTATIONAL is a special real-type that can only hold whole-number values. A COMPUTATIONAL variable can hold values in the approximate range ±9.2E18. You can have meaningful digits in all nineteen places.

All real-type values are converted to EXTENDED before any expressions are evaluated, and the result of all real operations is an EXTENDED value. You can always assign the results to DOUBLE, REAL, or EXTENDED variables, as long as the result is within the range of the target type.

Expressions

An expression is any group of variables, constants, operators, and functions that result in a single value. The following are all examples of expressions:

```
thisVar-454+(123-15)*thatVar div (thisVar mod thatVar)
3.5/15
3 div15
Sin(thisVar)
thatVar
5
-123
```

An expression can range from a single constant to a long group containing many elements.

In general, expressions are made up of **operators** and **operands**.

Operands can be constants (that is, regular numbers), variables, and functions. All operands must have definite values when an expression is executed. Operators are described in the next section.

Note that functions, constants, and variables can appear in expressions, but you can only assign the value of an expression to a variable, or use the expression in a subprogram call. A single value appears on the left side of a ":=" assignment. Whatever is on the right side of the ":=" is an expression.

Operators

Operators define the relationship of the operands. That is rather formal language for something you have seen since elementary school. Operators include the familiar " + " and " – " symbols, used for addition and subtraction. The multiplication operator, " * ", may look a bit odd, but it works like the familiar "X" you used in 4th grade.

Division gets a bit more complicated. There are two division operators: "/" and **div**. The **div** operator is used to divide INTEGERs and LONGINTs. The remainder is always dropped in integer division. The remainder is also called the modulus. For example, when you divide 5 by 2, there is a modulus of 1. The result of:

```
5 div 2
```

is 2. For that reason, a special operator is provided that produces only the modulus. It is the **mod** operator. The result of:

```
5 mod 2
```

is 1.

When you are working with real numbers, you use the "/" operator, which results in an exact answer. The result of

```
5.0/2.0
```

is 2.5.

To summarize, the following are the arithmetic operators:

+, –, *, div, /, and mod

Unary Operators

There are three operators that work on a single operand, which are therefore called **unary operators**. These work much like functions, in that they can change the value of their operand in some way.

Two of these unary operators are " + " and " – ", which, as you have seen are also binary operators (that is, operators that take two operands). When you use "+" and "-" as unary operators, they simply express sign, as they do in normal arithmetic. Here are examples of the use of unary operators:

```
+5
-3.3
-thisVar
```

The "+" operator has no real effect. +5 and 5 are equal representations. The "-" operator usually indicates a negative number—a number whose value is less than 0. It can also reverse the sign of a negative number. For example, the value of:

```
-thisVar
```

is positive if thisVar has a negative value.

The third unary operator is the "@" operator. The value of an "@" followed by a variable is the variable's address. An address is a **pointer-type value**. Suppose a program contains the following declarations:

```
thisVar : INTEGER;
thisPointer : INTEGER;
```

The following statement is valid:

```
thisPointer := @thisVar;
```

This statement is invalid:

```
thisVar := @thePointer;
```

Although an address is a whole number value, it is *not* an integer value, and cannot be assigned to an integer-type variable.

You can use the Ord function to convert a pointer-type value to an integer. You can use the Pointer function to convert an integer to a pointer-type value.

The "@" operator is not a standard part of Pascal, and you should be careful if you use it. In general, you should use it only when you need to pass a pointer-type to a library routine, as is done in Chapter 7 of this book when using the sound generator.

Boolean Expressions

You have used Boolean, or logical, expressions in previous chapters. In particular, the Boolean operators **and, or,** and **not** were explained in Chapters 2 and 3.

This section explains how to use relational operators to form Boolean expressions.

A Boolean expression, like a numeric expression, is a sequence of values connected by operators. With a Boolean expression, however, the result of each sub-expression and the overall expression must be a value of TRUE or FALSE.

You produce TRUE/FALSE values either with Boolean functions, such as the Button function, or by using relational operators on pairs of numeric or string expressions.

Relational Operators

A relational operator tests the relationship between two values, and returns TRUE or FALSE depending on the operator and the values.

There are seven relational operators. Here are six of them, along with the relationship that they test for. (The seventh operator is quite different and is discussed below.)

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

(Note that you have to type two characters for "<=", ">=" and "<>".)

You may have run across symbols and concepts like these in your math classes. Some of the symbols are different; "<>" is used for not equal in Pascal, instead of the more usual "≠".

You can use any of these operators between two values. The two values generally have to be of the same type. (You can compare real-type and integer-type values, however.)

Pascal checks the relationship, and gives the expression a value of TRUE if the given relation is TRUE, and FALSE if the given relation is FALSE. For example, all these expressions are TRUE:

```
5 > 3
5 <> 3
(3+2) = 5
'Yeats' <> 'Eliot'
'Me' = 'Me'
'ME' <> 'me'
```

All of these expressions are FALSE:

```
5 < 3
5 = 3
3 > 5
'A' > 'B'
```

When you compare character or string values, the character codes are compared. In general, with ordinal types (the group that includes INTEGERs, CHARs, BOOLEANs, pointers, and user-defined ordinal types) the Ord of the values are compared.

Note that although an expression like:

```
x > y > z
```

was allowable in math class, you cannot have more than two elements in a relational expression in Pascal. You can, though, use the Boolean operators **and** and **or** to combine relational expressions and achieve the same effect. For example:

```
(x > y) and (y > z)
```

You can use mathematical expressions within relational expressions. The mathematical expression is fully evaluated, and the resulting value is used for comparison. For example:

```
(thisVar-5+Ord('A')) > (thatVar*Trunc(aReal))
```

is perfectly legal. The mathematical expressions are surrounded by parentheses for the sake of clarity. In general, mathematical operations are evaluated before relational operations, but you can use parentheses to make any expression more clear. See the operator precedence section in this chapter for an explanation of the effect of parentheses.

The seventh relational operator is **in**. The **in** operator is used to check if a value is in a set. For example:

```
Eliot in [Eliot, Pound, Yeats]
```

is TRUE, while:

```
Williams in [Eliot, Pound, Yeats]
```

is FALSE.

Relational Expressions and Real Numbers

Consider the expression:

```
1.22222222 = 1.222222221
```

The value of that is FALSE, as it should be. Suppose, though, that you had a program that began the following way:

```
var
  aReal : REAL;
  anExtended : EXTENDED;
begin
  aReal := 10/3;
  anExtended := 10/3;
```

What do you think happens if you compare the values of *aReal* and *anExtended*? For example:

```
aReal = anExtended
```

returns FALSE. Why? Because an EXTENDED variable has more digits than a REAL variable, and therefore has a greater value in this case.

The message here is that relational operators are not well suited for real-type numbers, because the values of reals are generally approximations. For example, given the same program beginning given above:

```
(aReal * 3) = 10
```

is FALSE. While:

```
(anExtended * 3) = 10
```

is TRUE. That is because the value stored in *aReal* is 3.333333, while the value in *anExtended* is 3.3333333333333333333. Notice that in ordinary arithmetic, three times either value is less than 10, but the value of 3**anExtended* is close enough that Macintosh Pascal can't tell the difference.

If you need to test the value of a real number, you should allow it to be within some range of the target value. Suppose *aRange* is the range you've decided on. You could use an expression like the following:

```
Abs(anExtended - aReal) < aRange
```

The function Abs (described later in this chapter) makes certain the result of the first expression is a positive number. As long as *anExtended* and *aReal* are close enough together, this expression returns TRUE.

Operator Precedence

What is the value of the following expression?

```
3-2*2
```

Although you might have come up with a 2, Pascal produces a −1.

The reason is that Pascal evaluates operators in a certain order, referred to as the **order of precedence**. Pascal does multiplication before subtraction, even if the subtraction comes first, as it does in the expression given above.

You can put sub-expressions in parentheses to force evaluation in a different order. For example, the value of:

```
(3-2)*2
```

is 2.

Here is a chart of operator precedence. The operators on higher lines are evaluated before operators on lower lines.

unary +	unary -	not		
*	/	div	mod	and
+	-	or		

all relational operations

All operators on a given line have the same level of precedence. If there is more than one operator in a given level, they are evaluated from left to right.

Once again, everything in parentheses is done first. If there are several operators in a given set of parentheses, they are evaluated in the normal order of precedence.

Functions

One of the rules of good programming style can be summed up by two words:

"Be lazy."

Not always, you understand. But, when someone has already done work for you, use that work rather than redoing it. In particular, you should know and use the built-in functions.

There are three groups of functions in Macintosh Pascal.

- The standard functions. These are functions that exist in all versions of Pascal, and are the ones you will probably use the most.
- The Macintosh Pascal extensions. These are functions that are not part of every version of Pascal, but which are part of this version. The arithmetic functions of this group are documented in one of the following sections. There is also a group of string functions which are documented in Chapter 4. Many of these functions are similar or identical to extensions that exist in most other versions of Pascal.
- The SANE (Standard Apple Numerics Environment) extensions. You have to include the line

  ```
  uses SANE;
  ```

  at the beginning of your program (on the line after the program name) to use these functions. They are generally for use in advanced numerics applications; most programmers will have no need for them. For that reason, they are not covered in this book. See the SANE appendix of the *Macintosh Pascal Technical Appendix* for a description of the SANE package. (Note: the text of that appendix describes the SANE package in a general way that applies to all versions of SANE, including those on other computers. Following the text is a summary of the Macintosh SANE package that shows the parameter lists of all the functions and procedures included.)

Notes

QuickDraw contains a number of functions and procedures that perform arithmetic operations on POINTs and RECTs. Those are not covered in this book. See the QuickDraw appendix of the *Macintosh Pascal Technical Appendix* for information about those subprograms.

Standard Functions

One of the original purposes of Pascal was to create a standardized language that could be used on a wide variety of computers. In practice, most versions of Pascal include "extensions"—additional capabilities that are not included in standard Pascal. Although there are other differences, what most differentiates Macintosh Pascal from standard Pascal is the addition of many built-in functions and procedures. The functions in this sections are the only ones that are part of standard Pascal. Every other procedure and function you've used is an extension.

Many of these "standard functions" differ from the standard versions in one way: when the result type of the standard version is an INTEGER, the result type of the Macintosh version is LONGINT; similarly, when the result type of the standard version is a REAL, the result of the Macintosh version is EXTENDED. LONGINT and EXTENDED are not part of Standard Pascal. Within the range of the types INTEGER and REAL, these functions are equivalent to the standard versions. As usual, you can assign the result of the functions to an INTEGER or REAL variable as long is the value is within the range ±MAXINT, or within the range of the REAL data type.

- **Abs**

  The Abs function produces the **absolute value** of its parameter. The absolute value is the unsigned, or positive, value. In other words, given:

  ```
  Abs(aNumber)
  ```

  the result is −*aNumber* if *aNumber* is less than 0; the result is *aNumber* if *aNumber* is greater than or equal to 0.

For example:

Abs(-123)

produces 123.

Abs(10)

produces 10.

You can use an integer- or real-type parameter. For example:

Abs(-1.5)

produces 1.5.

The result is of type LONGINT if the parameter is of an integer-type, and of type EXTENDED if the parameter is of a real-type.

- **Arctan**

The Arctan function produces the **arctangent** of a numeric value. The arctangent is the angle whose tangent is the given value. (Tangent is a trigonomeric value equal to sine/cosine.)

The parameter must be greater than or equal to 0, and can be an integer or real number. The result is an EXTENDED value, expressing an angle in radians. (There are 2π radians or 360 degrees in a circle. Therefore, for example, π (pronounced like pie) radians equals 180 degrees. The value of π is approximately 3.14159.)

- **Chr**

The Chr function returns the character whose code is given as the parameter. The parameter must be an integer-type value in the range 0 to 255.

The result type is CHAR.

For example,

Chr(65)

produces 'A'.

Chr(13)

is equivalent to a Return.

The Ord function is the inverse of Chr. In other words:

Ord(Chr(13))

produces the integer value 13.

- **Cos**

  The Cos function produces the trigonomeric cosine of the parameter. The parameter is an angle expressed in radians.

- **Exp**

  The Exp function returns the natural exponential of its parameter. Given the function call:

  Exp(x)

  the return value is equal to "e^x". "e" is the base of the natural logarithm. The value of "e" is 2.718281828. . . (The ellipses (. . .) indicate that "e" is a repeating fraction: the digits, "1828", repeat infinitely.)

Notes

The "e" which is the base of natural logarithms has nothing to do with the "E" which is used in floating-point notation.

Exp is the inverse of the LN (natural log) function.

The result is of type EXTENDED. The parameter can be a real- or integer-type value.

- **LN**

  The LN function returns the natural logarithm of its parameter. The expression:

  LN(x)

  is equivalent to the mathematical expression:

  $\log_n(x)$

  The base of natural logarithms is "e", which has the value "2.718281828. . ." (Again the digits, "1828", repeat infinitely.) This is the inverse of the Exp function.

  The result is of type EXTENDED. The parameter can be an integer- or real-type value which is greater than 0.

- **Odd**

  The Odd function returns TRUE of its parameter is an odd number, FALSE if the parameter is an even number.

  For example:

  ```
  Odd(2)
  ```

  is FALSE.

  ```
  Odd(1111)
  ```

  is TRUE.

- **Ord**

  The Ord function returns the ordinal number of its parameter.

  The result is of type LONGINT.

  As usual, you can assign the result of Ord to an INTEGER variable as long is the value is within the range ±MAXINT.

  The parameter must be of an ordinal type, such as INTEGER, LONGINT, CHAR, BOOLEAN, an enumerated-type, or a pointer-type. For all ordinal types except INTEGER and LONGINT, the ordinal number is the sequential position of the value in its set of values.

  For example:

  ```
  Ord('A')
  ```

  produces 65.

  With integer values, the Ord of the value is the value itself.

  ```
  Ord(-112)
  ```

  produces -112.

  It is an error to use Ord on a non-ordinal value. Ord(1.2) for example, is illegal.

  Ord can be used on enumerated or set types. For example, where a program contains the following definition:

  ```
  fruits: set of (APPLE, ORANGE, KIWI);
  ```

  the value of:

  ```
  Ord(KIWI)
  ```

  is 2. The ordinal numbers of all ordinal types except for integer types begin at 0.

  Ord is most often used to obtain the character code of a character. The Chr function, in those cases, is the inverse of

Ord. Ord(*aValue*) produces the character whose character code is *aValue*.

The Pointer function (described in the Macintosh Pascal extensions section) is the inverse of the Ord of a pointer-type value.

The parameter can be any ordinal type. The result is of type LONGINT.

- **Pred**

The Pred function takes an ordinal-type value as a parameter, and returns the preceding value in that ordinal type. For example:

```
Pred(10)
```

returns 9.

You can apply this to any ordinal type. For example:

```
Pred('B')
```

returns "A".

The ordinal type can be programmer-defined. For example, suppose your program contains the following type definition:

```
poets = (WILLIAMS, LAWRENCE, POUND);
```

The following call:

```
Pred(LAWRENCE)
```

returns the value WILLIAMS.

The type of the result is the same as the type of the parameter. The parameter must be an ordinal-type. (You cannot, for example, use Pred with a real-type value.)

- **Round**

  Round converts a real-type value to a LONGINT value by rounding the real-type value to the nearest integer. For example:

  ```
  Round(1.9999999999999)
  ```

  produces 2.

  ```
  Round(1.4999999999999)
  ```

  produces 1.

  ```
  Round(1.5)
  ```

  produces 2.

  Use Trunc when you want to truncate the value to the next lowest (or highest, if the number is negative) integer.

- **Sin**

  The Sin function produces the trigonomeric sine of the parameter. The parameter is an angle expressed in radians.

- **Sqr**

  The Sqr function returns the square of its parameter. The square of a number is the number multiplied by itself. For example:

  ```
  Sqr(2)
  ```

  produces 4.

  ```
  Sqr(-12)
  ```

  produces 144.

  The result of Sqr is always positive.The result is a LONGINT value if the parameter is of an integer-type. The result is an EXTENDED value if the parameter is of a real-type.

- **Sqrt**

  The Sqrt function returns the square root of its parameter. The square root is the number which, when multiplied by itself, produces the parameter. For example:

  ```
  Sqrt(4)
  ```

  returns 2.0

  ```
  Sqrt(144)
  ```

  returns 12.0.

The parameter can be a real-type or integer-type value, but must be greater than or equal to zero.

The result is of type EXTENDED. If you need an integer result, you must use one of the conversion functions, Trunc or Round, to convert the result of Sqrt.

- **Succ**

  The Succ function takes an ordinal-type value as a parameter, and returns the next value in that ordinal type. For example:

  ```
  Succ(10)
  ```

  returns 11.

  You can apply this to any ordinal type. For example:

  ```
  Succ('B')
  ```

  returns "C".

  The ordinal type can be programmer-defined. For example, suppose your program contains the following type definition:

  ```
  poets = (WILLIAMS, LAWRENCE, POUND);
  ```

  The following call:

  ```
  Pred(LAWRENCE)
  ```

  returns the value POUND.

  The type of the result is the same as the type of the parameter. The parameter must be an ordinal-type. (You cannot, for example, use Succ with a real-type value.)

- **Trunc**

  Trunc, short for truncate, converts a real-type value to a LONGINT value by cutting off the fractional part. For example:

  ```
  Trunc(1.9999999999999)
  ```

  produces the LONGINT value 1.

  If the number is positive, the result of Trunc(number) is always less than the number. If the number is negative, the result of Trunc(number) is always greater than the number.

  Use Round when you want to round a real value to the nearest integer, instead of truncating to the next lowest (or highest, if the number is negative) integer.

Macintosh Pascal Extensions

The functions described in this section give you a few more useful resources. You do not have to do anything special to use these functions. You use them just as you would use the standard functions.

- **FixRound**

  The FixRound function takes a parameter of type FIXED and rounds it to the nearest INTEGER value.

- **HiWord**

  The HiWord function takes a LONGINT or FIXED value (which takes up four bytes of memory) and returns the value in the top two bytes as an INTEGER value. For example:

  ```
  HiWord(1000000)
  ```

  returns 15.

  The LoWord function returns the value of the lower two bytes.

- **LoWord**

  The LoWord function takes a LONGINT or FIXED value (which takes up four bytes of memory) and returns the value in the bottom two bytes as an INTEGER value. For example:

  ```
  LoWord(1000000)
  ```

  returns 16960.

  The HiWord function returns the value of the upper two bytes.

- **Ord4**

  This function acts in exactly the same way as Ord. See Ord in the standard functions section for more information.

- **Pointer**

  The Pointer function converts an integer-type value to a pointer-type value. A pointer-type value is an address. You can assign the result of the Pointer function to any pointer-type variable. Notice that this avoids Pascal's normal type-checking, and must be used with great care.

  This function is the inverse of Ord and Ord4 when they are used on pointer-type values.

- **Random**

The Random function returns a pseudo-random number from −32767 to 32767.

A random number is a number chosen at random, in the way a pair of dice chooses a random number from 2 to 12.

A pseudo-random number is a simulated random number. It is not truly random, but you would have trouble telling the difference.

In this case, the value returned depends on the system global variable *randSeed,* which is initialized to 1 when Macintosh Pascal starts. The value of *randseed* changes whenever you call Random.

If you need to repeat a sequence of pseudo-random numbers, such as when you are testing a program, you can set *randSeed* to some particular value. Whenever you reset *randSeed,* it produces the same sequence of numbers.

The fact that *randSeed* is a system global variable means you don't have to do anything special to access it. You can use its name in your program as if it were defined in your program's **var** part.

The result is an integer-type value. Random takes no parameters.(Random is actually a QuickDraw function. That fact makes no difference.)

- **SizeOf**

The SizeOf function returns the number of bytes occupied by the variable or type which is given as its parameter. For example:

```
SizeOf(INTEGER)
```

returns 2.

```
SizeOf(LONGINT)
```

returns 4.

Warning

Although SizeOf should be able to operate when given a data type as a parameter, as of Release Version 1.0 it must be given a variable as a parameter. In other words, in order to find out the size taken up by variables of a particular type, you must create a variable of that type and give the variable name to SizeOf. Doing this will not change the value of the variable in any way.

Fixed Point Functions

The following functions are used for arithmetic with fixed-point numbers. A fixed-point number is a four-byte number with the whole-number portion in the upper two bytes and the fractional part in the lower two bytes. They are primarily used as parameters to some Macintosh subprograms, such as StartSound.

Although a fixed-point number holds real number values, the type FIXED is defined to be equivalent to the integer-type LONGINT. A LONGINT takes four types of storage.

- **FixRatio**

  The FixRatio function returns the product of the division of two INTEGER values. The return value is of type FIXED. Note that unlike the integer division operator **div**, the FixRatio function produces an exact result. For example:

  ```
  FixRatio(7,2)
  ```

  returns an fixed-point number with 3 in the upper two bytes and 5 in the lower two bytes.

- **FixMul**

  The FixMul function multiplies two values of type FIXED. The result is of type FIXED.

Do More

1. You can use the information given in the explanation of hexadecimals and the value of bits in the cursor section of Chapter 9 to demonstrate to yourself why 16 bits leads to a MAXINT value of 32767. (A clue—one bit is needed for the sign.)
2. Use the button framework used to explore text in Chapter 4 to create a calculator that lets you use some of the mathematical functions and procedures provided by Macintosh Pascal.
3. A spreadsheet program is one of the most common and useful types of computer programs. A spreadsheet is made up of cells which can be connected using formulas. For example, two cells might be multiplied together, with the result deposited in a third cell.

   Create a mini-spreadsheet, using the *Database Framework* program. Use only three cells and use buttons to create the formula. You can use this method to create a checkbook balancer, for example.

QUICK SUMMARY

This chapter describes how to do arithmetic on the Macintosh. It discusses what expressions are, the important points about numbers, the significance of the difference mathematical data types, the difference between variables and constants, operators, and the mathematical functions available.

Because this chapter consists largely of simple descriptions of functions and data types, those are not included in this summary. The following concepts were introduced.

Expressions	are collections of operators and operands that produce a single value.
Floating-point notation	is a way of writing a number so it consists of a number multiplied by a power of ten. The first number is normally between 1 and 10.
Functions	are possibly complicated routines that produce a single value. Functions can be used as operands in expressions.
Operands	are values used in expressions. An operand can be an explicit value (a number), a named constant, a variable, or a function.
Operators	are symbols that let you do addition, subtraction, multiplication, division, and some other operations. Operators act on or combine operands in expressions.
Order of magnitude of a number	is the power of ten used for that number when written in floating-point notation.
Modulus	is the remainder of a division operation, which is produced by the **mod** operator.
Order of precedence	is the order in which operations in expressions are performed.
Precision	of a data type is how many characters can be given, and therefore how exact a value a variable of that type can hold.
Range	of a data type is the size of the largest number a variable of that type can hold.
Scientific notation	See Floating-point notation.

CHAPTER 11

Debugging

What is a Bug?

The story goes that, back at Harvard in the 1940's, the early days of electronic computers, one of those huge, multi-room-sized computers with thousands of tubes (there weren't transistors in those days, let alone chips) stopped working. When that happened, someone had to crawl inside and check every single tube to find which one was no longer working. Well, they looked and looked and couldn't find anything. Finally, someone reached in and found a cockroach that had crawled into the computer and ended its life by shorting a circuit.

Problems in computers and computer programs have been called "bugs" ever since.

There are two distinct kinds of bugs:

- A mistake in the way something is used. This is sometimes called a **syntax error**, because it is like a grammatical mistake in English. Macintosh Pascal usually catches these errors either after you type them, or when you try to run the program. Occasionally, though, you will use something in a way that is subtly wrong, so Macintosh Pascal doesn't catch you. Errors like that sometimes cause your program to **crash**, that is, to stop running .

- A **logical mistake** in program construction. These bugs are far more difficult to find, partly because Macintosh Pascal rarely catches them. How do you know you have a logical bug? Well, either the program simply doesn't work, or it doesn't work the way it is supposed to. Sometimes, a logical bug causes a crash, too. That might be because a mistake in the logic of your program makes some value get out of range, or makes something else illegal happen.

In summary, a syntax error is a mistake in the way some particular statement is written, while a logical error involves a mistake in the idea governing some sequence of statements.

Logical errors can be much harder to find, because there often is no single culprit that can be located. The best defence is good preparation: good program design helps prevent logical errors. Read the section of this chapter on programming to avoid bugs.

But, bugs do attack everyone on occasion. Macintosh Pascal is well suited to the task of finding and exterminating bugs—the process called **debugging**. The rest of this chapter explores the various techniques for debugging.

When the Macintosh Catches a Mistake

One of the problems that computers and humans have getting along is that computers do not tolerate mistakes, and humans make them all the time. Actually, human beings are very good at understanding things even when there are mistakes. That is what makes proofreading so difficult. For example:

April is said to be the
the cruelest month.

You have no trouble understanding that sentence, and may even have to read it twice to see the error. Pascal, though, could not accept:

```
if x > y then
then write('X is greater');
```

Computers have almost no tolerance for mistakes.

Macintosh Pascal checks for errors at two separate times:

1. When it reads something you've typed. When Macintosh Pascal reads what you've typed, which it does when you enter a Return, press the Enter key, or type a semicolon, it puts the words in proper format, puts reserved words in bold type, moves comments to the end of each line, and outlines anything it doesn't understand. You can correct whatever is wrong. Hit Enter to get Macintosh Pascal to read it, and see if the outlining remains. If it does, there is still something wrong.
2. When you run the program. Whether you give the Run or Step command, Pascal does a syntax check before running the program. (It also does a syntax check if you choose the check command.) If it finds an error, it stops checking and displays a message that attempts to describe the error. Pascal error messages in general are notoriously misleading. Macintosh Pascal error messages are pretty good, but missing semicolons can often mislead Macintosh Pascal. Check the lines around the one that Macintosh Pascal points at as having the error.

Also, the program stops checking when it finds one error. It may save you time if you look for similar mistakes after correcting one mistake.

Programming to Avoid Bugs

The best advice about bugs is like the best advice about colds: "Don't get them".

Good as that advice is, it is easier said than done. Bugs are like colds in that even with the best intentions and being as careful as possible, you still come down with one every now and then. Just as you can take steps to minimize the number of days you are sick, though, you can write your programs to minimize the amount of time you'll have to spend finding out why they don't work.

The trick is in avoiding the worst bugs.

The bugs that are hardest to find are logical, or design, mistakes.

You can write your programs in such a way that design bugs are less likely, and the others that occur are easier to find.

Here is the first rule of programming to avoid bugs:

Begin programming on paper.

The most common and most difficult to fix bugs come from poor program design. Good advance planning can save you time, work, and energy.

When you want to design a program, begin by stating clearly and simply what the program is going to do. Do that in general terms. For example, the purpose of the database program (the *Multiple Field Editor*) produced in this book is:

A program that creates and stores records constructed of editable fields.

Then break the task down into pieces. For example, for the same database program:

1. Get a file name.
2. Display screen.
3. Wait for command.
4. Do command.
5. Go back to step 3.

Continue this process of breaking the problem down into pieces, as if you were writing an outline for an essay. Eventually, when the pieces are small enough that each one represents a subprogram, or, better yet, each piece represents a statement, you can start actually writing the Pascal code.

Ideally, a single subprogram should do one kind of thing. If, for example, your program doesn't display things correctly on the screen, you only have to check the subprograms that write to the screen. If every subprogram does onscreen drawing, it will be much harder to find the culprit.

Ideally, your program should be written in independent modules. If you can, do this. It simplifies debugging if you get a small part of the program working correctly, and then add functions to it, testing along the way, until the entire program is finished. You should still write an initial plan that covers the whole program, and follow it as you add the pieces.

Finally, don't be afraid to rewrite. Writing a program is like writing anything: you often do a better job when you do it over. Treat your original program design with skepticism.

Finding Bugs

One of the great advantages of an interpreted language like Macintosh Pascal is that it is far easier and faster to find bugs than with a compiler. You can make changes in your program and immediately run it and see their effect.

Aside from that natural advantage of interpreters, Macintosh Pascal has a number of built-in debugging facilities.

Macintosh Pascal Debugging Aids

There are four debugging aids built into Macintosh Pascal. They are:

- Step and Step-Step.
- The Instant window.
- Stops.
- The Observe window.

The following sections discuss these features.

Stepping Through A Program

Aside from examining a printed copy of a program, the most useful debugging technique is to step through the program.

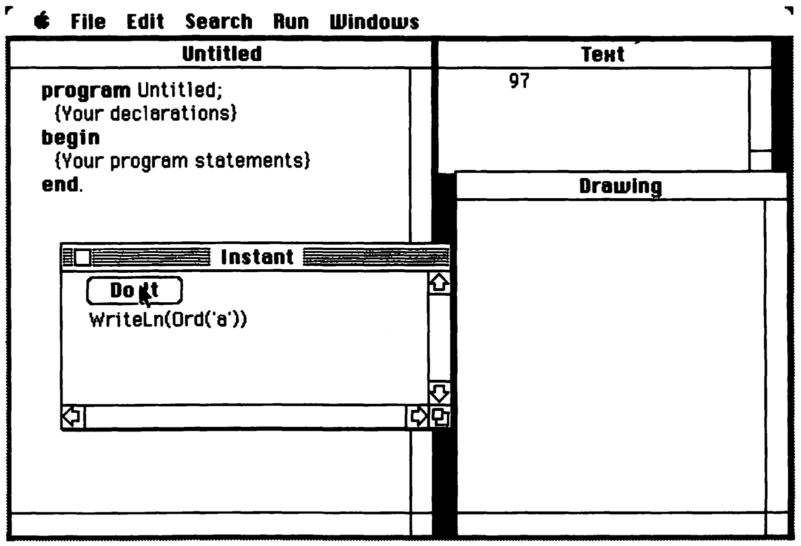
You were introduced to the Step command in Chapter 2 of this book.

You can run a program up to a certain place, halt it with the Halt command or with a Stop (discussed below), and then step through a suspect section. Very often, just seeing the flow of commands reveals the error.

The Instant Window

The **Instant window** allows you to run statements separately from your program any time a program is not running.

Choose **Instant** from the Windows menu to see the Instant window. It first appears as a small box, but, like most windows, you can expand it by grabbing the size control box in the lower right corner of the window. Figure 11-1 shows the Instant window. Notice that the Text window displays the result of a calculation from the Instant window.



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

```
WriteLn(Ord('a'))
```

```
97
```

Figure 11-1 Instant Window

You can type and edit statements in the Instant window just as you do in the programming window. When you click the mouse in the Do It button, the statements are run.

There is an important restriction for the Instant window, however: You *can't* have anything other than statements. In particular, you can't define variables, types, or named constants.

You *can* use functions, and expressions of any kind.

If you halt your program (using the Halt command in the Pause menu) you can use the Instant window to examine the values of variables, *and change them*. This is a very powerful debugging tool. If you think that your program's problems result from a variable having the wrong value, you can stop your program with the Halt command, change the value of the variable with the Instant window, and restart your program by choosing Go again.

To hide the Instant window, click in the Close box in the upper left corner of that window, or click in your programming window, which will bring the programming window to the front.

The Instant window isn't merely a debugging aid. You may often use it when writing programs, to check if you can use some statement in a certain way, or to find out the result of some operation, without having to run an entire program.

Stops

Look at the Run menu, as shown in Figure 11-2. The last command in the menu is the Stops In command.

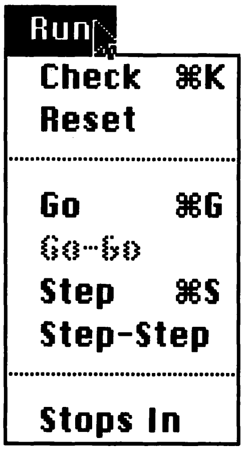
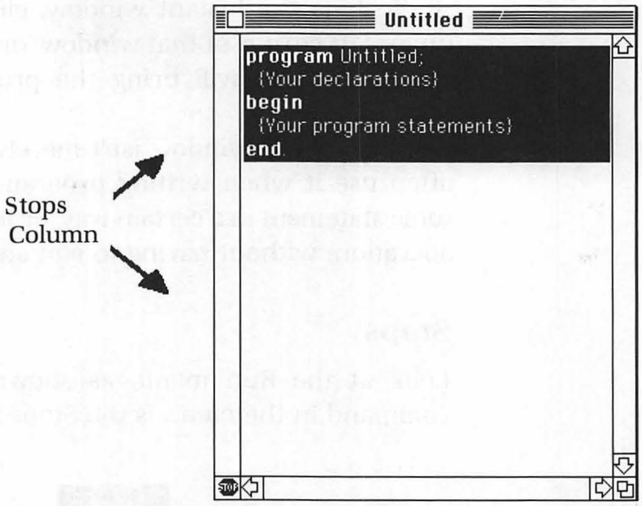


Figure 11-2 The Stops In Command

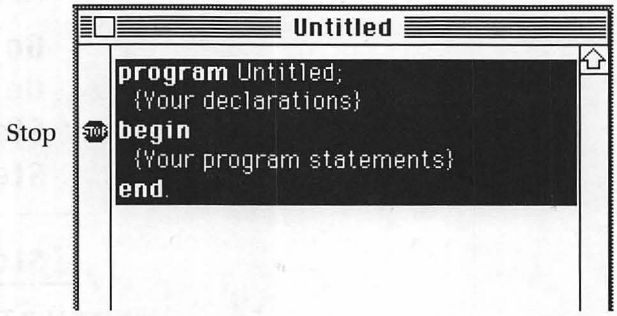
Choose the Stops In command.

A new column appears on the left side of the window, as shown in Figure 11-3. There is a small Stop sign at the bottom of the column. Move the mouse pointer into that column. A Stop sign follows the mouse. Move the mouse so that it is next to a program statement, and click the mouse button. When you move the mouse away, you will have left a Stop sign behind, as shown in Figure 11-4. If you choose Go, the program runs until it reaches the line with the Stop sign, and then halts. A small hand shows the statement where the program halted.



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

Figure 11-3 With Stops In



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

Figure 11-4 A Stop on the First Program Line

The effect of the Stop sign is the same as choosing the Halt command, except that the program stops at the place you've marked.

You can use Stops to stop the program at a place where you suspect there may be a problem. For example, if your program crashes (stops running) you can determine exactly where in your program the crash occurs with the use of Stops.

You can also use the Instant window to change or examine the value of a variable.

In addition, Stops are very useful with the Observe window, which is described below.

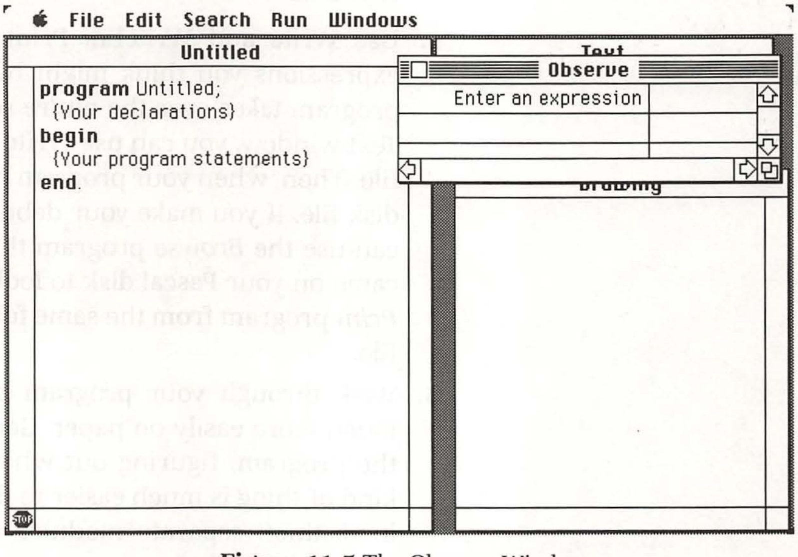
To remove a Stop, place the Stop sign cursor over the stop and click the button. The Stop is removed.

To remove the Stops column and all Stops, choose the Stops Out command from the Run menu. That command appears in place of Stops In whenever the Stops column is displayed.

The Observe Window

You can automate the process of checking the values of variables by using the Observe window.

Reveal the Observe window by choosing Observe from the Windows menu. The window is shown in Figure 11-5. You can type any expression in the right hand part of the window, where there is an insertion point blinking. The expression usually includes variables from your program. When you run your program, the expression is evaluated whenever your program halts.



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

Figure 11-5 The Observe Window

You can type a number of expressions in the right hand columns. Expand the Observer window as you expand any window.

You can halt the program with the Halt command, or you can use the Stops In command, and place Stops in your program.

If you want, you can run your program with the Go-Go command from the Run menu. If you do, the program pauses when it hits a Stop, updates any expressions in the Observe window, and continues.

When You Can't Use the Debugging Aids

Unfortunately, none of the debugging aids given above can be used with a program that is too large or that takes over the entire screen.

When you have a program like that, you are best off developing it in smaller modules that can be debugged with the aids Macintosh Pascal provides.

If you have no choice, see the following suggestions for help.

1. Use Write and WriteLn. Print the values of variables or expressions you think might be causing problems. If your program takes over the entire screen, and you can't see the Text window, you can use Write or Writeln to write to a disk file. Then, when your program finishes, you can examine the disk file. If you make your debugging file of type TEXT, you can use the *Browse* program that is in the Tools folder that came on your Pascal disk to look at the file. You can use the *Print* program from the same folder to print your debugging file.
2. Walk through your program by hand. This can be done much more easily on paper. Get a printout, and go through the program, figuring out what each statement does. This kind of thing is much easier to do if your program is written in distinct, separate modules and if the program is well commented. That way, you can decide which modules might be causing the problem, and which must be innocent. One piece of advice—be suspicious. Everyone overlooks things. Be particularly watchful for global variables that are accidentally modified where they should not be.

3. Redesign. Sorry, you may not consider this to be a very helpful suggestion. But, if you have a large program that you can't break into pieces and that just won't work right, go back to your original design and look for problems or, better yet, start all over again—assume you've learned from writing the first version and are now a better programmer.

General Debugging Hints

Here is some collected wisdom about bugs.

1. Bugs travel in groups. It is not unusual for one problem to result from several mistakes, each of which contributes to the problem. For that reason, it is better to go after bugs in a systematic rather than an ad hoc fashion: figure out whether or not something is wrong, don't just change things randomly and see if the program runs.
2. Bugs are easier to find in smaller programs. This may seem obvious, but it indicates a larger point: design your program to solve the entire problem, but implement your program in stages. Debug the smallest version that can run, then add another function, debug that version, and so on until you have the whole thing running correctly.
3. Many bugs result from the difference between variable and value parameters of subprograms. Be aware of the fact that when something is a value parameter, any changes made in the subprogram are not passed back to the main program. In an early version of the *Field Editor* program, the button RECTs, which were supposed to be passed from DrawButton back to the main program, were defined as value parameters in DrawButton. The Picked procedure therefore never reported a click in one of the buttons.
4. Be careful that you initialize variables before they are used in expressions. A variable has an undefined value before you assign something to it.
5. Be aware that all local variables are reallocated each time you call a subprogram. Don't count on values being where you left them.

6. Watch for boundary conditions. Problems tend to happen at the edges of things. Very often, you may think that something starts at 1, when it actually starts at 0, or the other way around.

Bugs in the Macintosh

Finally, although the possibility is relatively remote, you may run across a bug in Macintosh Pascal, or in the Macintosh system.

Simonoff's first rule:
"Finishing anything completely takes forever".
(That is, every program has at least one bug.)

If you find some feature of Pascal that just won't work the way it should, and you are sure your program is OK, try writing a small program that uses only that feature. It is easier to be sure there isn't some other problem when you write a small program. If the problem is still there, you may have found a bug in the Macintosh Pascal program.

Quick Summary

This chapter discusses how to find mistakes, called bugs, in programs. The following concepts, commands, and features of Macintosh Pascal were introduced.

Boundary conditions	are the conditions near the edges of your program's algorithm. For example, if your program is supposed to work on 0 to 1000 items, the boundary conditions would occur when you have no items and when you have 1000 items.
Bug	is a problem or mistake in a program. Anything that doesn't work right is a bug.
Crash	is when a program stops running because of some error, or sometimes because of some hardware problem.
Debugging	is the process of finding and fixing bugs.
Instant window	is the window that lets you enter programming statements for immediate execution. The statements are not connected to your

	program, except that you can alter the values of your program's variables.
Logical bug	is a mistake in the way a program is constructed. Even programs where all the programming statements are written correctly may have logical bugs.
Observe window	is the window that allows you to enter expressions which are evaluated whenever the current program pauses. See Run-Run in this chapter and Step-Step in Chapter 2.
Run-Run	is the menu command that runs a program so that the program pauses briefly at every Stop to update the Observe window. Execution continues once the Observe window is updated.
Stop	is a small symbol, a Stop sign, you can put next to any statement which makes the program stop when that statement is reached. When the Run-Run command is used, the program pauses to update the Observe window, and then continues immediately. You must use the Stops In menu command before you can enter Stops.
Stops In	see Stop.
Syntax error	is a mistake in the way a programming statement is used. This is similar to a grammatical mistake in speech or writing, except that, while people usually can figure out what you mean, a computer cannot.

APPENDIX

A

Everything About Semicolons

Many people find the use of semicolons in Pascal confusing. Don't worry about it; you'll get used to them. The programs in this book always include the semicolons in the places you need them. Besides, Macintosh Pascal will usually catch you if you make a mistake.

Actually, there are two rules governing the user of semicolons, and they both look simple:

- *A semicolon is required after any variable, type, constant, or label definition.*

Notes

Labels are declared symbols used for the Pascal **goto** statement. This book does not use labels or **goto**, because their use can make programs confusing.

Whenever you define any variable, type, label, or named constant, you end the definition with a semicolon. For example:

```
x : INTEGER;
y : BOOLEAN;
```

or:

```
writerType = (ELIOT, STEINBECK, JOYCE);
```

The only complication is when you define a record. In that case, every field definition is *separated* from the next field definition by a semicolon. Therefore, the last field definition does not have to be followed by a semicolon:

```
poem = record
  author : string;
  poem : array [1..LEN] of string
  end;
```

Putting the semicolon after the last field definition does not do any harm, however, and may prevent errors. (If you add another field, it is very easy to forget to add the additional semicolon.) For that reason, that extra semicolon is usually included in this book.

Notice that the entire record definition must be concluded with a semicolon, like any other definition.

- *A semicolon separates any two statements or parts of a program.*

The most important and easily ignored word in this definition is "separates." A semicolon does not mark the end or begining of a statement, instead it comes *between* two statements, parts, or declarations.

Here are the three types of constructs that must be separated by semicolons:

1. Any two **declarations** are separated by a semicolon. This is an example of a declaration separator:

```
procedure April;
begin
end; {This semicolon separates two procedure declarations.}
procedure Spring;
```

2. Any two **parts** of a program are separated by a semicolon.

   There are three program parts:

   - The program heading. This contains the word "**program**" and the program's name.
   - The declaration part, which can contain the **uses, label** (not used in this book), **const, type, var,** and subprogram declarations.
   - The statement part (that is, the main program).

   Here is the interface between two parts:

```
procedure Spring;
begin
end; {This semicolon marks the end of the declaration part.}
begin {main} {The beginning of the statement part.}
```

Notice that the semicolon isn't required after the procedure declaration, but it is required between the procedure declaration, which is the last declaration in the declaration part, and the **begin**. That distinction makes little difference in practice.

Figure A-1 has a diagram of the parts of a program. A subprogram has a similar structure.

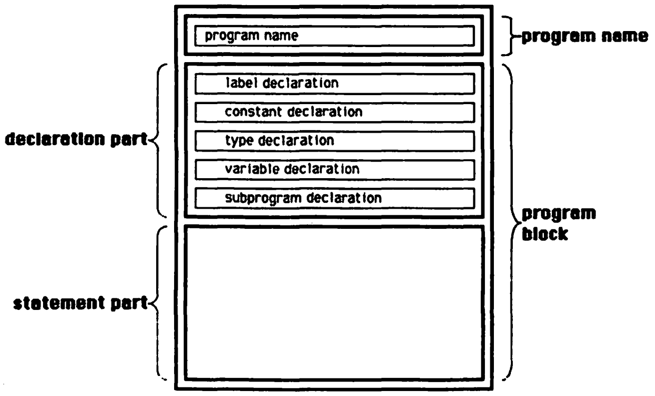


Figure A-1 Overall Program Structure

Sections of a program can be members of several parts; the **var** part of a procedure, for example, is:

- Part of the declaration part of the procedure.
- Part of the body of the procedure, called the procedure's block.
- Part of the declaration part of the main program.
- Part of the block of the main program.

3. You must have a semicolon between two consecutive statements. Here is a simple case:

```
x:=100;
y:=50
```

Statements can be contained within other statements, and you still have to have separators between the contained statements:

```
if Button then
begin
x:=100;
y:=50
end
```

Notice that there is no semicolon between **begin** and the statement following it, or between **end** and the statement preceding it. Neither **begin** nor **end** is a statement. This is also true of **repeat, of, do, else,** and **then,** which are parts of the **repeat/until, case, while, for, with,** and **if** statements. Because those reserved words are parts of statements, they are never followed by a semicolon, although they all are usually followed by a statement. The statement following the **repeat, of, do, else,** or **then** is part of the **repeat/until, case, while, with, for,** or **if** statement.

end, however, often marks the end of a compound statement, and hence is usually followed by a semicolon. You do not have to put a semicolon before any **end**. Although you must put semicolons between any two consecutive statements, you do not need semicolons between a statement and one that encloses it, because that statement is actually part of the enclosing statement. Any particular program line can be part of several statements, as well as being a statement in itself.

In general, if you follow **begin, repeat, of, do, else,** or **then** with a semicolon, or give a semicolon before an end, you create an **empty statement**. In the cases of **do, of, else,** and **then** the empty statement is definitely unwanted, because it displaces the statement that should be there. For example:

```
if condition then
  ;
```

There is nothing illegal about that statement. However, when the condition is TRUE, nothing special is done, because the statement following the **then** is empty.

Another important point is that **do, of, else,** and **then** cannot be preceded by semicolons. In all cases, you would have a statement separator in the middle of a statement, which is clearly illegal.

Although you do not have to put a semicolon before any **end**, empty statements before an **end** rarely have any bad effects. In fact, this book usually has semicolons before an end because doing so prevents errors; there is a tendency to forget to add the semicolon if you add another statement before the **end**.

APPENDIX

B

Help, Please, My Mac is Burning

General Advice

1. Install the Programmer's Switch that came with your Mac. That is a small, gray piece of plastic. You install it by inserting it in the second slot from the bottom, all the way toward the back on the left side of your Mac. (See the Macintosh manual for more information.) To reboot, rather than using the power switch, press the front part of the Programmer's Switch. (*Very* rarely, a problem occurs which is not cleared up by rebooting with the Programmer's Switch, but it is by rebooting with the On/off Switch.)
2. If you try the methods in the appendix and can't fix your problem, you may have a hardware problem—something wrong with the Mac itself. Talk to your computer store.

Disk Problems

Disk insertion causes system crash

1. There is a file on each disk which records information about the disk. That file can get damaged if a disk is not ejected properly. Fortunately, the file can be rebuilt. Hold down the Command and Option keys while inserting the disk. You may have to try this several times. If that doesn't work, turn off your Mac and boot it with the bad disk (even if the disk is not bootable), holding down the Command and Option keys while the disk starts booting. Either of these methods destroys all folders, but your files should be unharmed. Unfortunately, these methods do not always work.
2. If the disk contains documents that you need to save, run the application that uses those documents, close the Untitled document that first opens up, and choose the Open command from the File menu. When the dialog box that lets you choose from files on the disk shows, eject that disk and put in the one that gives you trouble. If you are in luck, you will be able to see your documents. Open one of them, and use Save As... to save that file on another disk. Repeat for each document you need to save.

Attempting to boot bootable disk causes unhappy Mac icon

1. The disk may not actually be bootable. Boot a disk that you know is OK, then eject that disk, and insert the suspect one. If that causes a system crash, see the next section. Otherwise, check if it has system files on it.
2. There is a file on each disk which records information about the disk. That file can get damaged if a disk is not ejected properly. Fortunately, the file can be rebuilt. Hold down the Command and Option keys while booting with the disk. You may have to try this several times. This method destroys all folders, but your files should be unharmed. This method doesn't always work, unfortunately.

Disk drive keeps running and won't stop

1. Wait patiently for a while—a half hour, for example. If it still hasn't stopped press the rear part of the Programmer's Switch to cause an error, which stops the drive, and then press the front part of the switch to reboot. Otherwise, turn off your Mac, and turn it back on. Rebooting should eliminate the problem.

Can't eject disk

1. Is the disk selected when you choose Eject? The disk is selected if its image is darkened.
2. If the Macintosh system is running, press Command + Shift + 1 to eject the internal disk; Command + shift + 2 ejects the external disk.
3. Reboot, either with the Programmer's Switch or with the On/off Switch, while holding down the mouse button. The disk should pop out.
4. Unbend a paper clip and poke one end firmly into the hole just outside the lower right corner of the disk slot.

Display Problems

Screen has strange lines across it

1. Stop the running program if you can. In any case, turn your Mac off, and turn it on again, or reboot with the Programmer's switch. Rebooting should eliminate the problem.

Screen stays dark

1. Try turning the brightness knob, which is under the left side of the Mac, above the little sun symbol.
2. Check power cord and plug.

Printing Problems

Not enough room to print

1. A disk needs as much as 34K of free space to print a file. Try removing some documents from the disk. (If you have a two-disk system, the one with the application is the one that needs the extra space.)

Can't print

1. Check cables. You should tighten the screws at the ends of the cable, so the cable is firmly connected to the printer and the Macintosh. (You need a small screwdriver to do this properly on the Imagewriter side of the cable.)
2. Check printer lights. Assuming you are using an Imagewriter, there should be two green lights. If there is only one, press the Select button on the Imagewriter, and try printing again. If there are no lights, check the power switch and the power cord.
3. If running on a Macintosh XL (or Lisa), make sure you are plugged into the right port. That is the port all the way to the right, looking at the back of the XL. Also, if you are using an ordinary RS-232 cable, you may need a modem eliminator (sometimes called a null modem) to make the connection properly. If you are using a parallel printer (the Imagewriter is a serial printer, but some older Apple printers are parallel) you need to run the Install Parallel Printer application to use the parallel printer with the Macintosh Pascal disk. That application is on your Macintosh XL system disk.
4. Depending on what program is trying to print, you may need an Imagewriter file on the disk that contains the application. Check in the System Folder to see if there is one. If not, copy one from another disk.

Every page starts at the wrong place

1. Use the knob on the side of the printer to position the paper to the top of the page. Press the Select button on the printer to deselect the printer. (If an Imagewriter, the light next to

the Select button should turn off.) Press the Form Feed button. The printer should stop at the top of the next page. If it doesn't, turn the knob to position the paper, and try again.

2. Turn the printer off, and turn it on again.
3. Turn your Mac off (eject any disk first) and boot it again.

Printing looks wrong

1. Turn the printer off, and turn it on again.
2. Turn your Mac off (eject any disk first) and boot it again.

Draft Mode Doesn't Work Right

1. I don't think it does, either. Try using Standard mode. It is nearly as fast, and looks much better, anyway.

Memory Problems

Out of Memory Error in Pascal

1. Hide the programming window by clicking in the window's Close box. That helps you when there is not enough memory to reach a HideAll call in your program.
2. Look for sections of code that can be replaced with calls to predefined procedures.
3. Look for identical sections of code and replace them with procedure or function calls.
4. Avoid using QuickDraw2 or, especially, SANE. Both take up quite a bit of space.
5. Use events in your program. If you do so, Macintosh Pascal drops some of the code it uses to handle events, giving you more room.
6. Shorten the names of variables that are used often.
7. Shorten the names of subprograms that are called often.
8. Eliminate subprograms that are only called once.
9. Try to reuse pointers and handles, instead of reallocating them. The Dispose and DisposeHandle procedures do not properly free space in Release Version 1.0.

Mouse Problems

Mouse pointer doesn't move

1. Check cable. Unscrew it, and screw it back in carefully and evenly.
2. Try cleaning the mouse. (See the Macintosh manual.)
3. Reboot. The problem may be related to insufficient electrical power. Try isolating the Mac as much as possible, so that as few other appliances as possible are on the same electrical line.

Mouse only moves in one direction (usually up and down)

1. The cable is probably not fully in its socket. Unscrew the connection on the back of your Mac, and screw it back in carefully and evenly.Other Nothing seems to be happening 1. Try holding the Command key down and pressing the period. This cancels many actions.
2. Check the mouse plug. If the mouse won't move, try giving a menu command with a key combination.
3. Wait a few minutes.
4. Turn off your Mac and reboot.

Burning Macintosh

1. Pull the plug. Use a fire extinguisher to put out the fire.
2. If no extinguisher is available, try baking soda.
3. Use water as a last resort.

Index

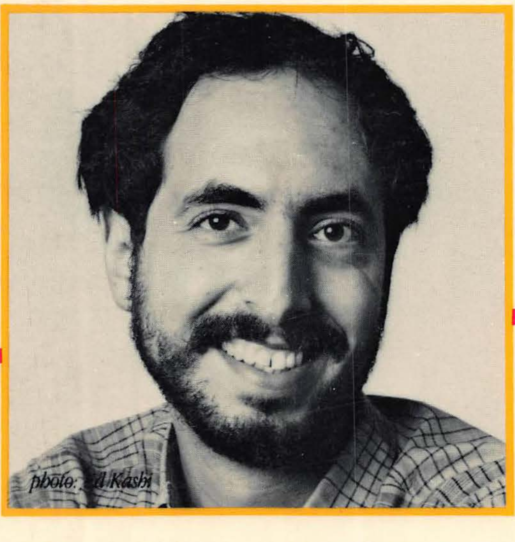
6562-0

Introduction to Macintosh™ Pascal

Learn to create and design Macintosh Pascal programs easily and instantly. Build your programming skills with practical desktop applications such as creating an address book and telephone dialer.

Introduction to Macintosh Pascal, by Apple documentation writer and Pascal expert Jonathan D. Simonoff, emphasizes the complete capabilities of Pascal for the Macintosh, beginning with a solid foundation of program design concepts, and proceeding through graphics, text editing and files, handling sound, and using math operators and functions. Featuring many program examples and screen illustrations, the book covers programming and line drawing; using subprograms, procedures, and functions; using the mouse as an input device; exploring the capabilities of QuickDraw; editing text; handling files; generating sound and music synthesis programs, calculating standard functions and using Pascal Macintosh extensions; and debugging programs.

Introduction to Macintosh Pascal is one of several Apple Press titles in the popular Hayden Macintosh Library, which also includes *Macintosh Revealed, Volumes One and Two.*



About the Author...
Jonathan Simonoff has been affiliated with the Macintosh and Lisa projects at Apple Computer Inc., writing manuals on how to program computers. He has worked as a programmer and technical writing consultant for many computer companies and institutions, including Wang Laboratories, Apollo Computers, Fortune Systems, and the Massachusetts Institute of Technology. A graduate of M.I.T., Simonoff is a principal of Ink Company, a San Francisco-based technical publications company.

Hayden Book Company
A DIVISION OF HAYDEN PUBLISHING COMPANY, INC.
HASBROUCK HEIGHTS, NEW JERSEY / BERKELEY, CALIFORNIA

ISBN 0-8104-6562-0