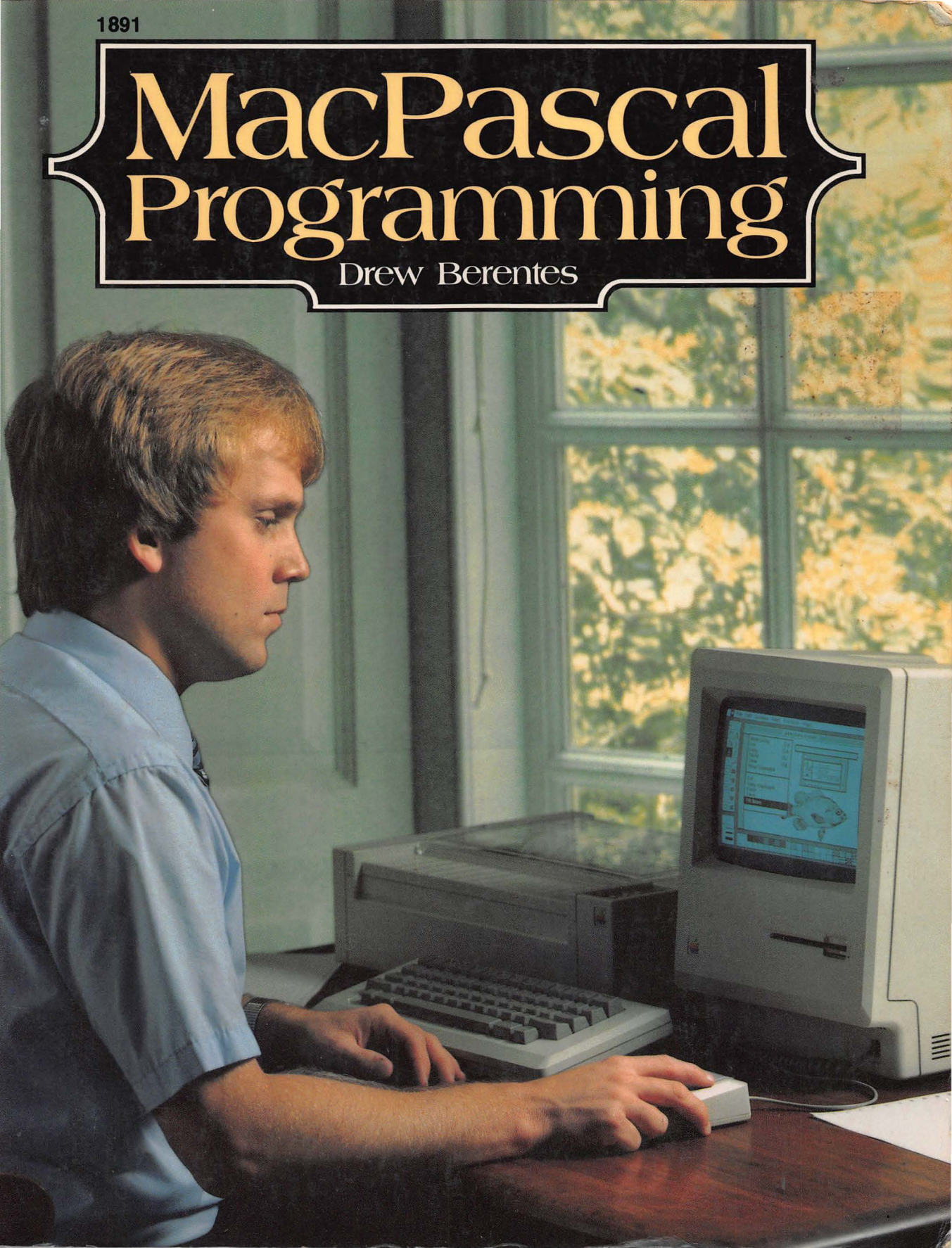
1891

MacPascal Programming

Drew Berentes



MacPascal Programming

Drew Berentes

TAB BOOKS Inc.
Blue Ridge Summit, PA 17214

For Woody

FIRST EDITION

FIRST PRINTING

Library of Congress Cataloging in Publication Data

Berentes, Drew.
MacPascal programming.

Includes index.
1. Macintosh (Computer)—Programming. 2. PASCAL (Computer program language) I. Title.
QA76.8.M3B47 1985 001.64'2 85-9740
ISBN 0-8306-0891-5
ISBN 0-8306-1891-0 (pbk.)

Cover photograph by the Ziegler Photography Studio of Waynesboro, PA.

Contents

Acknowledgments

This book could not have been written without the support of several persons.

Kevin Jones at Apple Computer graciously supported my efforts, making software and manuals available.

Kevin Burton at TAB BOOKS Inc. provided constant support during the genesis of this book. I hope that I have not driven him crazy with my many questions and problems. He has dealt with them all promptly, effectively, and sympathetically. I look forward to working with Kevin in the future.

To a special, seemingly unflappable person, to Marilyn Johnson at TAB my deepest thanks. Her efforts in editing my last book went above and beyond the call of duty. I can only imagine the frustration of dealing with a nervous and demanding author under the necessity of meeting a production deadline, but through all of the confusion and pressure, Marilyn was nothing but diligent and helpful.

Finally, to Lydia Berentes, my gratitude for her hours spent in proofreading and for her loving support. No one knows how much work goes into a book as well as the person who lives with the author, and no one must have greater patience.

Introduction

For me computer programming is one of the most fascinating activities that is available in the modern world. It is never ending in its complexity—one can never learn everything about it—but it is almost unlimited in the sense of accomplishment it can inspire. Through programming we learn an approach to problem solving that has broad applications in many other areas of our lives. Through programming we can create tools that solve complex problems, games of great challenge, and graphic displays of complex beauty.

Programming is a powerful incentive to learning. Mathematical concepts that may have bored you in the past take on new vitality when they are seen working in a program. And yet, extensive knowledge of mathematics is in no way a prerequisite to programming. The simple fact is that the desire to do something is a strong motivation towards learning how to do it. If you want to animate a figure in a computer game, you will be inclined to learn the mathematics and techniques of simulating motion. And you will have fun along the way.

Mathematics is not the only area in which this phenomenon takes place. If you want to write a program that creates poetry, you must teach the computer to construct proper sentences. The rules of grammar that seem abstract and dull in school take on new meaning since you need them to solve your problem. If you need an accounting program, you must learn accounting procedures.

Pascal is a particularly suitable language with which you can learn programming. Most modern programming concepts are available in Pascal, in sharp contrast to BASIC, perhaps the most popular beginning language. Although it is easy to get started in BASIC, things become much more complex as programs grow in size. Pascal programs, by contrast, expand gracefully. You will find that the largest program in this book is little more difficult to understand than many of the smaller ones.

The concepts you learn with Pascal are applicable in almost any programing environment. Once

thoroughly grounded in Pascal you should be able to move readily to other languages. Also, Pascal is more consistent than BASIC when moving from one computer to another. Regrettably, different versions of Pascal abound, but the uniformity of Pascal is still fairly high.

I have wanted to write a book about Pascal for some time. My particular bias in writing computer texts is to provide strong support for beginning programmers who are teaching themselves the art. In this respect, Macintosh Pascal was worth waiting for. MacPascal is by far the most supportive environment I have found for learning Pascal. In fact, it is one of the best for any implementation of any language. It provides numerous programmers tools unavailable in any other Pascal package of which I am aware, and it has allowed me to introduce you to Pascal in small, manageable steps. Thanks to the special features of MacPascal, you will be able to do interesting things in Pascal in the first chapter.

MacPascal takes full advantage of Macintosh user interface, which is presently the best in the microcomputer business. Much less time is required to learn the essentials of Macintosh operation than to learn them for any other computer I can think of.

This book has two major objectives:

- To help you teach yourself the basics of Pascal programming
- To introduce you in detail to the use of Macintosh Pascal.

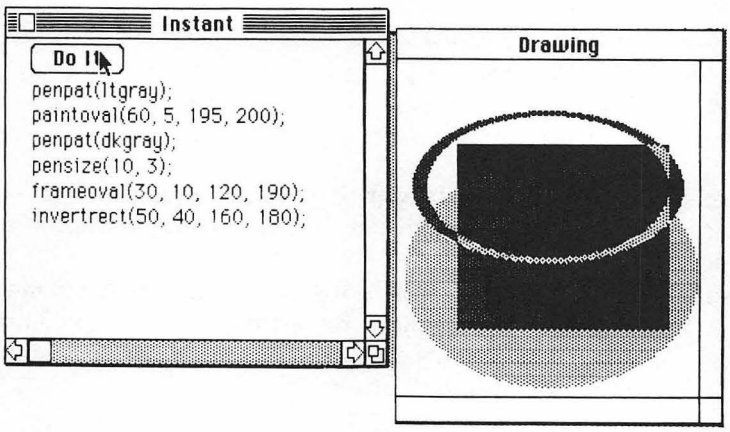
Many activities that will get you working with your Mac, not just reading about it, are included. Through these activities you will use most of the MacPascal programmer's support tools as well as enough features of Pascal to enable you to create interesting programs of many different sorts. If you do everything I suggest, I am sure that you will learn a lot from the book.

The going will not always be easy. I will not issue idle promises about this book making programming a task to be learned with elementary ease. If programming could be made easy, would we need all of the introductory programming texts that have flooded the book market in recent years? As I noted earlier, the complexity of computer programming continues without end. And, unfortunately, many of the mental skills programming requires must be learned through hard work and practice. But nothing worthwhile is learned without some work. That is why athletes, musicians, and programmers must work hard to become great.

However, programming is also a skill that anyone can learn. Elementary school children have become competent programmers. I firmly believe that anyone can learn to program and can find reward in the process.

With those remarks, I invite you to embark on a programming adventure. Above all, have fun. In the process, you may be amazed at how much you will learn.

Chapter 1



Introducing Macintosh Pascal: Pascal Procedure Statements

In this first chapter we will jump right in and look at Pascal procedure statements. These are the simplest structures in Pascal, and every program is built by carefully combining them to perform complex tasks.

In the process of examining procedure statements, we will take a first look at the Macintosh Pascal programming environment, which is extremely powerful and supportive. Before we can begin, however, I have to make sure that you have acquired some basic skills in using the Macintosh.

TOPICS COVERED IN THIS CHAPTER

- Selecting options from the Macintosh pulldown menus
- Typing
- Selecting text for replacement
- Starting up Macintosh Pascal
- Using the MacPascal Instant, Text, and Drawing windows
- Executing procedure statements in the Instant window
- Using proper syntax for WRITELN statements
- String expressions
- Using semicolons to separate Pascal statements
- Numeric expressions and operations
- Using field parameters in WRITELN statements
- Pascal functions
- Generating a musical note
- Drawing several MacPascal graphics shapes
- Ending a Macintosh Pascal session

THINGS YOU WILL NEED

You can do everything in this book using only these four items:

- A Macintosh computer (128K or 512K)
- A Macintosh Pascal program disk,
- One or more blank storage disks
- This book.

Two additional items may be helpful. An Imagewriter printer will allow you to print out programs and other items associated with your Pascal activities. It is often easier to work with large programs if they are printed out.

An auxiliary disk drive is a very convenient thing to have because it makes it much easier to copy things between disks. The Pascal disk has very little free space on it. You will eventually find yourself needing to store programs on other disks. This can be done using only the disk drive built into your Mac, but copying is much easier with two disk drives.

BEFORE YOU BEGIN

After this section, I am going to assume that you know a thing or two about using the Macintosh. If you are at all experienced with the Mac, you probably know everything you need to know to keep up with me.

If you are a new user, however, you should spend some time with your Macintosh manual. It won't take you long to read the sections I suggest. The lessons are excellently presented and can be read very quickly. While you do not have to be familiar with every detail about Macintosh operation, the more you know the better.

Before you continue, I recommend that you go through the following chapters in your Macintosh reference:

1. Learning Macintosh
2. Finding Out More About Macintosh
3. Using the Finder

You should be sure that you become familiar with these topics:

- Starting up the Macintosh
- Mouse operations: pointing, clicking, double clicking, dragging, etc.
- Selecting and manipulating icons
- Using the pulldown menus
- Opening windows
- Using scroll bars in windows
- Moving and changing the size of windows
- Starting Macintosh applications
- Using folders
- Copying, renaming, moving, and removing documents, folders, and applications.

Don't read for detail right now. Just go for the main features. I am going to prompt you a bit dur-

ing these early chapters. However, you should plan on returning to the manual at regular intervals. For awhile, every reading will probably reveal a new feature that will solve a problem you have encountered. The Macintosh is very simple to use, but that does not mean that you can learn everything about it all at once.

There is one other thing you need to pay particular attention to: text editing. The process of programming computers involves quite a bit of this, and the more comfortable you are with editing, the easier you will find the activities in this book. If you have used MacWrite, you will have no trouble editing Macintosh Pascal programs. If not, the following introduction will prove helpful.

EDITING TEXT ON THE MACINTOSH

One nice feature of the Macintosh is that most editors work in about the same way. Several basic editing operations may be illustrated through use of the Macintosh Note Pad. Please perform the following exercises unless you are familiar with Macintosh editing. Be sure to perform all of the steps exactly as described. Each task is marked with a ■ to make it easy to spot.

■ Start your Macintosh by inserting the Macintosh Pascal disk into the disk drive until it snaps into place. It will only go in one way. Then turn on the power to your Macintosh. Soon the screen in Fig. 1-1 will be displayed.

The small picture in the upper right corner of the screen is called an *icon*. An icon is a symbolic picture. This icon is shaped like a Macintosh storage disk and represents the Macintosh Pascal disk.

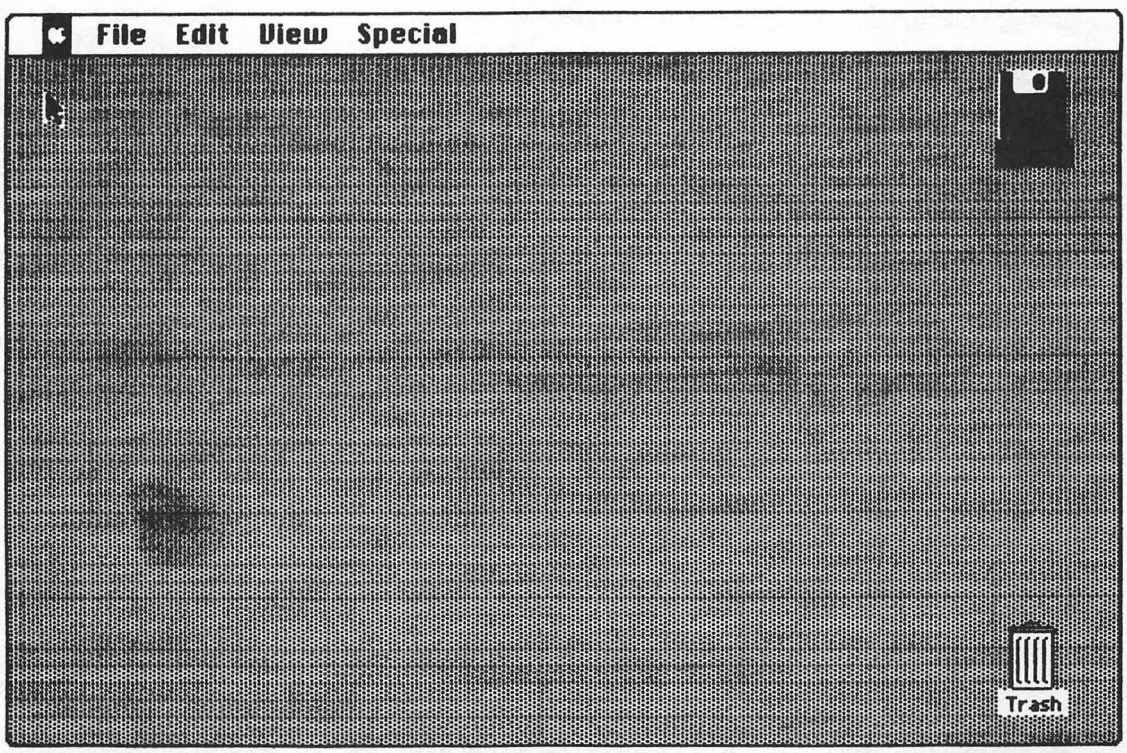


Fig. 1-1. The Macintosh Pascal startup screen.

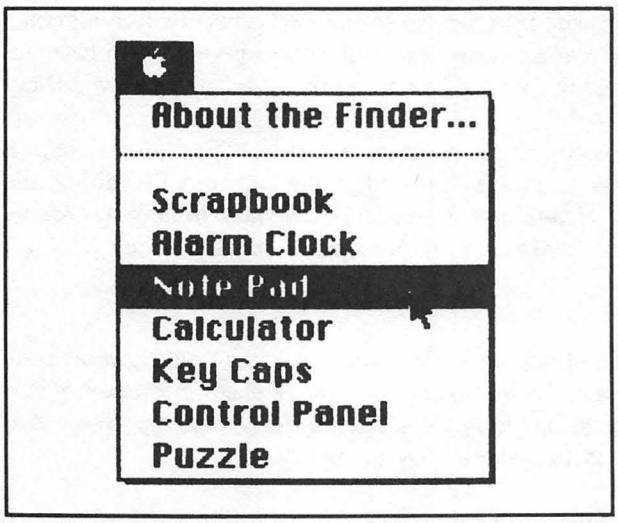


Fig. 1-2. The Macintosh Desk Accessories Menu.

The items at the top of the screen are the headings of Macintosh pulldown menus. Many Mac features are accessed from these menus. You can learn all about the pulldown menus in the Macintosh reference. We will examine the menus that are important in Pascal as they are needed.

■ Open the Note Pad by choosing it from the Desk Accessories menu. To choose an item from a Macintosh menu, point the mouse pointer to the menu item at the top of the screen and press and hold down the mouse button. A list of options will appear. Open the Desk Accessories Menu by pointing to the apple in the upper left corner of the screen and holding down the mouse button.

■ To choose the Note pad, pull the mouse down the menu until the Note Pad line is shown in inverse type. Your menu will look like the one in Fig. 1-2.

■ Finally, with Note Pad shown in inverse type, release the mouse button. After some disk activity, a note pad will appear on the screen.

The note pad has a blinking bar in it. When you type text, it will be placed at the location, which is called the *insertion point,* shown by the blinking bar.

■ Type this: "A rose is a flower." If you make a mistake, press the Backspace key to erase your error. When you are through, your note pad will look like the one in Fig. 1-3.

That was easy enough, and it would be enough if you never needed to change what you type. However, you will make mistakes, and you will make changes, so you need to know something about changing what you type.

The first editing technique involves inserting text. Text is always typed at the location of the insertion pointer, so you must learn how to move the pointer around.

■ Move the mouse around and observe its pointer, which is called the mouse *cursor.* Outside of the Note Pad, the mouse cursor is an arrow. Inside the Note Pad, it takes on a different shape, a vertical line with tails at top and bottom. This new cursor is used to determine where typing and changes to typing will take place.

■ Position the mouse cursor just before the "f" in the word "flower." Then click the mouse button. Notice that the insertion point moves to the new location. You are now ready to insert some text.

■ Type "very pretty," noticing that it is inserted into the text without replacing anything. If you did not position the pointer just before the "f" you may have to type a space before "very."

To replace text, the old text must first be *selected.* To select a word, position the mouse pointer anywhere inside of the word. Then click the button twice quickly.

■ Point the mouse at the word "flower." Then click the mouse button twice. If the letters turn white on a black background, the word has been selected. If nothing happens, you did not double-click fast enough. Try again.

■ Now, anything you type will replace the selected text. Type "rose". When you type the first character, notice that "flower" disappears, replaced by an "r".

To select more than one word, follow these steps:

1. Position the mouse cursor at the one end of the text to be selected,
2. Click the mouse button to move the pointer to that location,
3. Move the mouse to the other end of the text you wish to select
4. Hold down the Shift key and click the mouse again.

Using this technique any size block of text can be selected.

■ Select "very pretty flower". Position the mouse cursor before the "v" in the word "very." Click the mouse button. Then position the mouse cursor after the "r" in the word "flower". Finally, hold the Shift key down and click the mouse button. The colors of the text of the entire phrase will be reversed indicating that it has been selected.

Alternately, you can select a block of text by dragging the mouse across it:

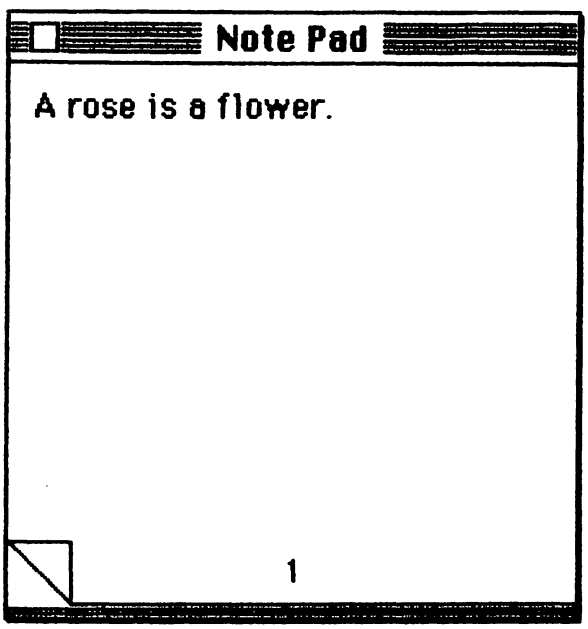


Fig. 1-3. The Macintosh Note Pad.

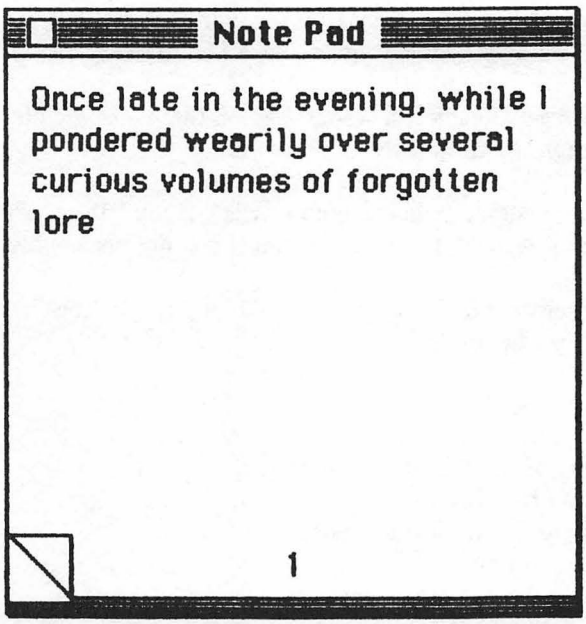


Fig. 1-4. Practice editing by typing this text.

1. Place the mouse pointer at one end of the block of text.
2. Press and hold the mouse button.
3. Move the mouse to the other end of the block.
4. Release the button. The entire block will be selected.

I generally prefer the Shift key method, but dragging has its advantages in certain situations.

Any text that is printed in white on black is selected. Be sure this is text you wish to change, since it will be deleted if any character is typed. If you wish to cancel a selection, position the mouse pointer anywhere in the text and click the button once.

■ Delete this text without replacing it. Press the Backspace key. The selected text will disappear and nothing will take its place.

■ Now type "rose is a" and notice that it is inserted where the old text was.

When you are typing something that occupies multiple lines, you can start a new line by pressing the Return key. Pressing the Return key with the insertion point in the middle of a line will break the line in half, placing the second half on a new line. The Return key places an invisible *end-of-line* marker into the text.

■ Using the mouse, place the insertion point after the second "rose" and press the Return key. The line will be broken.

■ To rejoin the two lines, position the insertion point at the beginning of the second line. Then press the Backspace key. This will delete the end-of-line marker at the end of the first line, instructing the Mac to display the two lines as one.

You now have performed all the basic editing functions. They are summarized below.

- To enter and display text simply type it. Your text will always be placed at the insertion point.
- To erase text to the left of the insertion point, press the Backspace key.

- To replace other text first select the old text, and then type the new. The old text will be erased and the new will take its place.
- To erase text, select the text to be erased and press the Backspace key.
- To start a new line, press the Return key.
- To split a line, place the insertion point at the desired location and press Return.
- To join two lines, place the insertion point at the beginning of the second line and press the Backspace key.

Take some time to practice editing. Type the text in Fig. 1-4. Then edit it until it looks like the screen in Fig. 1-5. In the process you will have a chance to practice most of the editing techniques that have been introduced. When you feel comfortable with editing, you can start in with Pascal.

SOME PROGRAMMING CONCEPTS

Now that you know your way around a Macintosh, you can begin to look at some Pascal programming concepts. Computer programming is the process of assembling the instructions that tell a computer how to perform a given task. You will discover that these instructions are very explicit and break a large task down into quite small pieces.

At first, the level of detail involved in programming may seem excessive. However, several of your day-to-day activities involve almost the same sort of detailed analysis. Consider the rules in a card game. A good set of rules will never leave any doubt concerning the steps involved in playing the game, the resolution of conflicts within the game, or the determination of a winner. The winner of most card games is not declared by somebody's opinion, but by rules that make the result clear and incontrovertible. Anything else leads to confusion.

Another common situation involving detailed analysis involves giving traffic directions. How many times have you had difficulty because someone gave you incorrect driving instructions?

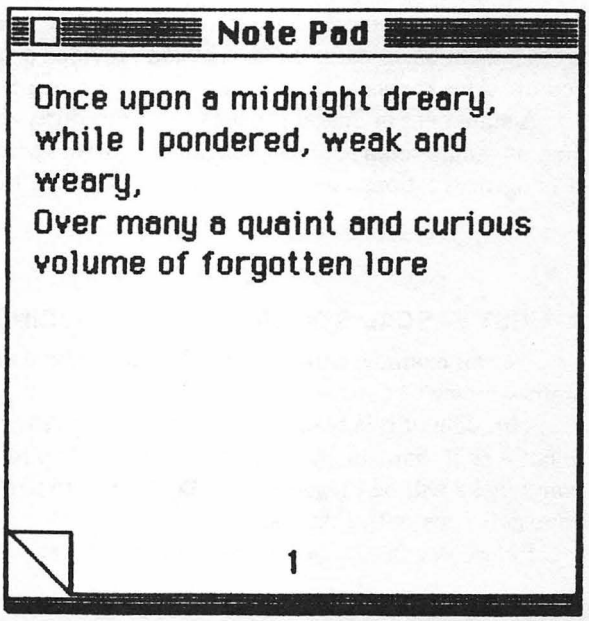


Fig. 1-5. Edit the text of Fig. 1-4 so that it looks like this.

The writing of good traffic directions can be used to illustrate many of the principles of computer programming. When you start to write a set of directions, you must break the route down into small sections and analyze what the driver must do to traverse each section. In other words, the large problem of getting from A to Z must be broken down into a number of smaller problems.

This analysis can stop when you have written simple statements, such as, "Drive north on Main Street for five blocks." Provided the blocks on Main Street are easy to distinguish, this would constitute a fairly good instruction. Another instruction might be, "Turn right onto Grant." This, too, is all right, as long as the driver will not see signs for both Grant Street and Grant Place.

However, unless the available choices were limited, it usually would not be adequate to simply say, "Turn at the light." Equally bad would be the direction, "Drive south four blocks," when you have miscounted and want the person to drive five blocks; or, "Turn right on Park Place," when the sign for Park Place is missing.

Do I make my point? It is very easy to write bad directions, but it can be rather difficult to write good ones. This is especially the case when working with computers. You can use your knowledge of streets to interpret bad directions. Computers cannot make such interpretations. They will always do exactly what you tell them to do, even if what you told them isn't what you thought you meant.

Let's look a little more closely at our typical driving instructions. At the very least, every instruction contains a command word or phrase, such as "turn" or "drive." In Pascal, commands are called *procedures.*

In addition, most instructions contain some information that modifies the command. The instruction "turn" carries with it the question "which direction?" So we might add the word "left" to a turn instruction, making the entire instruction phrase "turn left." If the command is "drive," we might add the phrase "three blocks." The word "left" and the phrase "three blocks" modify the commands they are associated with. Also, if properly written, they make the command unambiguous.

Phrases that modify procedures are called *parameters* in Pascal. Not all procedures require parameters, but most do.

The modifying phrase itself is called an *expression.* An expression can be simple, for example, "right" in the statement "Turn right." Or it can be complex, such as "two miles past the second railroad crossing." In both cases, the expression provided a value that could be used as a parameter for a procedure. An expression acts as a parameter when it is used to modify the action of a procedure.

A statement in Pascal consists of a procedure, along with any parameters that might be required. Just as simple instructions ("turn left" "drive three blocks") are used to build up complete sets of driving instructions, simple statements are used to build up Pascal programs.

A FIRST PASCAL STATEMENT: INTRODUCING THE WRITELN PROCEDURE

We can examine some Pascal statements by using a special feature of Macintosh Pascal, the Instant window.

Throughout this book, I will be asking you to perform actions on your Macintosh. Programming is not a skill that can be learned by reading. You must do. You must experiment. To emphasize this, many tasks will be tagged with a ■. Failure to perform these tasks will greatly reduce the value of your activities with this book.

Before beginning, you must start up Pascal.

■ Insert the Macintosh disk in the disk drive and turn on your Macintosh.

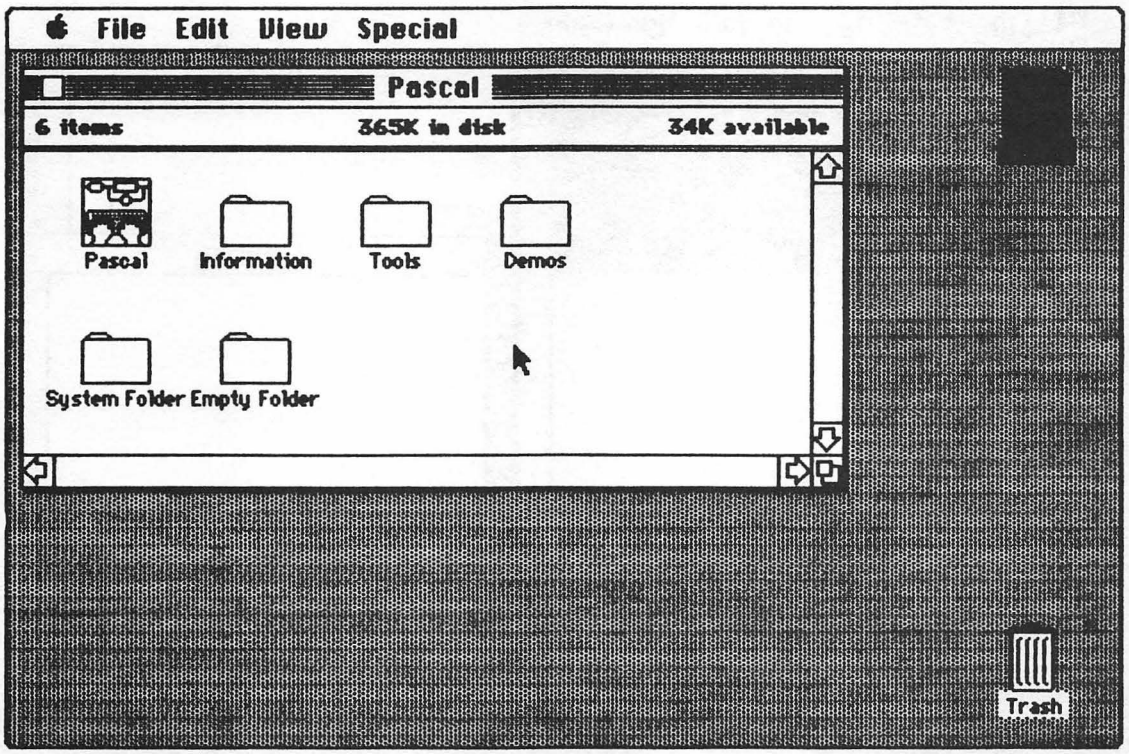


Fig. 1-6. The Macintosh screen after the Pascal disk icon is opened.

■ Open the Pascal window by pointing the mouse to the Pascal disk icon and and double-clicking the button. The screen shown in Fig. 1-6 will appear.

The new large box is a Macintosh window. It contains several new icons. The one we are interested in is labeled "Pascal." This is the Macintosh Pascal application icon. It is used to start Pascal.

■ Locate and double-click the Pascal application icon. The Pascal startup screen will appear, as shown in Fig. 1-7.

The Macintosh is oriented around windows. This screen contains three windows: Program (marked "Untitled"), Text, and Drawing. At any one time, only one window is *active.* This is the window that has the lines on either side of the title. In Fig. 1-7, the Untitled window is active.

There are several manipulations that may be performed on an active window. Let's examine the methods used to move windows and change their sizes. First, let's remove the Program and Drawing windows from the screen by *closing* them.

Figure 1-8 illustrates a typical window, labeling a few of its features. Identify the box in the upper left corner of the window. To close a window, first make the window active by clicking the mouse anywhere inside of the window.

■ Click several of the windows in turn. Notice the changes that take place.

■ Close the Untitled window by making it active and then clicking the little square in its upper-left corner.

■ Close the Drawing window following the same procedure.

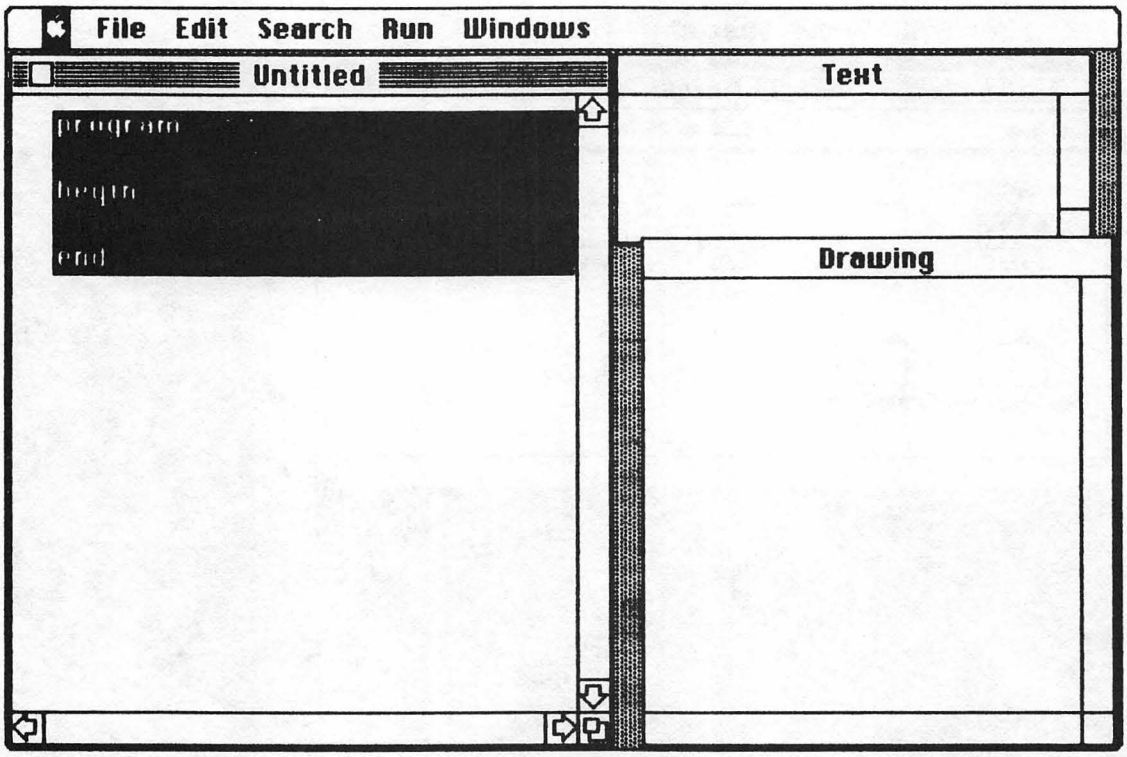


Fig. 1-7. The initial Macintosh Pascal screen.

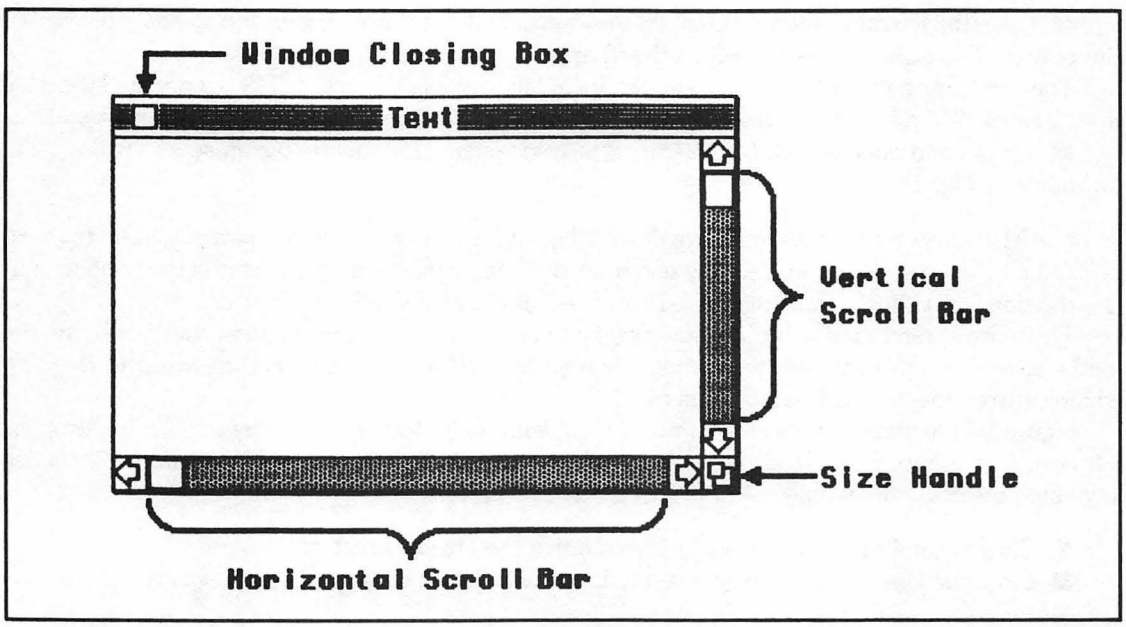


Fig. 1-8. Features of a Macintosh window.

To change the position of a window, drag the title to the desired position on the screen. To drag an item on the Mac:

1. Point to the item.
2. Press and hold down the mouse button.
3. Move the mouse to the new location.
4. Release the button. The object will be redrawn in the new position.

■ Move the Text window to a new location on the screen by dragging its title.

To change the size and shape of a window, drag the Size Adjustment Handle, found in the lower-right of the window.

■ Change the size of the Text window. Do this several times. Notice that the window can be made to fill the screen, or it can be made very small indeed.

In addition to the Text window, we will need to use the Instant window.

■ To open the Instant window, pull down the Windows menu at the top of the screen. MacPascal certainly has a lot of window types!

■ Choose Instant from the menu. The new window will appear on the screen.

■ Using the techniques you just practiced, adjust the Instant and the Text windows until your screen resembles Fig. 1-19. Now we are ready to begin.

■ Select the Instant window by clicking inside it with the mouse. Notice that the arrow pointer

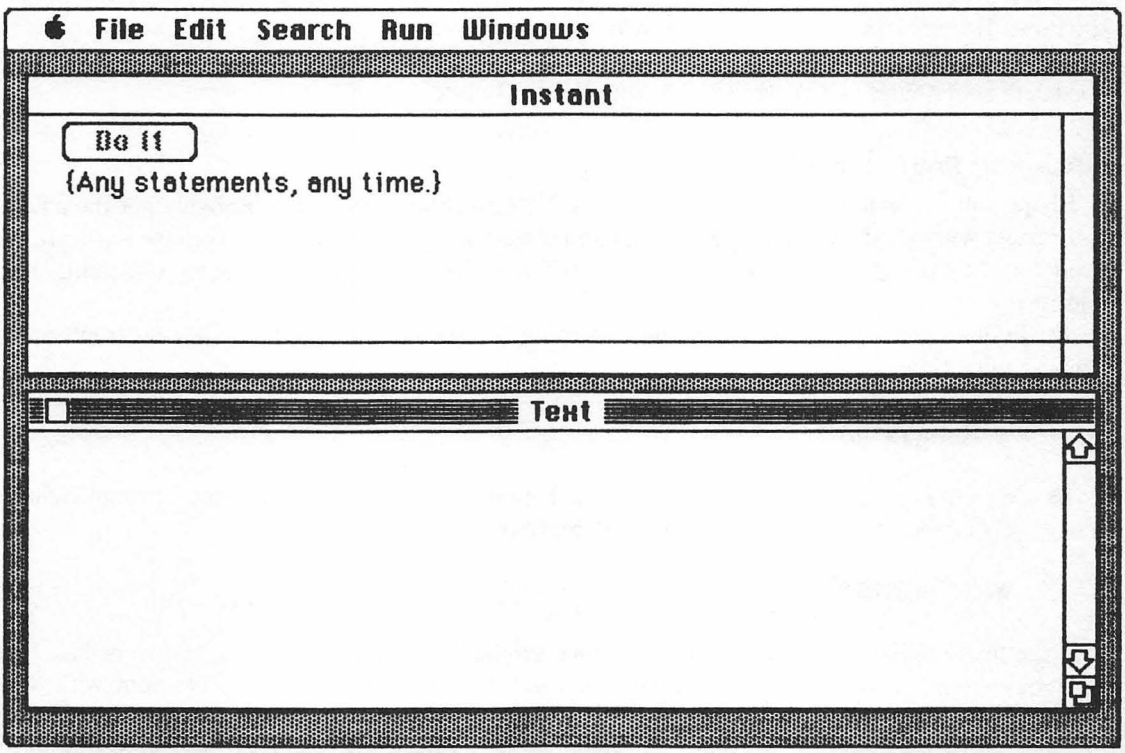


Fig. 1-9. Arrange your screen like this.

becomes a text pointer when the mouse is pointing inside the Instant window. This indicates that Pascal is expecting you to type something in the window. Notice that some text already appears in the window. Pascal has selected it for you, and it is ready for deletion.

■ Prepare to type by pressing the Return key. The typing cursor will move to the left side of the window, and you are ready to get started. The preselected text will disappear.

■ Now, type the following text. Be careful to type everything exactly as shown, including both apostrophes and the parentheses. type this:

```
writeln('hello')
```

This is a Pascal procedure statement. You may be able to guess what it is instructing the computer to do, but let's check out your intuitions.

■ Locate the box in the Instant window that says Do It, and click the box with the mouse. If all goes well, the following events will take place:

1. The Do It box will change from black to grey.
2. The disk drive will start up and work awhile.
3. The Do It legend will change to Doing It.
4. The word "hello" will appear in the text window.

If all of this happened—great! Otherwise recheck your typing. You can return to the Instant window and edit your text using the methods discussed in Chapter 1. If your typing is correct, reread my instructions. Be sure that you follow them exactly.

Generally we describe this process as *running* or *executing* a Pascal statement. If I ask you to run or execute something, you do this by clicking the Do It box.

ERRORS IN PASCAL STATEMENTS

I hope you are wondering why I made such a big fuss about errors. You probably got the advertised results without any problem. However, computers are very picky, and humans are error prone. If you don't say things in just the right way, you will not be understood. A missing apostrophe is a major error.

■ To illustrate this, edit the text line, removing the second apostrophe. (The apostrophes act as single quotation marks.)

```
writeln('hello)
```

■ Now click the Do It box and watch what happens. This time, nothing seems to go right. First your Mac will beep, and the text will be reprinted to look like this:

```
writeln('hello)
```

Then the line will be labeled with a thumbs down symbol. Finally, the error box shown in Fig. 1-10 will appear. After all this, no new text appears in the text window. No, Pascal is not happy with your edited line at all.

Let's deal with the problems one at a time. The new Bug window is used by Pascal to inform you that you have violated one of Pascal's many rules of form. A *bug* is a computer term for an error,

Fig. 1-10. A Macintosh Pascal Bug window.

hence the lady bug graphic. The message in the box is not particularly important; we will see some meaningful bug messages later.

■ Before you will be allowed to continue, you must close the Bug window by clicking anywhere within its border.

When Pascal encounters an error in a line, it will sometimes reprint the questionable portion in outline type. These are the easy errors to find, but we shall encounter errors that are not so elementary. If you tell Pascal to execute the line with the error, Pascal will also flag the offending line with the thumbs down symbol and produce a Bug window with an appropriate message.

■ To correct the problem, just edit the apostrophe back in. You can check the correction by taking advantage of a special feature of Macintosh Pascal.

■ Pressing the Enter key will cause an error check. After you have replaced the apostrophe, press the Enter key. The outline type and the thumbs down will disappear, indicating that Pascal is satisfied with the line again. Run the line again by clicking the Do It box. This will confirm that the line once again performs acceptably.

Errors of form such as the missing apostrophe are often called *syntax errors.* Syntax refers to the structure of a sentence or, in this case, of a programming statement. Syntax has to do with grammar and the building of sentences that obey the rules of sentence structure. Every programming language has rules of syntax, which are needed to allow the computer to understand your commands. By leaving out the second apostrophe, you confused Pascal. The word "hello" was never ended properly.

If both single quotation marks are left off, other problems appear.

■ Edit the line to remove both apostrophes and try to execute the command. This time, although no outline text appears, the thumbs down graphic is displayed. Also, the Bug window appears, and a message declares that

```
The name "hello" appears as an undeclared identifier.
```

At this point, it is too early to fully explain this message, but I wanted you to have encountered it and to know what caused it to appear: the missing single quotation marks.

MORE ABOUT PRINTING TEXT

Now we can begin to dissect the features of the statement we have been using. It begins with the word WRITELN, which is a procedure instructing Pascal to print something in the text window. After WRITELN, Pascal looks for something to be printed. Since 'hello' serves to modify the effect of WRITELN, it is a parameter of WRITELN. All parameters are enclosed in parentheses.

Pascal does not care about case, and you can type commands using any combination of upper- and lowercase. All of the examples in this book will use lowercase. To make Pascal terms stand out in the main text, however, I will capitalize them. This will eliminate confusion between Pascal and English words. For example, "IF" is a Pascal term, but "if" is an English word, used in a sentence as a conjunction.

In this case, the thing to be printed is the word "hello." Words, and text in general, are referred

to as *strings*. A string is a series of characters, always set out by apostrophes (also called *single quotes* or just *quotes*). Strings can contain letters, numbers, punctuation marks, and any other printable character. Within A Pascal text, I will use the single quotes expected by Pascal. In other places I will use standard double quotation marks.

A string is one example of a Pascal *expression*, a structure in Pascal that represents a value. A *value* is another term for a piece of *data*, which is a piece of information manipulated by the computer program.

So our statement has these three features:

1. The procedure WRITELN
2. An expression, which in this case is a string
3. The parentheses, which mark the expression as a parameter of WRITELN

The string printed by WRITELN can be rather long.

■ To illustrate, change the current statement to read like this:

```
writeln('The moving finger writes; and having writ,')
```

■ Run the statement in the Instant window. To continue this little poem, we can add another WRITELN statement.

■ Edit the Instant window to include the three statements shown below. Notice that a semicolon has been added to the end of the first line.

```
writeln('The moving finger writes; and having writ,');
writeln('Moves on: nor all your Piety nor Wit');
writeln('Shall lure it back to cancel half a Line.')
```

The semicolons at the ends of the lines are the new features in this example. Pascal requires that statements be separated by semicolons. This seemingly simple requirement will require us to spend a great deal of time learning the structures of the various types of Pascal statements. But there is no way around it. If the semicolons are used improperly or if they are missing difficulties arise.

■ First execute the lines as presented above, and observe the results.

■ Then edit out one of the end-of-line semicolons, and watch what happens when you run the lines again. The thumbs down graphic will appear along with the Bug window. Pascal does not take kindly to missing punctuation.

This can be a problem if you simply add a new statement to the Instant window without deleting the old statements or adding semicolons to them. Pascal attempts to execute all of the statements in the Instant window when you click the Do It box. If semicolons are missing, an error will be signaled. The secret is to edit out any statements you do not wish to retain before trying new ones.

At some point, you may enter enough lines in the Instant window to cause old text to disappear off the top of the window. These statements are still active, even though they cannot be seen. This can produce some mysterious results when you run some statements. If in doubt, check the entire Instant window. You can scroll through a window by sliding the vertical scrolling bar up or down. You can also enlarge the window to accommodate more text by dragging the size adjustment box.

I hope that several messages are getting across, but one remains unstated. If you are not sure about how a feature works, experiment with it. You cannot possibly hurt anything, and you might

learn something useful. For example, what would happen if the second WRITELN did not appear in the statements you just entered? Would things be different now if the quotation mark were missing after "having writ?" Take a moment to experiment.

As mentioned, the procedure WRITELN instructs Pascal to print the following expression on the text screen. But the full meaning of WRITELN is, "print the expression and start a new line." When we wish to continue printing on the current line, we would use the Pascal procedure WRITE.

■ Erase all text from the Instant window and type in these lines:

```
write('The moving finger writes;');
write('and having writ,');
write('Moves on.')
```

■ When you DO these lines, Pascal prints this in the Text window:

```
The moving finger writes;and having writ,Moves on.
```

There are a couple of problems with this. First, we would like a space between the semicolon and the word "and." Second, "Moves on." should begin a new line. In the first case, we must add a space inside the quotation marks, since WRITE will not add spaces.

To correct the second problem, we will add a WRITELN after the second line. This is the first example we have seen of a procedure that was not accompanied by a parameter. However, one feature of WRITELN is that it starts a new line, and it does this even if it does not have any text to print.

■ Edit the Instant window, adding a space between "writes;" and the single quotation mark that ends the text. Also add the new WRITELN statement.

```
write('The moving finger writes; ');
write('and having writ,');
writeln;
write('Moves on.')
```

■ When you run the statements now, Pascal should print the text like this:

```
The moving finger writes; and having writ,
Moves on.
```

Alternately, you could have started a new line by substituting a WRITELN for the second WRITE.

We should pause to note that the text we are printing contains a semicolon after "writes." This semicolon is not confused with the semicolons that separate the statements, since it is enclosed by the quotation marks. In fact just about any printable characters—letters, numbers, and punctuation marks—can appear in a string. Even a space is considered a printable character and can appear in a string.

The one difficult character is the apostrophe, since Pascal might confuse it with the apostrophes (single quotation marks) at the ends of the strings. To indicate an apostrophe within a string, it is simply entered twice, like this:

```
writeln('Nothing so needs reforming as other people''s habits.')
```

■ Enter this line and run it to observe the results. (You will probably want to edit out the lines

we have been working with so that you can start with a clean Instant window.) Please notice that there are two apostrophes in "people's," not a double quotation mark. This can be very easy to misread. When the string is printed, however, only one of these apostrophes will be reproduced.

With any luck, you have begun to develop some curiosity concerning Pascal errors. I have told you that an apostrophe in a string is indicated by two apostrophes, implying that a single apostrophe within a string would be incorrect. This could cause you to wonder what Pascal would do with a single apostrophe. Well, edit one of them out and see. Can you explain what happens when you DO the modified statement?

WORKING WITH NUMBERS

Strings are not the only things that can be printed. Pascal can also work extensively with numbers.

- As an example, type in and DO this statement:

```
writeln(12345)
```

- This probably doesn't seem to be so special, but try this:

```
writeln(123 + 456)
```

This time, Pascal prints 579, the sum of 123 and 456. I am sure you suspected that Pascal could do arithmetic, but we have at last seen an example of this capability. We will soon see that it can do much more.

We should pause here to distinguish between numbers and strings.

- Edit the Instant window to read:

```
writeln('123 + 456')
```

In other words, change the contents of the parentheses to agree with our definition of a string.

- Now, when you run the line, Pascal prints "123+456," not "579." So, numbers can appear in strings, but they are not interpreted as numbers.
- Next edit the line to read as follows:

```
writeln('123' + '456')
```

- When you try to run this statement, Pascal gets upset. Thumbs down is displayed, and the Bug window appears with the message "Types are incompatible." By placing the + outside of the strings, you have told Pascal that it should be interpreted as an addition symbol, not as a character in a string. However, digits in strings are not numbers and cannot be added, so Pascal produces the error message.

In fact, numbers and strings are completely different entities as far as Pascal is concerned, and each has special properties. We shall learn much about these properties in the next few chapters.

The phrase 123+456 is another example of a Pascal expression. Expressions often use operations such as addition or subtraction to prepare values for use as parameters in statements. In this example, the operator + combines the numbers 123 and 456 to form a single number, 579.

Often we would like to display things that combine strings and numbers. This is quite easy to do. One method is to use WRITE.

- Clear the Instant window. Then enter and DO these statements.

```
write('The sum of 123 and 456 is');
write(123 + 456);
writeln('.')
```

This will print the message

```
The sum of 123 and 456 is      579.
```

The extra spaces that precede the 579 result because Pascal is allowing space for the largest possible integer. This will be done unless we instruct Pascal to print in a narrower space by including a minimum width in the printing instruction. To instruct Pascal to print with a minimum width of four, edit the second line like this:

```
write(123 + 456:4);
```

Now Pascal prints the text like this:

```
The sum of 123 and 456 is 579.
```

If we had specified a minimum width of 3, no space would have appeared between "is" and "579." The number specifying the printing width is called a *field width parameter.*

To print this line, we used three WRITE statements. Often a simpler method may be used. By separating the items with commas, multiple items can appear in the same WRITE or WRITELN statement. For example, this statement will print exactly the same thing as the last line example:

```
writeln('The sum of 123 and 456 is ', 123 + 456:4, '.')
```

Statements such as this can be difficult and confusing to type. It is very easy to leave out a quotation mark, a comma, or a space. Examples in this book should look just as they will appear on your Macintosh screen, but you will still have to be careful to type them exactly.

Pascal will perform the four basic arithmetic operations. The symbols for these operations are:

+	Addition
-	Subtraction
*	Multiplication
/	Division

This last operation, division, works somewhat differently from the other three. The result of a division is often a fraction. Therefore, Pascal represents the results of all divisions as decimal fractions.

■ To illustrate, enter and execute this line.

```
writeln(5 / 3)
```

You might have expected Pascal to print something like 1.66666 or perhaps 1.667. However, Pascal responds by printing 1.7e+0, an expression that may seem curious. This is the normal way that Pascal represents decimal fractions, which are referred to as *real numbers* in Pascal. A full explanation of this method of representing real numbers will be presented in Chapter 8. For now, let's take steps to make the number appear more normal.

■ Edit the line to read:

```
writeln(5 / 3:7:4)
```

■ Now when you run the line, Pascal prints 1.6667. With real numbers, we may include a second *field parameter.* The first parameter instructed Pascal to reserve seven spaces in which to print the entire value of the expression. The second number, the 4, told Pascal to print the number with four decimal places. Real numbers (and only real numbers) can accept a second formatting parameter to indicate the decimal places desired. If this number is missing, the standard real notation is used.

Pascal works with two types of numbers: *integers* and *reals.* Integers are whole numbers, which may be represented without decimal points. Real numbers are always represented with decimal points. So, 3, written as a real, would be written as 3.0. 3 and 3.0 are not equivalent so far as Pascal is concerned. This is an important distinction, for we will often encounter situations where one number type may appear, but the other cannot.

Field parameters are examples of places where integers must be used.

■ To illustrate, try using the real number 4.0 as a field parameter by editing the statement in the Instant window as follows:

```
writeln(5 / 3:7:4.0)
```

When you DO this version of the statement, Pascal's Bug box appears with the warning that, "Only integers are allowed as colon modifiers."

Multiplication, subtraction, or addition will produce integer results if they operate on integers. If either or both numbers in the expression are real numbers, however, the result will be real. Consider these statements and what they print, as shown in the column on the right:

`writeln(5 * 3)`	15
`writeln(5 * 3.0)`	1.5e+1 (equivalent to 15.0)
`writeln(5 * 3.0:5:4)`	15.0000
`writeln(5.0 * 3.0:5:4)`	15.0000

Division always produces a real result:

`writeln(5 / 3)`	1.7e+0 *(equivalent to 1.7)*
`writeln(5 / 2:5:4)`	1.6667
`writeln(5 / 2.0:5:4)`	1.6667
`writeln(5.0 / 2.0:5:1)`	1.6667

Usually, Pascal will accept an integer where it expects a real number, but when it does, the integer will always be converted to a real number. So the result of any calculation involving a real number will be real. And the result of any division performed with "/" will be real. What happens if we want integer results from a calculation involving a real number?

To begin with, Pascal provides two operators that are associated with integer division. DIV is used to calculate an integer quotient. MOD is used to calculate the *remainder* of an integer division. Here are some examples:

THESE STATEMENTS	PRINT THIS	THESE STATEMENTS	PRINT THIS
writeln(12 div 4)	3	writeln(12 mod 4)	0
writeln(12 div 5)	2	writeln(12 mod 5)	2
writeln(4 div 5)	0	writeln(8 mod 5)	3

In later chapters we will encounter several situations that require the use of DIV and MOD.

Depending on the desired result, Pascal provides two methods of converting real numbers to integers. One method involves rounding: the decimal fraction is converted to the closest whole number. For example:

```
5.123  rounds to  5,
5.789  rounds to  6
```

If the fractional part of the number is .5 or greater, the rules of rounding call for rounding up to the next higher integer. So,

```
5.5    rounds to  6.
```

Rounding is performed with the Pascal function called ROUND. A *function* is similar to a procedure in that it instructs Pascal to perform a task. However, the purpose of a function is to produce a value. Therefore, functions are always found within the parameter part of the statement. A function alone does not make a complete statement. So, to round off a number, we would have to say something like:

```
writeln (round(5.789))
```

This statement would cause Pascal to print "6" in the text window. The expression "round(5.789)" has two parts: ROUND, which is the function, and the parameter 5.789. The parameter provides the value that ROUND is expected to work on.

Functions, like procedures, expect parameters to be enclosed in parentheses. In this example, parentheses have been placed inside of parentheses. Parentheses used in this manner are called *nested* parentheses. When using nested parentheses, you need to make sure of a few things. Of course, the parentheses should enclose the proper things. But they must also be balanced. For every left parenthesis, there must be a matching right parenthesis. When nesting schemes get complicated, we can very easily find ourselves with unbalanced parentheses.

■ Remove the last parenthesis from the WRITELN statement. What is Pascal's response when you execute the statement?

The value produced by a function is often described as the *output* of the function. Alternatively, we may say that a function *returns* a value. We often speak in terms of *calling* a function, and the use of a function is often referred to as a *function call.* This is an appropriate way of phrasing things since it emphasizes the fact that some other Pascal structure uses or calls upon the function to produce a value.

■ Try to execute the statement

```
round(5.789)
```

to see how Pascal reacts. Functions always are used to produce a value and output the value to some procedure that will use it. Therefore, function calls will never be found alone. They will always be in the context of a procedure or some other statement.

On some occasions, rounding is not what we want. Instead, we would simply like to throw away the fractional part of a real number. This process is called *truncation* (to truncate is to shorten something by cutting off a portion of it). Truncation is performed with the TRUNC function. Compare the results of rounding and truncating several real numbers:

FUNCTION	OUTPUT	FUNCTION	OUTPUT
round(7.1)	7	trunc(7.1)	7
round(7.49)	7	trunc(7.49)	7
round(7.50)	8	trunc(7.50)	7
round(7.9)	8	trunc(7.9)	7

We will see examples of the use of ROUND and TRUNC in later chapters.

We should investigate one more topic regarding arithmetic expressions.

- Enter and run this statement. It is intended to average the numbers 4, 5, and 9:

```
writeln(4 + 5 + 9 / 3)
```

What is printed when you run it? The correct average would be 6.0((6.0e+0), but the computer has printed 12(1.2e+1). The problem is that Pascal does not interpret the statement in the way that we tend to.

Expressions such as these are ambiguous. Here are two possible interpretations of this expression:

"Add 4 + 5 + 9 and divide the result by 3".
"Add the sum of 4 + 5 to the quotient of 9 / 3".

The first interpretation is the one we want if 4, 5, and 9 are to be averaged, but the second interpretation is the one Pascal chose.

To eliminate confusion, Pascal obeys *rules of precedence.* Certain operations are always carried out before others. When given an expression to evaluate, Pascal first works from left to right, carrying out all of the *, /, DIV and MOD operations. The Pascal again works left to right, performing the + and – operations.

If we wish to defeat this normal process of evaluation, we can group expressions in parentheses. Operations in parentheses will always be carried out first. So, to properly average the numbers, we must form the statement like this:

```
writeln((4 + 5 + 9) / 3)
```

Now the additions will be completed before the division is performed. As in all situations involving nested parentheses, we must be careful that the parentheses are balanced.

A FIRST LOOK AT GRAPHICS AND MUSIC

Macintosh Pascal is not limited to working with text and numbers. It also has a very impressive repertoire of graphics and sound features. While we will need to look at some more advanced Pascal techniques to really take advantage of Pascal's sound and graphics, we can easily get a taste of what lies in store.

■ First let's generate some sound. Type in and DO this statement:

```
note(440, 100, 50)
```

A single note will be produced. The NOTE procedure requires three parameters, all of them integers. Many Pascal procedures require more than one parameter. In these cases, the parameters are separated by commas. Here is the explanation of the parameters in this example:

440 determines the frequency in cycles per second (Hertz).
100 determines the loudness. This must be an integer in the range of 0 to 255. The higher the value, the louder the sound.
50 determines the duration. This also is an integer in the range of 0 to 255. The higher the value the longer the duration.

Experiment with the NOTE procedure, substituting different values for each of the parameters. You could play a simple tune by placing several NOTE procedure statements in the Instant window.

To illustrate Pascal graphics, you will need to open the Drawing window.

■ Do so by selecting Drawing in the Windows menu. Leave the shape of the window as it is, but move it if necessary so that you can see the Instant window.

■ Here is a sample MacPascal graphics statement. Try it out.

```
framerect(10, 20, 100, 150)
```

When you execute this statement, a rectangle will appear in the Graphics window. FRAMERECT instructed Pascal to draw an open rectangle, using the four parameters to determine the dimensions.

Positions in the Graphics window are determined by row and column number. The dot at the top left of the window is in row zero and column zero. As we move to the right and down, the row and column numbers increase, each reaching about 200 at the bottom right corner.

The four parameters in the statement told Pascal to draw a rectangle based on these dimensions:

- The top is at row 10.
- The left side is at column 20.
- The bottom is at row 100.
- The right side is at column 150.

Experiment with some different parameters. If you want to clear the Drawing window, choose Reset under the Run menu.

What, if anything, happens if the left side parameter is greater than the right side or if the top is greater than the bottom? Experiment and find out.

■ Reset the Graphics window (choose Reset in the Run menu) and try this graphics expression:

```
paintrect(10,20,190,150)
```

MacPascal also has routines for drawing ovals. As with rectangles, ovals can be either painted or framed. Let's investigate several of the other MacPascal graphics procedures.

■ Enter and DO the following statements one at a time so that you can observe the effect of each. We will examine the details concerning each statement after you are done.

```
penpat(ltgray);
paintoval(60,5,195,100);
penpat(dkgray);
pensize(10,3);
frameoval(30,10,120,190);
invertrect(50,40,160,180);
```

The dimensions of ovals are specified just as the dimensions of rectangles are. The parameters indicate the top, left, bottom, and right sides of the rectangle that would contain the oval.

Normally, Pascal draws using a black pen, but this can be changed by PENPAT, which accepts parameters of WHITE, BLACK, GRAY, LTGRAY, or DKGRAY. The first PAINTOVAL was executed with a pen pattern of LTGRAY.

Notice that the words WHITE, BLACK, GRAY, etc. are serving as parameters for PENPAT. These words are obviously not numbers, and they are not strings since they are not quoted. They are examples of Pascal's ability to assign names to data, a feature that we will examine in detail in future chapters.

Next the pen pattern was changed to DKGRAY. Then we see that, in addition to changing the pen color we can change its shape. PENSIZE accepts two parameters, specifying the width and the height of the pen. This change caused the next oval to be drawn with a frame of varying width.

The final command is particularly interesting. Macintosh graphics are drawn as dots on the screen. You can clearly see these dots in the gray patterns you have already drawn. The procedure INVERTRECT inverts the color of all of the dots within the rectangle that it draws. That is, white dots become black and black dots become white. A similar procedure, INVERTOVAL, will do the same thing for an oval.

We have barely scratched the surface of MacPascal graphics. Many of the graphics commands that are available require advanced knowledge of Pascal. I have tried to give you some interesting ones that you can use without having to study the whole book first.

ENDING A PASCAL SESSION

You are now through with this chapter. To leave Pascal, choose Quit in the File menu. This should return you to the main Macintosh screen.

If by any chance a dialog box appears asking if you wish to save the changes to your program, click the box that says Discard. The significance of this box will be discussed in the next chapter.

YOUR PASCAL VOCABULARY

Here are the Pascal words that you learned in this chapter:

Procedures

WRITE WRITELN

NOTE

FRAMERECT PAINTRECT FRAMEOVAL PAINTOVAL

INVERTRECT INVERTOVAL PENPAT PENSIZE

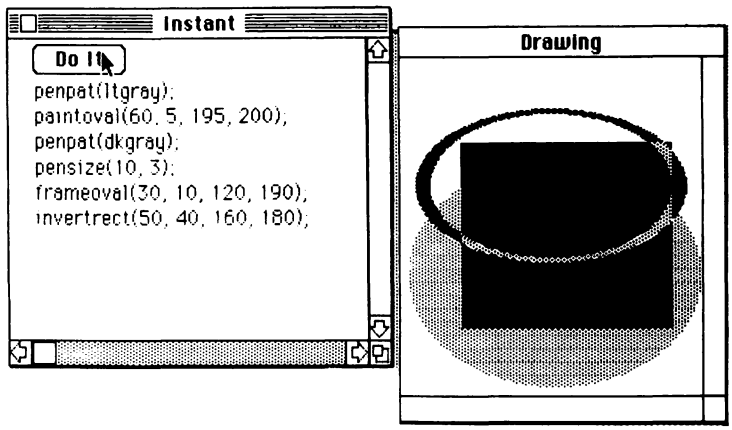
Operations

\+ - * / DIV MOD

Functions

ROUND TRUNC

Chapter 2



Writing and Running Pascal Programs

The first chapter may have given you a sense of the way complex tasks are performed in Pascal: large tasks must be broken down to smaller parts. You saw how complicated graphics can be created by drawing a large number of rectangles or elipses. You might surmise that long tunes might be played by stringing together several NOTE statements. Very large text passages could be printed using many WRITELN statements. However, doing this with the tools introduced so far would not be particularly efficient or interesting. The next step toward using Pascal in more powerful ways is to learn how to combine statements into programs.

A *program* is a way of grouping together the statements that are used to perform a large task. Once placed in a program, these statements can be stored and recalled for later use.

TOPICS COVERED IN THIS CHAPTER

- Typing programs in the Program window
- Running MacPascal programs
- Saving, closing and opening programs
- Manipulating different versions of a program
- The features of simple Pascal programs
- Pascal reserved words
- Searching and replacing in the Pascal editor
- Printing your programs

ENTERING AND RUNNING A PASCAL PROGRAM

■ If you have not done so already, start up your Mac and open Macintosh Pascal. This time,

the Program (labeled "Untitled"), Text, and Drawing windows are exactly what we want, so you will not have to do any rearranging.

Pascal starts out with the Program window selected. Several lines of text are present in the window. They are printed in inverse text, showing that they have been automatically selected for editing.

■ Press the Return key to clear the window.

You are now ready to type in a program. As you type in text, don't be surprised when Pascal makes some format changes. Some words will be converted to bold text, and some will be automatically indented. We will see that Pascal does these things to make large programs easier to read.

All of the error checking features of Macintosh Pascal will be active while you type text in the Program window. So, if you find that some text has been converted to outline type, look for errors such as the ones we examined in Chapter 1.

Here is a Pascal program. First I will show you the program as you will type it. You do not need to indent any lines or to worry about typing anything in bold text. As you complete each line and press Return; Pascal will retype the line as shown in Fig. 2-1.

■ Type this text in the Program window:

```
program first;
begin
writeln('My first program');
writeln('will add 456 and 789');
writeln(456+789)
end.
```

Notice several things while you are checking your typing. No semicolon follows BEGIN, but a period appears after END. We will examine the significance of these features in a later section.

While you were typing in your program, Pascal was converting it to the form shown in Fig. 2-1.

Again, all of these changes will be explained. For now, let's try the program out. Pull down the Run menu from the top of the screen. As shown in Fig. 2-2, this menu presents several choices. The one we are interested in right now is Go, which is similar to Do It in the Instant window: it tells Pascal to run the program in the Program window. After you select Go, the following events will occur:

1. A black box will surround Run.
2. The disk drive will start up. Pascal is checking your program to make sure that it is structured properly.
3. All of the menu items turn grey and a new black menu item, Pause, appears.
4. Pascal carries out the instructions in the program. This produces the text in the Text window.
5. The Pause menu then disappears, and the other menu options turn black again. This indicates that program execution is complete.

There should not be any real surprises. If you typed the program correctly, the WRITELN statements printed text in the Text window, just as they did when they were typed in the Instant window. In fact, you have as yet seen no advantage to placing commands into program form.

Advantages there are, however. An important one is that programs can be saved in disk files. As mentioned earlier, this allows you to reuse and modify programs in the future without typing them in again. This lets you get around the Macintosh's (or most computers') unfortunate inability to remember programs if its power is turned off. Let's see how this is done.

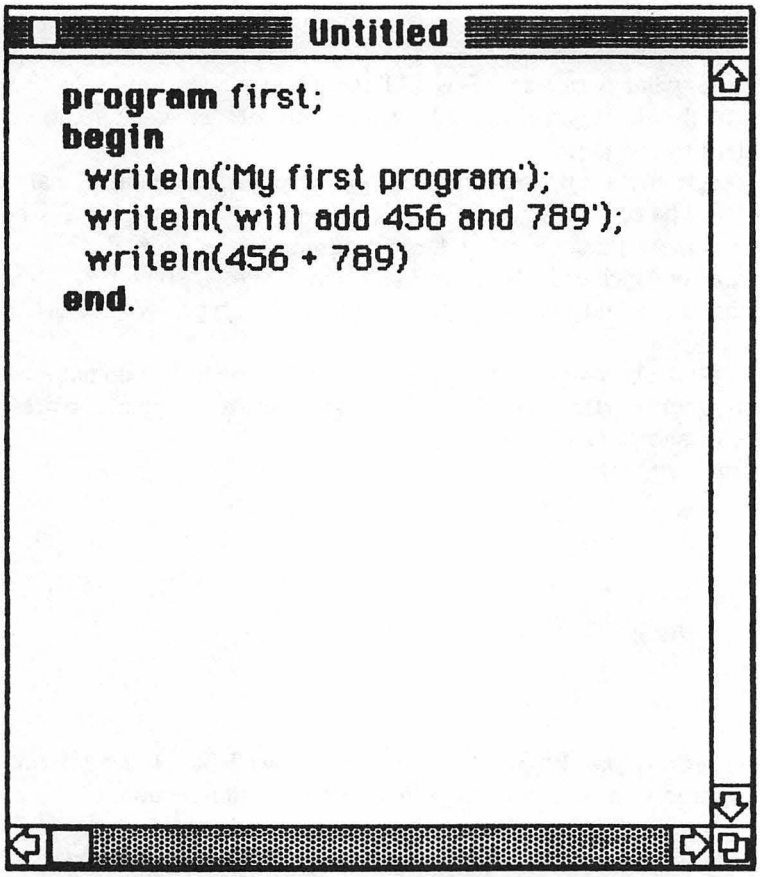


Fig. 2-1. The program window with the program FIRST.

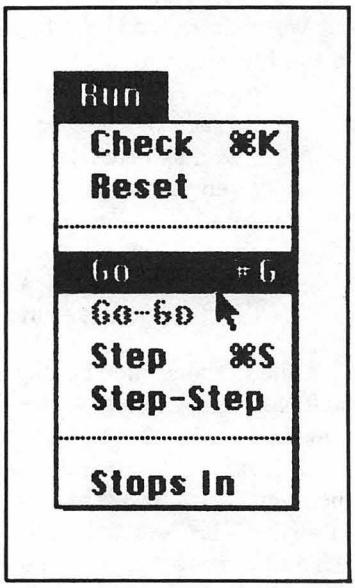


Fig. 2-2. The Run menu.

SAVING AND RETRIEVING PROGRAMS

Saving a program couldn't be easier. Choose the Save As option in the File menu. The dialogue window shown in Fig. 2-3 will be produced. The rectangle with the text cursor is there so that you can type a name for your program. This name will be assigned to the file that stores the program on disk.

■ For now, type this:

```
First Program
```

To save your program to disk, either press the Return key or click the box labeled Save. (If you wished to cancel the save operation, you could click the Cancel box.)

■ Press Return to save your program. The disk drive will operate for a few seconds. After that you will be returned to the Pascal windows screen.

Notice that the Program window is now labeled "First Program." Before it was labeled "Untitled." Your program now has a filename that will remain associated with it. This name could have been just about anything. Macintosh Pascal places only a few restrictions on the characters that can appear in a program name. The only restrictions are the following:

1. No colons are allowed.
2. The name cannot begin with a period.
3. The name can contain no more than 63 characters.

Since the restrictions are so few, I would strongly suggest that you make your program names meaningful so that you will recognize them easily in the future.

Now that your program is saved, you can leave Pascal without losing your work. You can turn off the computer, leave for days, and restore the program just as you left it. Let's see how.

■ Leave Pascal by choosing Quit in the File menu. This will return you to the Pascal file window screen. Notice that a new icon has been added. This is a Pascal program icon with the name "First Program." This demonstrates that your program has been added to the files on the Pascal disk. You could now turn off your Macintosh (after you've ejected the disk), secure in the knowledge that your program will be safely available the next time you want to look at it.

■ To retrieve your program, double click its icon. This instructs the Macintosh to start up Pascal and place your program into the Program window. When the Pascal windows appear, the Program window will be labeled "First Program," and your program will be displayed.

You can now modify the program without typing the whole thing again. In the following version

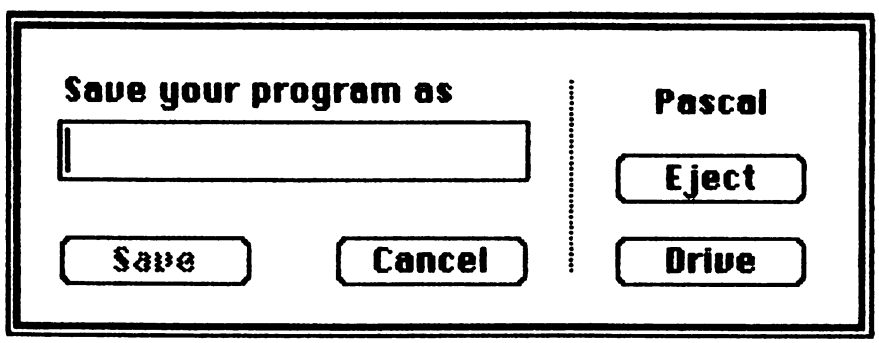


Fig. 2-3. The Save As dialogue window.

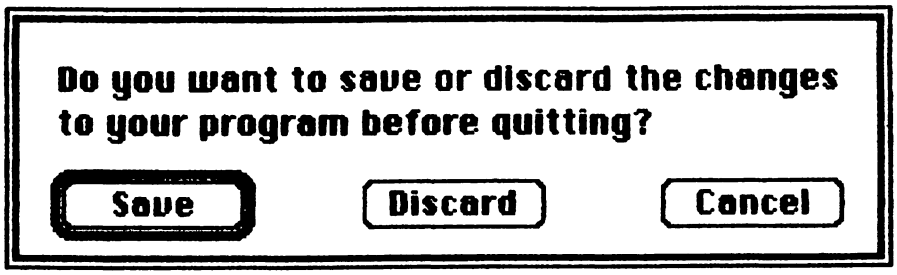


Fig. 2-4. The dialog box used to save programs when you are quitting Pascal.

of the program, I have used Pascal *comments* to point out the lines that are being changed. Pascal ignores text placed within curly brackets; this feature is frequently used to include explanatory notes within the program without affecting program operation. You will not want to include my change comments in your program. They are simply there to call attention to the modifications.

■ Edit your program to incorporate the following changes:

```
program first;
begin
  writeln('My first program');
  writeln('will multiply 456 and 789.');      {change to multiply}
  writeln(456 * 789)                          { change + to *}
end.
```

■ Try the modified version out by choosing Go. Not a very exciting modification, perhaps, but we will use it to make an important point. You are now the proud owner of two versions of this program. The second one you see mutliplies the numbers. But there is another version, the one you saved to disk. It is still safe and sound, and it still performs addition. You will often find yourself in this position. Now you must make a decision: do you wish to keep the first version of the program, the second version, or both?

First, lets assume that you wish to keep the multiplication version and discard the addition version. That is simple: merely throw the old version away.

■ Choose Quit from the File menu. Until now, Quit has simply returned you to the Macintosh file screen. Now, however, since you have changed the program, a new dialog box appears. This box, shown in Fig. 2-4, asks whether you wish to save or discard the changes you have made. Three responses are possible: Save, Discard, and Cancel. Clicking Cancel will return you to the Pascal screen.

■ To keep the changes, click the Save box. The disk drive will operate, and eventually the Macintosh file screen will reappear just as you left it. The icon for "First Program" has not moved. However, the contents of this file have been changed. To demonstrate this, restart Pascal by double-clicking the "First Program" icon.

This time, the multiplying version of the program will appear in the program window. Now, you ask, what happened to the addition version? It simply disappeared, to be replaced by the multiplication version. There is no way to recover the addition version without editing. So, you must be careful when you save programs. Don't replace a program version that you wanted to keep!

It is also possible that you would wish to discard the modified version of a program and retain the copy that was earlier saved to the disk.

■ To examine this possibility, modify the program you have been working with so that it performs a division.

■ Then click Quit in the File menu. This time, you should click the Discard box. The Save option returned you to the Macintosh file screen. So will Discard. The "First Program" icon remains where you left it, but you need to examine its contents before you have the whole story.

■ Double click the program icon. After Pascal has started, which version of the program is displayed? The multiplication version has returned. Now you know how to keep the old version of a program and how to replace the version with a new one. But what happens if you wish to keep both versions?

■ Again, change the program so that it performs a division of the two numbers.

■ Now, choose Save As from the File menu. You used this option the first time you saved your program. As before, Pascal puts up a box that is waiting for you to type a name.

■ Enter a name that is different from "First Program." An imaginative choice would be "Second Program." Then press the Return key, or click the Save box.

After the disk activity has ceased, you will be returned to the Pascal screen.

■ Choose Quit to leave Pascal. Notice that Pascal does not ask you whether you wish to save your program. Pascal is smart enough to know that you have not made any changes since the last time you saved it.

When you have returned to the File screen, you will notice that a second program file has been added. In addition to "First Program" a file was created with the name you typed in the Save As box. This file contains the most recent modification of your program. You can confirm this by double clicking the new file to start up Pascal.

Now that you have two programs saved, we can look at one more Pascal file feature. (There is much more, which we will cover at a later time.) It is not necessary to leave Pascal to retrieve a program from disk. At this time, you should be in Pascal with one version of your program before you. Suppose that you wished to switch to the other version or to any other Pascal program.

First, you must close the current program.

■ To do this, choose Close in the File menu. If you have modified the program, Pascal will ask you if you wish to save the modified version, using the same option box as we saw earlier with Quit. You will probably not see this box right now. Closing the file will cause the Program window to disappear.

■ To load in a different program, choose Open in the File menu. A dialog box will appear, similar to the one shown in Fig. 2-5. Inside the smaller box you will see a list of the names of several of the programs that are stored on disk. Since your Pascal disk probably contains several demo programs, it is unlikely that either of the programs you have saved will be shown right now.

■ Locate "First Program" by dragging the vertical scrolling handle down. Finding a program is easy, because the names are displayed in alphabetical order.

■ To open the program, just double click its name. "First Program" will be read in from disk and will be displayed in the Program window.

Incidentally, all of the programs on the disk will be displayed, even if you have placed them in separate file folders. Pascal will save new programs into the same folder that holds an older program with the same name, but folders are otherwise ignored.

If you would like to start a completely new program, Close the current program, saving it if you like. Then choose New in the File menu. The Program window will soon be named "Untitled" and will display the startup text.

That is enough file manipulation for now. To conclude, let us make a few observations:

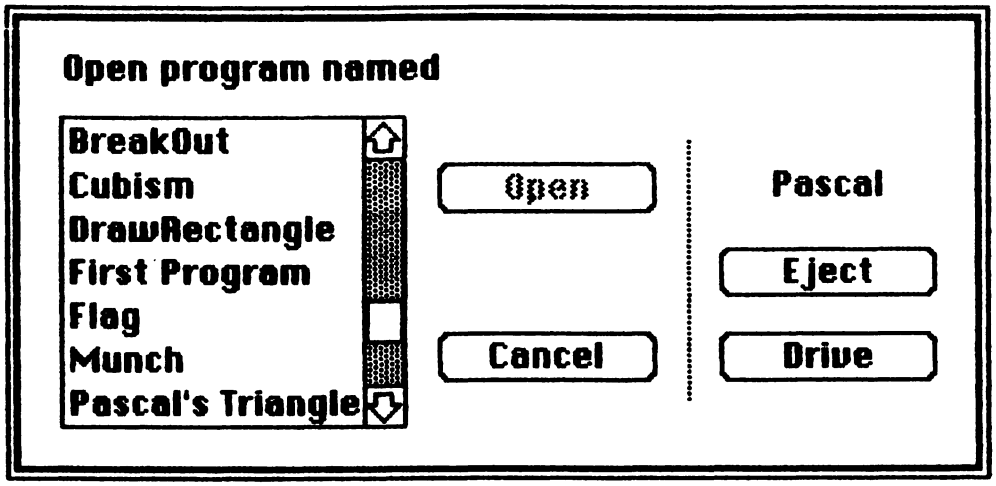


Fig. 2-5. The Open Program dialog box.

- To retain more than one version of a program, each version must be saved with a different name. If just one character is different in the two names, Pascal will be able to distinguish them. New names are generally created with Save As.
- Filenames may contain up to 63 characters, which may be upper- or lowercase. However the Macintosh does not distinguish between upper- and lowercase characters. This means that all three of these designations refer to the same file:

Program One program one PROGRAM ONE

- Filenames may contain any character you can type except the colon (:). The only other restriction is that they may not begin with a period.
- Unless a program is explicitly saved, it will be lost when you leave Pascal. This is why Pascal will not let you quit without asking if you want to save or discard your program.
- As a corollary to the last point, no changes to a program are remembered unless the new version of your program is explicitly saved. Provided you have not saved the new version, you can always recover the old one. In fact the Recover option in the File menu does just that: it throws away the version in the Program window and recovers the last version that was saved.

There is one last point about opening and saving programs. There will come a time when your Macintosh Pascal disk will not hold any more programs, or you will wish to execute a program that is stored on another disk. You can copy programs from one disk to another using techniques described in the Macintosh reference. However, you can also save programs to other disks or open programs that are stored on other disks.

Load a program into Pascal's program window. To save it to another disk, you will need a second Macintosh disk. Follow these steps:

■ Start to save it being sure to use the Save As option.

■ When you see the dialog box, notice the box marked Eject. Simply click this box and your MacPascal disk will be ejected.

■ Remove the Pascal disk and insert the disk you wish to save the program on.

■ If the disk has never been used before, a dialog box will appear, indicating that the disk must be initialized. If this happens, click the Initialize box. When initialization is over, MacPascal will then take over and save your program to the disk.

■ When necessary, Pascal will eject a disk and ask you for another. Just follow the instructions that Pascal displays.

Opening programs on the other disks is just as simple. Just eject the MacPascal disk, insert the disk that contains the program, and select the program you want from the menu. MacPascal will then tell you what to do.

THE ANATOMY OF A PASCAL PROGRAM

The program at the beginning of this chapter is about as simple as a Pascal program can get. Here it is again, so that we can discuss it.

```
program first;
begin
  writeln('My first program');
  writeln('will add 456 and 789.');
  writeln(456+789)
end.
```

At a minimum, a Pascal program must have two parts. The first part is the *heading*, which is always the first line of the program. The heading contains the Pascal word PROGRAM, followed by a name for the program. This name has no function other than to label the program; it need not be the same as the filename that is used to identify the program file on the disk. Program names can contain only letters, numbers, and the underscore character. The following are acceptable program names. Notice that you can mix upper- and lowercase.

```
version23          Smith_Company_inventory
```

However, these names are not acceptable, for the reasons cited:

Why_not?	Punctuation not allowed
Accounting Program	Spaces not allowed
76Trombones	Begins with a number.

Every program heading ends with a semicolon. Pascal will become upset if this is missing. Remove the semicolon, run the program by choosing Go, and observe the way Pascal reacts.

The second essential part of a program is the *statement* part. This part always starts with the word BEGIN and terminates with END. These words function much like punctuation marks in Pascal, marking the beginnings and ends of program sections. We will encounter them a great deal.

Because proper punctuation is so important in Pascal, let's point out a few things about the punctuation in this program. Notice that BEGIN is not followed by a semicolon. This is the first time we have seen a line that did not have to be separated by a semicolon. In fact, to place a semicolon after BEGIN would result in an error message.

Semicolons are used in Pascal programs to separate statements. No semicolon follows BEGIN because BEGIN is not a statement. BEGIN and END are more like punctuation marks than anything else. They do not do anything, they just divide the program up into segments. Since you will undoubtedly enter semicolons by accident in your future Pascal activities, it would be a good idea for you to place a semicolon after BEGIN and to observe the results when you try to execute the program by selecting Go.

For much the same reason, a semicolon does not appear between the last program statement and END. Since END is not a statement, it need not be separated from the preceding statement. This is, however, a rare case where Pascal will allow an optional semicolon. If you place one after the last WRITELN statement, the program will work properly, just as it did the first time you executed it.

(Actually, doing this introduces a new statement just ahead of END. This statement does nothing, so it is called the *null statement*. This statement has some important uses, as we shall see later. However, in this case it enforces consistency in the Pascal program structure. When we place a semicolon ahead of an END, we introduce a null statement, which is not followed by a semicolon. So, in a sense there is still no semicolon immediately prior to the END.)

The final bit of punctuation is the closing period. Pascal programs always have a period after the final END. Remove this period and execute the program to see how Pascal reacts.

There is one more aspect of the program listing that we might examine. Why are some of the words automatically printed in boldface by Pascal? PROGRAM, BEGIN, and END are examples of *reserved words*. These are Pascal words that cannot be redefined by the user. Reserved words are usually printed in bold face to distinguish them, something that the Pascal editor does for you automatically. Reserved words cannot appear as identifiers in a program. For example, no program can be named "BEGIN" since this is a reserved word. We will encounter several places where we must be careful not to use reserved words in Pascal.

All of the program statements you met in Chapter 1 may appear within the statement part of a Pascal program. This would be a good time for you to try creating, saving, opening, and editing some programs. Since you can now save your work, you can undertake some more complex tasks, such as assembling the NOTE statements to play a complete song.

■ In any case, write one or two programs, borrowing from the activities in Chapter 1 if you like, before you go on to the next chapter. The more comfortable you are with the Pascal editor, error checkers, and file systems, the easier you will find it to do the work in later chapters.

CHARACTERISTICS OF THE MACINTOSH PASCAL EDITOR

The Pascal editor is very similar to the Note Pad editor we used in Chapter 1. There are a few differences which should be pointed out, however.

Most obvious is the automatic indentation scheme. Ninety percent of the time this is a dandy feature. Unfortunately, there is no provision to override it the other ten percent of the time.

In most situations, the Macintosh allows you to select words by double clicking them. MacPascal also allows you to select entire lines by triple clicking.

MacPascal has a Type Size option in the Windows menu. This option allows you to select the size for the display of text in the Program window. I have generally found the normal 12 point size to be most acceptable, but the 9 point size will allow you to fit more on a line.

The MacPascal editor does not perform word wraparound. In other words, it does not begin a new line when you have typed to the right edge of the window. You will keep right on typing, even though the cursor has passed out of the window! You will have to enlarge the window or scroll it horizontally in order to see your text.

Horizontal scrolling is important since MacPascal expects a statement to appear on a single line in the editor. On occasion, statements can get too long to fit on the width of the screen, and you will have to scroll horizontally to edit the entire statement.

MacPascal's editor performs search and replace operations differently from other Macintosh editors, such as MacWrite. Search and replace operations have two steps. First the search requirements are entered into the search and replace dialog box. After that, the search or replace is performed by choosing options in the Search menu.

Searching and replacing both begin by choosing What to Find in the Search menu. This brings up the dialog box shown in Fig. 2-6.

Every search and replace operation must have something to search for. Type the text to be searched for in the Search for box.

If a replacement is to be made, the replacement text is typed in the Replace with box. This text will be substituted for the searched for text when a match is found.

You may also need to modify the search criteria. To select an option, click the circle beside it. A black dot within the circle indicates that the option is selected. These options are organized in pairs. The first pair is:

- **Separate Words:** A word is found only if it is surrounded by word separators, such as spaces and punctuation. With this option in effect, search would find "apples" in the phrase "I like my apples sauced." However, "apples" would not be found in "I like my applesauce." This is the option that is selected when you first open the dialog box.
- **All Occurrences:** When this option is selected, all instances of the search text are found. In the above example, both instances of "apples" would be found. Obviously only one of these two options may be active at one time. When you select one, the other is turned off.

The second pair of options is:

- **Case is Irrelevant:** "Apples", "apples", and "APPLES" would all be considered as matches. Case is ignored when performing matches. This option is selected automatically the first time you start a search.
- **Cases Must Match:** "Apples" matches "Apples", but does not match "apples." An exact match of upper- and lowercase characters is required. Of course, only one of the case options may be selected at a time.

After the text to be searched for is entered, and after the desired options are selected, click the

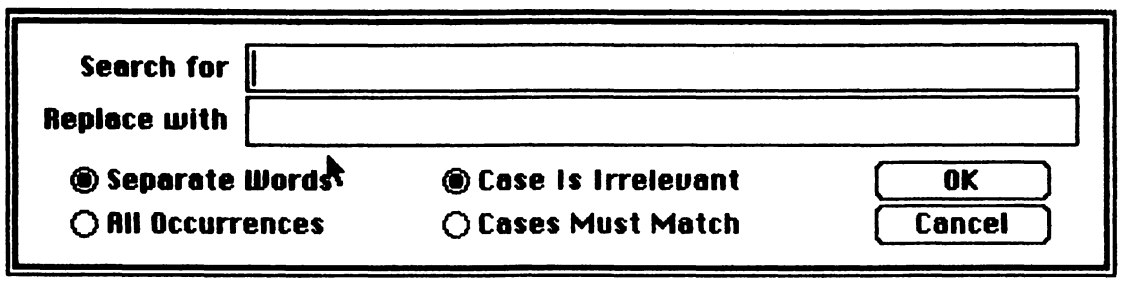


Fig. 2-6. The Search and Replace dialog box.

OK box to leave the dialogue window. Click the Cancel box to leave the window and discard any selections you made there.

After the search requirements have been entered, the other options in the Search menu may be used to perform the search or replace. There are three options:

- **Find**: Choose this option to locate the next instance of the text that you specified in the Search for box. The editor begins to search starting at the insertion point (the cursor location) or at the end of a selected section of text. When found, the text will be selected for editing.
- **Replace**: Choose this option after a Find has located search text that you wish to replace. The located text will be replaced with the text that was entered in the Replace with box. Replace is used when you wish to perform a replace in selected instances of the search text while leaving other instances alone.
- **Everywhere**: Choose this option to replace every instance of the search text with the replacement text. A dialog box will confirm that you really want to do this.

If you have used other text editors, you may find that Replace works a little differently from your expectations. Normally a replace operation replaces the search text with the specified replacement text. However, MacPascal's Replace substitutes the replacement text for the text that is currently *selected.*

To perform a replace, you must first select the text to be replaced. This can be done with the mouse or by using Find, since the found text is selected for you. After you have selected the text to be replaced, Replace may be used to substitute the Replace with text. Therefore, every selective replace is a two step operation.

If no text is selected when you do a Replace, the Replace with text will be inserted at the text insertion point.

Let's practice using Search and Replace.

■ Create the following program in your editor:

```
program doublecross;
begin
  invertoval(10, 10, 110, 110);
  invertoval(10, 90, 110, 190);
  invertoval(50, 50, 150, 150);
  invertoval(90, 10, 190, 110);
  invertoval(90, 90, 190, 190)
end.
```

■ Save this program with the name DOUBLECROSS.

First, we will change all of the instances of "oval" to "rect".

■ Choose What to Find in the Search menu.
■ Enter "oval" as the text to Search for.
■ Move the insertion pointer by clicking the Replace with box.
■ Enter "rect" as the text to Replace with.

Most of your editing options are available when entering text in dialog boxes.

Should we modify the search options? Let's try to search with the Separate Words option.

■ Click the circle to select Separate Words.
■ Click OK to leave the box.

■ Choose Everywhere in the Search menu, or type Command-E. (Command is the key with a looped design to the left of the space bar. To type Command E, hold down the Command key and press E.)

Oops, the editor didn't find anything—not surprising, since "oval" is not a separate word.

■ Enter the search dialog box by choosing What to Find.
■ Select the All Occurrences option and leave the dialog box.
■ Choose Everywhere.

This time the replace worked.

Now, let's do a partial replace by changing all but one of the instances of "oval" back to "rect". We will leave the third INVERTRECT statement alone.

■ Enter the search dialog box.
■ Type "rect" in the Search for box and "oval" in the Replace with box.
■ Leave the dialog box.
■ Move the text pointer to the beginning of the program.
■ Find the first instance of "rect" by typing Command-F or by choosing Find.
■ Do a replace by typing Command-R or by choosing Replace.
■ Find the second instance of "rect" by typing Command-F.
■ Do a replace by typing Command-R.
■ Find the third instance of "rect".
■ Find the fourth instance of "rect".
■ Replace the remaining two appearances of "rect" with "oval".

To illustrate a quirk in the replace feature,

■ Manually select the word DOUBLECROSS by double clicking it.
■ Perform a replace operation by typing Command-R.

"Oval" has replaced "doublecross". The editor was not influenced by the Search for text at all.

We used three command keys in performing search and replace:

- Command-F to find the next instance of the search text
- Command-R to replace found text with the replacement text
- Command-E to replace every instance of the search text with the replacement text

These functions may also be performed by using the mouse to select options from the Edit menu.

PRINTING YOUR PROGRAMS

If you have an Imagewriter or other Macintosh compatible printer, you can easily print your Pascal programs. To do this, choose Print from the File menu. The dialogue box shown in Fig. 2-7 will be displayed. There are several options in this box. Click the circle beside an option to select it.

Quality	High, Standard, or Draft: The best quality is High. The fastest is Draft. Standard is a good compromise of quality and speed; it will be selected for you unless you make a change.
Page Range	All or From . . To: To print the entire program use All. Normally, all of the program will be printed. To print part of the program you must click From . . and enter the page range to be printed in the associated boxes, for example, From: 2 To: 4. This refers to pages on the printer, not to the screen dimensions.
Copies	Enter the number of desired copies in the box. One copy is normal.
Paper Feed	Continuous or Cut Sheet: Select to match your printer setup.

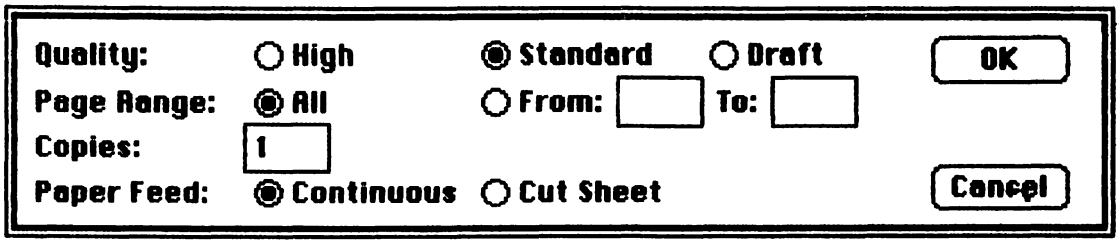


Fig. 2-7. The Printer Options dialog box.

Ordinarily, you will not need to change any of these options. To initiate printing, click the OK box. The message box shown in Fig. 2-8 will be displayed during the entire printing process. This takes a while since the printer information is first stored on your floppy disk. From there it is sent to the printer.

Since the printed information is stored in a temporary file on disk, you must be sure that you have sufficient free space on disk to accommodate the file. Therefore, you should not try to pack your disk with programs. If you have a second disk drive, store your programs on a disk in that drive. This will leave plenty of space on your main disk for printer spooling files.

If you have only one disk drive, copy little used programs off to a storage diskette. You can also save a bit of space by clearing out anything that is cluttering your Scrapbook.

In my experience, running out of disk space during a printer operation can result in a system malfunction that will prevent you from saving your program. It is a good idea, therefore, to save the program before trying to print it. Then, if the computer malfunctions you can turn it off and restart it knowing that your program is safely stored on disk.

There is one other dialogue box that controls printer function. It is selected through the Page Setup option in the File menu. The dialogue box is displayed in Fig. 2-9.

Normally the preselected options are what you want. The options are:

Paper:	US Letter	For 8 1/2 by 11 paper
	US Legal	For 8 1/2 by 14 paper
	A4 Letter and International Fanfold	For metric paper sizes
Orientation:	Tall	Best for most purposes
	Tall Adjusted	Not recommended
	Wide	Print sideways on page

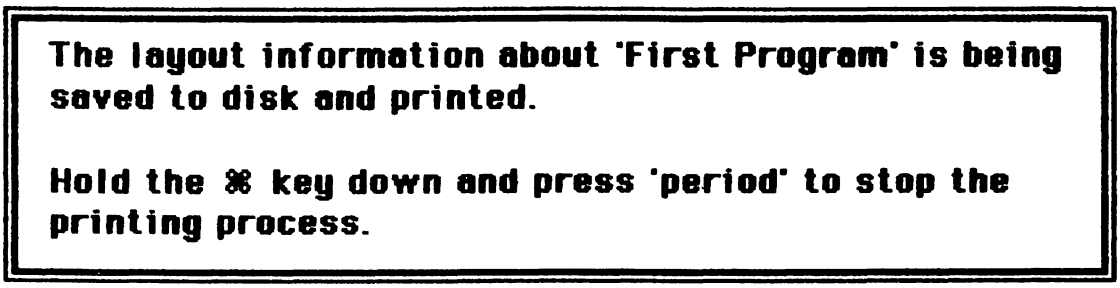


Fig. 2-8. The message displayed when printing is in progress.

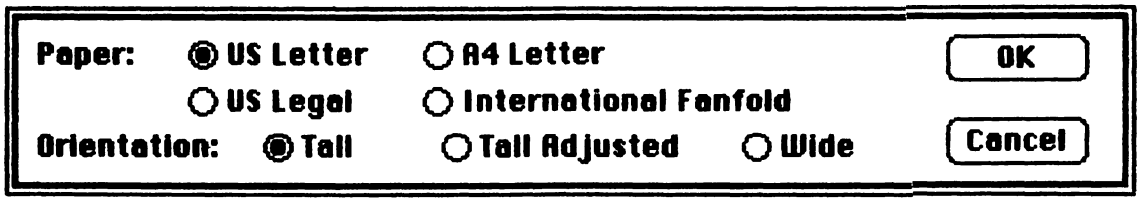


Fig. 2-9. The Printer Setup dialog box.

OK and Cancel function as they do in most dialog windows. OK accepts any changes you make, while Cancel discards them.

The only option switch you are likely to make here is between Tall and Wide. Wide will allow you to print programs with long lines across the long dimension of your paper.

YOUR PASCAL VOCABULARY

You now are familiar with the following Pascal words. Words that were introduced in this chapter are printed in bold face type.

Reserved Words

PROGRAM **BEGIN** **END**

Procedures

WRITE WRITELN

NOTE

FRAMERECT PAINTRECT FRAMEOVAL PAINTOVAL

INVERTRECT INVERTOVAL PENPAT PENSIZE

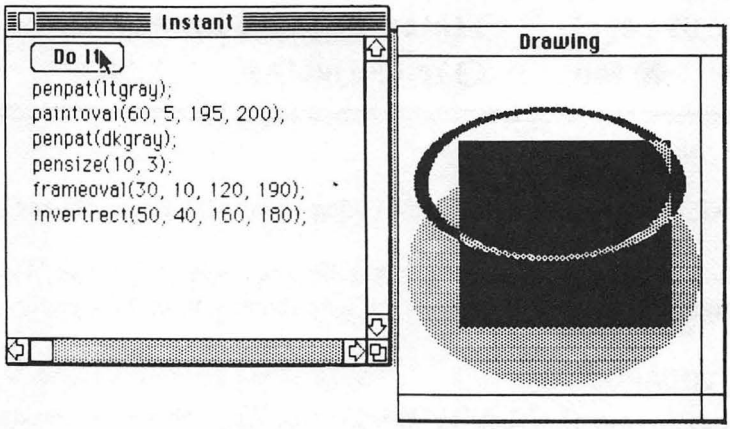
Operations

\+ - * / DIV MOD

Functions

ROUND TRUNC

Chapter 3



Simple Data Types and Variables

We cannot go much farther in Pascal without encountering the rather slippery concepts of data types and variables. Pascal is very formal with regard to data types, and requires us to understand them rather thoroughly. Once we have bitten this bullet, however, some interesting Pascal techniques will become available. In Chapter 4 you will begin to see why computers are powerful problem solvers. You will also, I hope, get a sense of the fascination of computer programming.

TOPICS COVERED IN THIS CHAPTER

- The standard Pascal data types Integer, Real, Char, and Boolean
- The Macintosh Pascal type String
- The Boolean operators >, <, >=, <=, and <>
- The three characteristics of variables: name, value, and type
- The restrictions on the use of Pascal reserved words
- Using the assignment statement to assign values to variables
- Using READLN and READ to accept data from the keyboard
- Displaying prompting messages to help make programs easier to use
- How READLN, READ, and assignment statements react to data and variables of different types

SIMPLE PASCAL DATA TYPES

Computer data is any information that is manipulated by the computer. In the real world, data could be information about bank accounts, temperature readings, or the frequencies of notes produced by a computerized music synthesizer. The text of this book became computer data as I typed it into my Macintosh.

For Pascal, each of these sorts of data has a *type*. Broadly speaking, of course, bank account balances are numbers and the text of this book consists of characters. But Pascal requires that each data item be precisely defined, with a specific type assigned.

Each data type has advantages and disadvantages, and programmers will choose to work with different types of data in differing circumstances. For example, integers are more exact than real numbers for representing quantities; integers are often used for financial operations since no inexactness can be tolerated. Real numbers, on the other hand, are better suited to representing very large or small values, or values that vary on a continuous scale such as measurements; real numbers are used extensively in scientific and engineering applications.

In order to optimize the use of computer resources, different methods are used to store data of different types. The integer 5 is stored differently from the character '5', and both are stored differently from the real number 5.0. We need all three forms of 5: the integer and real for different sorts of mathematical operations, and the character since it may be freely mixed with letters and punctuation in text. However, the fact that 5, 5.0, and '5' are stored differently requires us to carefully use the right version at the right time, so that Pascal can carry out our instructions.

Pascal is a *strongly typed* language, meaning that all data must be assigned a specific type, and that we cannot casually move from one type to another. This strong typing promotes efficient use of computer resources, and also serves two other purposes: it forces us to carefully think out the purpose and use of the data in a program, and it serves to make that purpose clearer in the text of the program.

Pascal establishes four basic data types:

1. Counting numbers: the type Integer
2. Decimal numbers: the type Real
3. Single characters: the type Char
4. True-False: the type Boolean

Macintosh Pascal also provides a String data type for manipulating strings of characters. We have examined integers and reals sufficiently to let them slide for the moment. The types Char, String and Boolean, however, require a little explanation.

The Types Char and String

A data item of type Char consists of exactly one character. Anything you can type on your keyboard could be type Char. To specify a character, simply type it in single quotation marks. For example,

```
writeln('T')
```

would print the letter T. (Yes, I know this looks just like the strings we met back in Chapter 2, but there is a difference: while data of type Char consists of exactly one character, strings can contain long series of characters—more on this later.)

Characters have a feature called *ordinality*. This simply means that characters form a sequence, just as the letters in an alphabet form a sequence. You will not be surprised when I point out that the character "P" comes after "O," or even that "5" comes before "6." But it is also true that "+" comes after "*" and that "[" precedes "/." Pascal provides two functions, SUCC and PRED, that may be used to determine the characters following or preceding a given one. For example,

```
writeln(succ('A'))
```

will print "B," the successor of "A." Similarly,

```
writeln(pred('8'))
```

prints the predecessor of "8" which is, of course, "7."

Were you to experiment, you would find that every character but the last one in the sequence of characters has a successor, and all but the first character has a predecessor. This feature is very handy, since it will later allow us to develop programs that will sort groups of items into alphabetical order.

Should you be curious, Appendix A contains a complete list of the characters used by Macintosh Pascal. In this list, you will notice that each character is associated with a number that defines its order with respect to the other characters. Pascal also provides a function that produces this number, the ORD function.

```
writeln(ord('A'))
```

prints the number "65," which is A's position in the entire set of characters.

Another thing to notice in the list of characters is that capital letters are distinct from lowercase letters. "A" is not the same as "a". This elementary fact will take on significance later.

CHR is a function that complements ORD. CHR will output the character associated with any ordinal value from 0 to 255. Therefore,

```
writeln(chr(65))
```

would print the letter "A".

Originally, the only tool Pascal had for manipulating text was the type Char, which will not work with more than one character at a time. While it is quite possible to function with this limitation, most programmers prefer not to do so. Macintosh Pascal and a few other versions of Pascal incorporate a data type that allows characters to be manipulated as groups, called *strings*. We have seen a number of strings already; they are simply groups of characters set off by single quotation marks.

Characters and strings are similar. Under the proper circumstances, we can use a character where Pascal expects a string. But, just as real numbers cannot appear where Pascal expects an integer, Pascal will not accept a string where a character is called for. We will have to look closely at this before the end of the chapter.

The Type Boolean

In addition to looking at characters, we must look at the type Boolean, which we will begin to use in the next chapter. Data of this type may have only two values: TRUE or FALSE. If you think it odd to consider TRUE and FALSE as values, you will have to broaden your perspective a little. Values are not merely numeric. Value simply refers to the contents of any particular data item, and we shall see that many different sorts of values are possible in Pascal.

The type Boolean exists to that we can make decisions in Pascal. Anytime we ask a true-false question, a Boolean value is produced. For example, we can ask if one number is greater than another like this:

```
writeln(123 > 45)
```

In this case, Pascal will print TRUE. The > is a Boolean operator. Just as + operates on two numbers

to produce a sum, > operates on two numbers to ask the question, "Is the first number greater than the last?" A numeric *expression* produces a numeric value. We can also have Boolean expressions, which produces Boolean values. In the WRITELN statement above "123 > 45" is a Boolean expression.

There are several Boolean operators that you may be interested in at this point:

>	Greater than,
<	Less than,
>=	Greater than or equal to,
<=	Less than or equal to, and
<>	Not equal to.

These operators may also be used on characters. This is how Pascal can decide how to order words when alphabetizing. So, when we use the statement

```
writeln('H' > 'S')
```

Pascal will print FALSE, meaning that "H" does not come before "S." On the other hand,

```
writeln('H' <> 'S')
```

will produce the response TRUE, since it is true that "H" is not the same as "S."

INTRODUCING VARIABLES

A *variable* is a place in the computer's memory that may be used to store data. You may have wondered why the Pascal examples presented so far were rather trivial, simply printing a message or drawing a graphic. One important reason for this simplicity is that we had not yet begun to use variables. In this chapter and especially in the next chapter, we will see how the use of variables amplifies fantastically the things we can do in Pascal.

A variable has three characteristics.

1. It has an identifier (a name).
2. It has a value. That is, a variable represents an item of data.
3. Because it represents data, and all data is typed, a variable has a type.

Here is a trivial program that illustrates these characteristics.

```
program vardemo;
  var
    N : integer;
begin
  N := 3;
  writeln(N);
  writeln(N * N)
end.
```

■ Enter and execute this program. You will find that it prints:

```
3
9
```

The statement "N:=3" is an *assignment statement.* This assignment is used to associate a value with a given variable. The variable is the letter "N," and this statement can be interpreted as saying, "assign the value 3 to the variable identified by N". From here on, whenever N is referenced, Pascal will retrieve the value assigned to it and substitute the value in its place.

So, these program statements:

```
writeln(N);
writeln(N * N)
```

have nothing to do with printing the letter "N." Instead, the WRITELN statements are equivalent to

```
writeln(3)
writeln(3 * 3)
```

The *value* of the variable N, therefore, is 3. This accounts for one characteristic of the variable. The second characteristic is the identifier N itself. Each variable must have an identifying name, and N is one example of an acceptable one.

Identifiers may contain letters; they may also contain numeric characters, provided the identifier does not begin with a number. However, the only punctuation they may contain is the underscore character. Variable identifiers may be of any length up to 255 characters. These are exactly the same restrictions that are placed on program identifiers, as we saw in the last chapter. In fact, identifiers with these restrictions will be encountered quite often in our dealings with Pascal.

In addition to value and name, a variable must have a type. The type is determined by the second and third lines of the program:

```
var
 N : integer;
```

These lines form the *variable declaration part* of the program, also called the *var block.* We will see that Pascal reserves the space between the program heading and the beginning of the statement part of the program for defining important program features. Every variable that will appear in the program must be declared in this variable declaration section of the program. The beginning of this section is marked by the word VAR.

This declaration says two things: that "N" will be used as a variable name, and that the variable identified by "N" can take on only integer values.

Now we should ask "What happens if any of the three characteristics of a variable is violated?" We saw before that the best technique for answering such a question is to introduce an error into a program that is known to work properly. Let's try to assign a real value to "N".

■ Edit the assignment statement in the program VARDEMO to look like this:

```
N := 3.456;
```

■ Execute the program, and observe the error messages. If you have followed the discussion so far, the results will not surprise you. A type mismatch has occurred.

■ Restore the assignment statement so that it reads "N:=3". We will be introducing other errors and we want to correct this one.

■ To see what happens when an illegal identifier is used, change the identifier in the variable declaration from "N" to "N%". The variable block will now look like this:

```
var
  N% : integer;
```

■ Execute the program. When you do, Pascal will reformat the line like this to indicate an error:

```
var
  N%
  : integer;
```

In addition the Bug box declares, "Variable declaration expected after a VAR." Since no legal identifier was found, Pascal did not know what to do with the information it found in the variable declaration section.

One last error should be investigated. What happens if we attempt to use a variable for which a type has not been declared. To investigate this, remove the entire variable declaration section. The program now looks like this:

```
program vardemo;
begin                                    {remove var block}
  N := 3;
  writeln(N);
  writeln(N * N)
end.
```

This time, your attempt to execute the program will produce this message: "The name 'N' appears to be an undeclared identifier." You have now seen the three ways to mess up a variable: by forgetting to declare its type, by using an improper name, or by attempting to assign a mismatched value. Variable mismatches can be tricky. Sometimes they will be accepted; sometimes they will not. We will have to sort this out before the end of the chapter.

IDENTIFIERS AND RESERVED WORDS

Here are a few more precautions regarding the selection of identifiers. Are there any words that cannot be used?

■ To see, try typing this program:

```
program gotcha;
  var
    div : integer;
begin
  writeln(div)
end.
```

As usual, Macintosh Pascal will reformat your text as you enter it. The reformatting, however, will display your program like this:

```
program gotcha;
  var
    div : integer;
begin
  writeln(div)
end.
```

DIV, you should recall, is the Pascal operator that performs division on integers. If you examine any program that contains DIV you will notice that it is printed in bold type. This is also true of BEGIN, END, VAR, and a number of other words. Earlier I described these words as *reserved words,* and we can now begin to see why that term is used. In GOTCHA, we attempted to define DIV as a variable name. This produced a syntax error, as reflected by the conversion of large portions of the program into outline type.

Execute the program to see Pascal's message about this error. It turns out that Pascal expects a variable declaration in the VAR block, but in this case none was found. The reserved word DIV, found where a variable was expected, is not acceptable. Any word that Macintosh Pascal prints in bold type is reserved and cannot be redefined. A list of these words is included in the Appendix.

Is this true of every Pascal word? What about the names of built-in procedures, such as WRITELN?

■ Enter this program to test the possibility of using WRITELN as a variable name.

```
program gotcha_again;
  var
    writeln : integer;
begin
  writeln := 5;
  write(writeln)
end.
```

■ Execute this program. You will find that Pascal has absolutely no problem with it. WRITELN is not a reserved word, and we can redefine it if we wish. But, "there ain't no such thing as a free lunch." Modify the program by adding one more line:

```
begin
  writeln := 5;
  write(writeln);                    {add semicolon}
  writeln(10)                        {new statement}
end.
```

this time, there definitely is a problem. In fact, Pascal announces that, "There is no procedure named writeln." When we defined WRITELN as a variable identifier, its former definition was voided.

So, Pascal allows us to use many of its built-in words, in fact most of them, as user-defined identifiers. However, it is important not to confuse liberty with license. If we take advantage of this capability, we can make our lives very complicated when inadvertant conflicts occur.

ASSIGNING VALUES FROM THE KEYBOARD: THE READLN STATEMENT

The assignment statement is one of two ways to associate a value with a variable. It allows us

to assign values to variables based upon data contained within the program. Here is a program that uses the assignment statement in the process of determining the average of three numbers.

■ Enter the program.

```
program average;
var
  n1 , n2 , n3: integer;
begin
  n1 := 17;
  n2 := 5;
  n3 := 28;
  writeln((n1 + n2 + n3) / 3 : 6 : 2)
end.
```

The VAR section allows us to make multiple variable assignments for a given type by separating the variable names with commas. This variable declaration section has declared the variables N1, N2, and N3 to be of type Integer.

The program operates simply by assigning a value to each variable. It then adds the variables together and divides by 3. Recall from Chapter 1 that we must use parentheses to group the data for proper sequencing of the arithmetic operations.

What happens if we wish to have the program average three different numbers? One approach would be to edit the assignment statements in the program to contain three new data items. But this would be cumbersome if we wished to average many sets of numbers. It would be more convenient to have the program simply request three numbers, which could be typed in at the keyboard. These numbers could be stored in variables, and the average could be calculated as we have already done.

To accomplish this, we must call on a second method of assigning values to variables: the READLN statement.

■ Edit the program to incorporate the READLN statement:

```
program average;
var
  n1 , n2 , n3: integer;
begin
  readln(n1);                          {change assignment}
  readln(n2);                          {statements to}
  readln(n3);                          {READLN statements}
  writeln((n1 + n2 + n3) / 3 : 6 : 2)
end.
```

The program will now operate in a very different way.

■ Choose Go and observe the screen. The only change you will see is that the text insertion pointed has appeared in the Text window. The insertion point always indicates the Pascal expects us to type something on the keyboard.

■ Type an Integer. Your typing will take place in the text window.

■ Nothing more will happen until you press the Return key. Do so now. This will cause the cursor to move down a line.

■ Type another number and press the Return key.

■ Then do the same one more time. At last, Pascal will print the average of your three numbers.

The READLN statement served to assign the values you typed at the keyboard to the three variables in the program. This makes this version of the program more versatile than the first, which used the assignment statements.

READLN is used to accept information from the keyboard. Information that is entered into the computer is called *input.* Similarly, information that is given out by the computer is called *output.* So, READLN and WRITELN may be viewed as complementary procedures: one is responsible for program text input and the other is responsible for program text output.

With rare exceptions, any program that solicits user inputs should display a message indicating what is expected. This can prevent quite a bit of confusion. Useful prompting messages for the averaging program could be incorporated like this:

```
program average;
var
  n1 , n2 , n3: integer;
begin
  write('Type first integer: ');                    {new statement}
  readln(n1);
  write('Type next integer: ');                     {new statement}
  readln(n2);
  write('Type last integer: ');                     {new statement}
  readln(n3);                                      {modify next line}
  writeln('The average is: ', (n1 + n2 + n3) / 3 : 6 : 2)
end.
```

The use of WRITE causes the user's numbers to be typed immediately after the prompting message. This is often a more attractive and meaningful way of arranging things. While I was at it, I added a message to label the final average. When a program is intended for use by others, it is desirable to make things as clear as possible. Printed information should be labeled to indicate its significance.

A CLOSER LOOK AT TYPE MATCHING

It is important to properly match data and variable types, but Pascal will occasionally grant us some latitude. Learning when you can "cheat" is an important part of your learning process. Let us perform some experiments.

■ Create the following program:

```
program vartest;
var
  X : real;
begin
  readln(X);
  writeln(X)
end.
```

I have violated my own advice. No messages will prompt you when you are typing. But we will modify this program several times and I wanted to save you some editing.

■ Execute the program several times. Each time respond with a different type of input, such as the following three:

```
123          45.678          abcd
```

How does Pascal respond in each case? You found that either an integer or a real number would be accepted. The integer was accepted, but when it was written, it appeared in real form. Pascal can assign an Integer to a Real variable, but the data will always be stored as a real number.

When you tried to type letters, however, Pascal would not accept them. Pascal simply ignored your letters, reinforcing the prohibition with a few beeps.

■ Change the type of X by replacing the "real" with "integer." Now try the same three inputs. What are your results?

■ Change "integer" to "char", and try the inputs again. Examine the printed results and try to explain them.

Variables of type Char may contain only one character. While the READLN statement appeared to accept everything you typed, only the first character was actually stored into the variable.

One catch of using READLN with Char type variables is that you cannot correct a typing error.

■ Execute the program again. After you have typed the first character, erase it by pressing the Backspace key. The character will go away. Type something different and press Return. Notice that the original character is printed, not the character you typed after pressing the Backspace key. Obviously, READLN is not a particularly good way to input data to Char variables. Later we will learn some better ways to input character data.

■ Finally, change "char" to "string", and try all of the inputs. Examine the printed results and try to explain them.

A variation on READLN is the procedure READ. There are subtle differences between these two procedures that make each preferable in different circumstances.

■ Change READLN to READ in VARTEST.

■ Perform the same tests that you made with READLN. Pay particular attention to the behavior of READ when the variable has been typed as Integer or Char.

To complete the picture, we must try out these mismatches using the assignment statement. We saw earlier that real numbers cannot be assigned to integer variables. I will not be spoiling much if I tell you that integers may always be assigned to real variables. So, the big mystery regards data and variables of the type Char.

■ Change the READLN statement in VARTEST to this assignment statement:

```
X := '6';
```

■ Try out vartest for several combinations of variable and data types. Do this by editing both the type in the VAR block and the value that is assigned to X in the assignment statement.

■ Try to assign an integer and a real to a Char variable.

■ Try to assign character data to a Real or an Integer variable. Which combinations, if any, produce no error messages?

You will find that the assignment statement always produces an error message when an unacceptable match is attempted. READLN is a little more forgiving since it will at least ignore improper data types. (This is true in Macintosh Pascal, but it is definitely not true in most other versions of

Pascal.) When the assignment statement encounters an error, however, the program is always terminated with an error.

From these experiments, we can make several statements:

- Integers may always be assigned to Real variables. The data will always be converted to real form.
- Real numbers may never be assigned to Integer variables. We have seen type mismatch errors in this situation before. The READLN statement simply refuses to accept the decimal portion of the number. The READ statement terminates input when anything but a digit is typed.
- In fact, READ terminates input whenever an illegal character is typed. A letter will end input when the variable is real, for example.
- READ will accept only one character when the input variable is of type Char. A Return is never required when using READ with a Char variable.
- Characters will never be accepted into Integer or Real variables.
- READLN will accept the first digit of an integer or a real number into a Char variable.
- Attempts to match integers or real numbers to a Char variable by using the assignment statement will always result in an error condition.

YOUR PASCAL VOCABULARY

You now know the following Pascal words. Words that were new in this chapter are printed in boldface type.

Reserved Words

PROGRAM BEGIN END **VAR**

Statement Types

Assignment (:=)

Data Types

BOOLEAN **CHAR** **INTEGER** **REAL**

STRING

Procedures

READ **READLN**

WRITE WRITELN

NOTE

FRAMERECT PAINTRECT FRAMEOVAL PAINTOVAL

INVERTRECT INVERTOVAL PENPAT PENSIZE

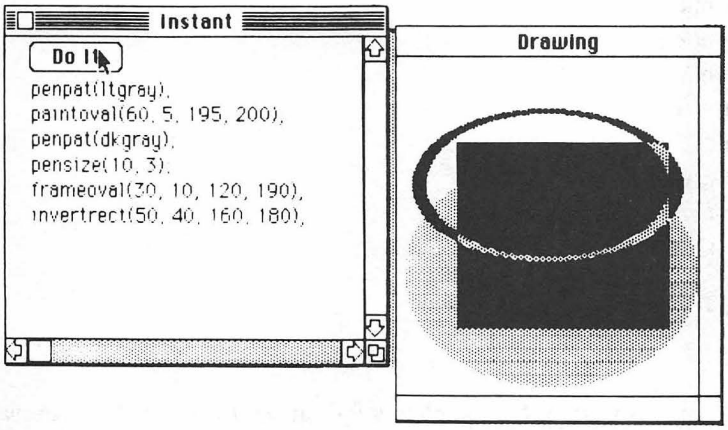
Operations

\+ - * / **DIV** **MOD**

Functions

ROUND TRUNC

Chapter 4



Control Statements Part 1: The FOR Statement

Probably every complex program you will encounter will employ some form of repetition. Computers are especially well adapted to repetition since they are fast and highly accurate. Many tasks that we would have trouble carrying out manually become child's play when properly programmed.

Pascal provides three distinct ways of controlling repetitive processes. Using these, we will at last be able to write programs that perform complex tasks. The first repetition control statement we will consider is the FOR statement.

TOPICS COVERED IN THIS CHAPTER

- Repetition with the FOR..TO statement
- The relationship of the FOR statement control variables and the control values
- The terms looping, repetition, and iteration
- Counting down with DOWNTO
- Counting by twos and other intervals
- The compound statement
- A FOR statement error that results from modifying the value of the control variable

THE FOR STATEMENT

At this point you would have to do quite a bit of typing to write a program that would count from 1 to 10. The only strategy you really have available is to utilize ten WRITELN statements, something like this:

```
program count;
begin
  writeln(1);
  writeln(2);
  writeln(3);
  writeln(4);
  writeln(5);
  writeln(6);
  writeln(7);
  writeln(8);
  writeln(9);
  writeln(10)
end.
```

This approach is tedious at best, but it completely falls apart if we want to count, say, from 1 to 1000. Since counting is such a common computer activity, that there simply must be a better way of doing it.

Here is a program that may be easily adapted to count through any range of numbers.

■ Type it in and execute it. Then we will look at what makes it tick.

```
program count;
 var
   counter : integer;
begin
 for counter := 1 to 10 do
   writeln(counter)
end.
```

This program uses the FOR statement, which has the following general structure:

for *variable* := *initial value* **to** *final value* **do** *statement*

Notice that the FOR statement contains three new reserved words: FOR, TO, and DO. Let us examine each part of the statement structure:

- The FOR simply marks the beginning of the statement.
- A control variable follows, in this case COUNTER. This variable must be declared in the VAR section, and must be of an *ordinal* type. The ordinal types we know now are Integer and Char.
- The assignment operator (:=) indicates that COUNTER will be assigned a value.
- An initial value, which must be of the same type as the control variable, follows.
- The word TO informs Pascal that it is to count from the initial value up to the final value.
- A final value, also of the same type as the control variable, follows.
- The word DO marks the end of the control section of the statement.
- A program statement containing the action that is to be performed as part of the FOR statement, follows DO. A FOR statement always contains exactly one executable statement. As we shall see, this forces us to exercise a certain degree of care when we punctuate the statement.

The FOR statement simply uses the control variable to step through the specified range, one value at a time. The control variable takes on each value in this range. Our demonstration program simply printed this value each time the WRITELN statement was executed.

The range indicators could be any two integers. The same program can as easily count from negative 512 to 697 if the FOR statement is edited like this:

```
for counter := -512 to 697 do
```

Before we examine this statement any closer, I would like you to try a few experiments:

1. Test the program to see what happens when COUNTER is declared to be real.
2. See what happens if the final value is less than the initial value.
3. Try putting a semicolon after DO. How many times does the WRITELN statement execute? What is printed?

The effect of the change in experiment 1 should, by now, not be a surprise to you. It is a simple type mismatch.

The second experiment demonstrates that this arrangement of the FOR statement cannot count down. If the second range value is less than the first, the statement does not execute. In that case, the sample program will not do anything visible.

Experiment 3 actually produces two abnormalities. First, the WRITELN statement executed only once. This illustrates an important feature of the FOR statement, but we will have to embark on an involved explanation in order to understand exactly what that feature is. The problem has to do with the way the FOR statement determines which actions it should repeat.

Pascal has two ways to mark the end of a statement. A semicolon may be used; we have already seen that a semicolon must be used to separate consecutive statements in a program. The exception is a statement that is followed by an END. In that case, the END serves to close off the statement, and no semicolon is needed.

So, the semicolon after DO marks the end of a Pascal statement, and the only statement it can end is the FOR statement. But, there are no instructions to be performed! What is going on?

By placing the semicolon immediately following DO, we introduced a Pascal feature called the *null statement.* The null statement is a statement that does nothing. Pascal uses it to permit us a little latitude in the use of semicolons: under certain circumstances, an extra semicolon will not produce an error condition.

The second abnormality is less obvious. Because of the semicolon placement, the FOR statement does not repeat the WRITELN statement. Instead, it is the null statement that is repeated. The FOR statement does nothing ten times.

After these ten repetitions, Pascal continues the program by executing the WRITELN statement. However, the value printed will make little sense. We might logically expect COUNTER to still have the last value that was assigned to it by the FOR statement. If the statement were:

```
for counter := 1 to 10 do;
```

What value would you expect COUNTER to have when the statement was completed? You might reasonably anticipate that COUNTER would equal 10 or 11. So where did 362 (or perhaps you got another equally mysterious number) come from?

When a FOR statement ends, an odd thing happens to the counting variable. In Pascal terminology, the variable's value is said to be *undefined.* This means that there is no rule that allows us to anticipate

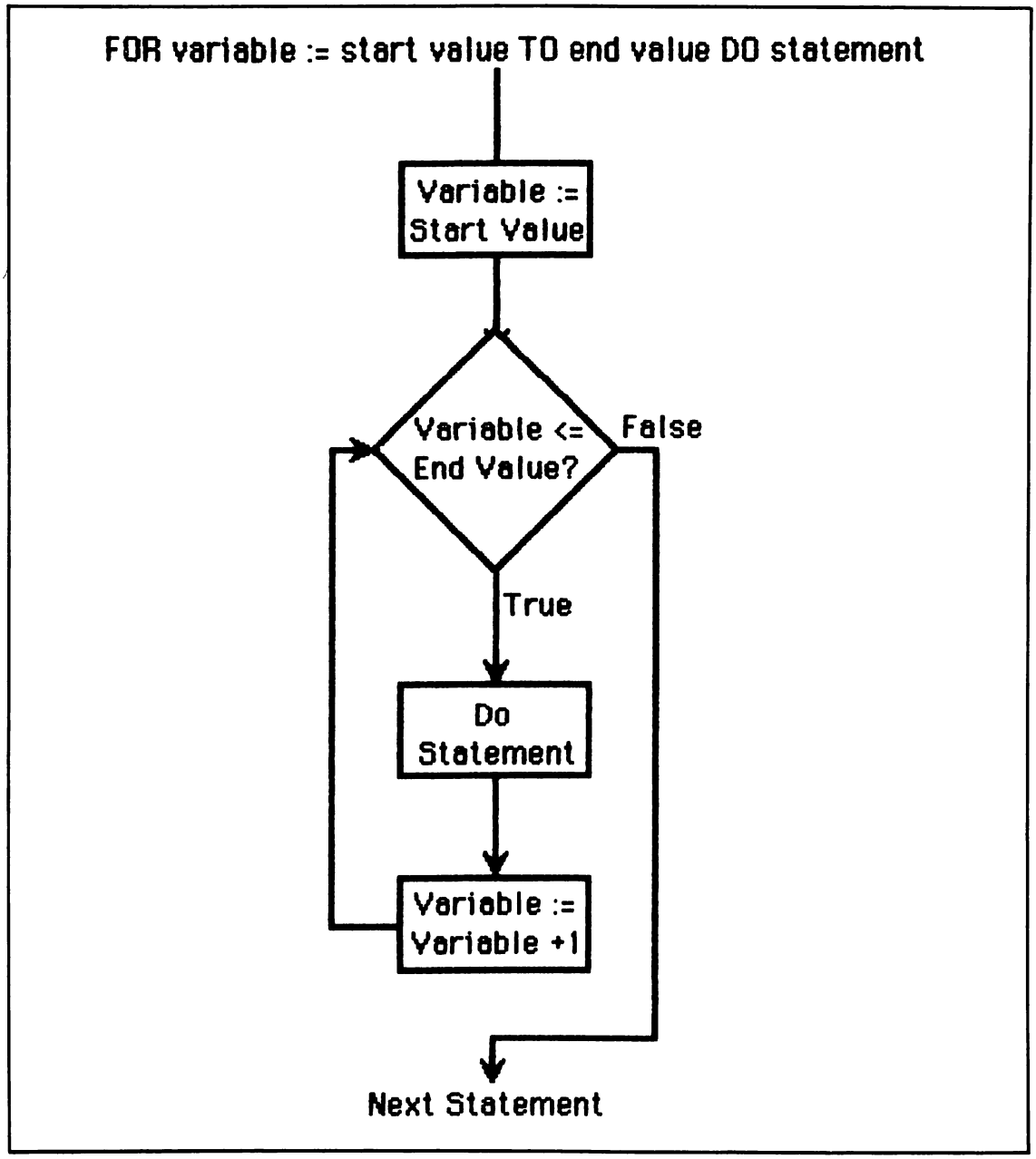


Fig. 4-1. The flow of control in a FOR loop.

the variable's value. The variable itself is still declared and typed, but it has no official value. In my experience, the counter variable will have the value 362, regardless of the upper limit on the FOR statement. You may encounter other values, however, since no value is guaranteed.

Let's review the operation of the FOR statement with the help of Fig. 4-1. There are four distinct steps in the execution of this statement:

1. Initialization. The upper and lower counting limits are defined. The counter variable is assigned the value specified by the starting value.
2. Testing. The variable is tested to see if its value exceeds the specified upper value. If the variable is within range, go to step three. If the variable exceeds the upper limit, go to step five.
3. Statement execution. If the variable is within the allowed range, the statement following DO is executed. Continue with step four.
4. Incrementing the variable. *Increment* is a fancy term for "adding one to the variable." Then go back to step two and test the variable.
5. Exiting. If the test in step two indicates that the variable has exceeded the allowable range, the FOR statement is exited, and execution continues with the next statement in the program.

Figure 4-1 illustrates why repetitive structures are often called *loops*. Diagrams of these statements often show program control going around in circles. We will often refer to the FOR statement as a *FOR loop*.

Actually, three terms are used fairly interchangeably: looping, repetition, and iteration. You will frequently encounter the word *iteration* in books about computer programming. It means essentially the same thing as repetition.

Step two explains why the second limit value must be greater than the first. If the FOR statement starts out like this,

```
for counter := 10 to 5 do
```

then COUNTER will be initialized with a value of 10. But then, testing will indicate that COUNTER exceeds 5, the upper limit. Because it follows the rules in step two, the FOR statement exits without even once performing the statement in step three.

If FOR statements always count by ones, how could we write a FOR loop that would print only even numbers, say from 2 to 10? Actually, the solution is pretty simple.

■ Make a few modifications and try the program again:

```
program count;
  var
    counter : integer;
begin
  for counter := 1 to 5 do                    {change final value}
    writeln(counter * 2)                      {add multiplication}
end.
```

That wasn't so hard. Now then, how would you count by odd numbers? Try to discover the modifications necessary to count the odd numbers from 1 to 11.

What if you would like to count down? One technique is to replace TO with DOWNTO.

■ Edit the appropriate lines in the program count like this:

```
for counter := 20 downto 1 do          {change TO to DOWNTO}
  writeln(counter)                     {remove mulitplication}
```

If you tried this modification, the function of DOWNTO should be obvious. As you would expect, when DOWNTO is used, the final value in the FOR statement must be less than the initial value for the statement to function properly.

Knowing what you know, how would you count down by twos? See whether you can discover a solution to this puzzle.

But enough dry examples. Let's see if we can have some fun.

DEMONSTRATION PROGRAMS WITH FOR LOOPS

For a first project, let's develop a program that draws a series of boxes on the screen. This program starts all of the boxes in the upper left corner of the Drawing window, and draws boxes of increasing size proceeding diagonally across the window. For each value of COUNT in the following program, FRAMERECT draws a rectangle with its top and left sides at 0 and with its bottom and right sides at COUNT. So, if COUNT has a value of 50, the bottom and right side will each be at 50.

```
program lotsaboxes;
 var
   count : integer;
begin
 for count := 1 to 100 do
   framerect(0, 0, count, count )
end.
```

■ Enter and execute the program. You will find it pretty dull, since the end result is just a filled black square, but there is a way to make it more interesting: put some white space between the squares.

■ Make these modifications and run the program. You should find this version of the drawing considerably more pleasing.

```
program lotsaboxes;
 var
   count : integer;
begin
 for count := 1 to 50 do                    {change final value}
   framerect(0, 0,count* 2, count * 2)  {add multiplication}
end.
```

Let's add the statements to draw a similar square that originates in the lower-right corner of the window. Again, let's arrange things so that the squares increase in size. The first two FRAMERECT statements would use these values:

```
framerect(198, 198, 200, 200)
```

and

```
framerect(196, 196, 200, 200)
```

Since the values are decreasing in size, it seems logical to use a FOR..DOWNTO statement to manage things.

■ Make the indicated changes and execute the program.

```
program lotsaboxes;
  var
    count : integer;
begin
  for count := 1 to 50 do
    framerect(0, 0,count * 2, count * 2 );  {add semicolon}
  for count := 99 downto 50 do              {new for statement}
    framerect(count * 2, count * 2, 200, 200)
end.
```

That is one way to count down by 2s. The FOR..DOWNTO, together with the multiplication that take place within FRAMERECT, functions to count down by 2s from 198 to 100. This gives us the values we required.

As the program stands now, it completely finishes the upper left square before beginning the bottom right one. Is it possible to draw them at about the same time? Can we work it so that one FOR statement controls two FRAMERECT statements? To do this we must introduce the Pascal *compound statement.* Here is a version of the program that uses one.

■ Enter and execute the program. An explanation of the new BEGIN and END follows.

```
program lotsaboxes;
  var
    count : integer;
begin
  for count := 1 to 50 do
    begin
      framerect(0, 0,count * 2, count * 2 );
      framerect(200 - count * 2, 200 - count * 2, 200, 200)
    end
end.
```

Recall that BEGIN and END are not statements. Rather they serve to mark the beginnings and ends of program sections. Every program we write uses them to mark the beginning and end of the statement part of a program. But Pascal uses them to mark off other program sections as well.

This part of the program:

```
begin
  framerect(0, 0,count * 2, count * 2 );
  framerect(200 - count * 2, 200 - count * 2, 200, 200)
end
```

will be treated by the FOR statement as a single statement. It is held together by the BEGIN and END. Such statements are called compound statements. The effect of this is that both FRAMERECT statements are included in the statement portion of the FOR loop. Only one statement can be included in the statement part of a FOR loop, but that can be a compound statement, which performs multiple tasks.

Notice the use of semicolons in this example. A semicolon must be used to separate the two pro-

cedure statements within the compound statement. However, each END is a sufficient statement terminator and no semicolons precede them. You need never place a semicolon before an END, although, it you experiment you will find that placing one there will do no harm. By placing a semicolon immediately ahead of an END, you would introduce a null statement between the semicolon and the END. This causes no problems, since the null statement does not do anything, but in extreme cases the extra null statements might slow a program down.

There is another feature of interest in this program. The drawing of both squares is controlled by the same IF statement, which counts from 1 to 50. But the lower right set of squares is being drawn by counting down! By applying subtraction to the counter values in the IF statement, the same statement can control both upward and downward counting.

For example, if COUNT has a value of 2, the first parameter for the second FRAMERECT statement will be 200 – 2 * 2, which is 196. When COUNT is 3, the same parameter will have a value of 200 – 3 * 2, which is 194. Thus, the parameter decreases in value as COUNT increases. DOWNTO is not essential; we can get along quite well with TO.

Let's complete the picture by drawing two more sets of squares in the remaining corners. Unfortunately, the coordinates do not work as neatly in these corners. Consider the square we wish to draw in the lower left corner of the Drawing window. This corner has coordinates of 0,200. All squares in this corner will have their left sides at 0 and their bottoms at 200. Again, we will start by drawing the smallest square first. Remembering that the parameters define the top, left, bottom, and right sides in that order, the first FRAMERECT statement should use these parameters:

```
framerect(198, 0, 200, 2)
```

The second statement will use these values:

```
framerect(196, 0, 200, 4)
```

You will notice that one value is increasing, while the other is decreasing. A similar condition will be encountered in the upper right corner. This will make the subtraction and multiplication situation a bit confusing. Let's try to find a simple way to solve the problem.

We will do that by introducing variables to keep track of the counting down and the counting up.

■ Let's start by modifying the present program to use the new variables COUNTUP and COUNTDOWN:

```
program lotsaboxes;
  var
    count, countup, countdown : integer;          {2 new variables}
begin
  for count := 1 to 50 do
    begin
      countup := count * 2                                  {new line}
      countdown := 200 - count * 2;                         {new line}
      framerect(0, 0, countup, countup );    {introduce variables}
      framerect(countdown, countdown, 200, 200)   {same here}
    end
end.
```

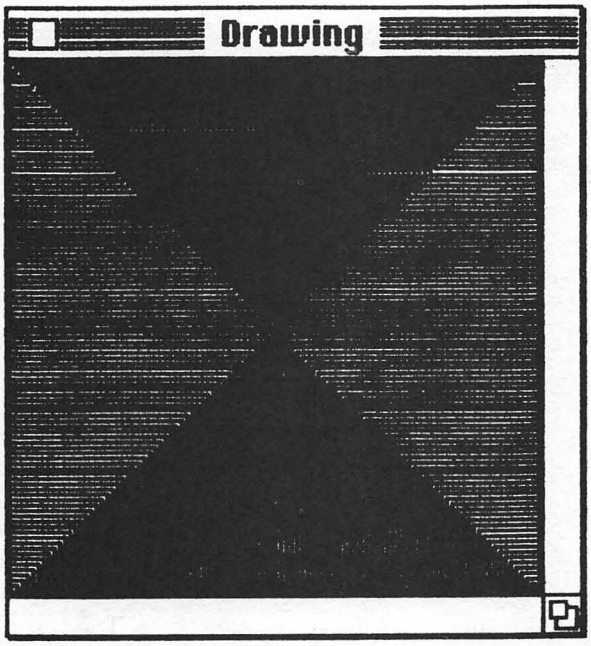


Fig. 4-2. The drawing created by LOTSABOXES.

Notice that the values of COUNT and COUNTDOWN are calculated only once in each pass through the loop. This is more efficient than calculating each value twice within the FRAMERECT parameter lists. This considerably simplifies the FRAMERECT statements. Also, the choice of variable names makes it clear which parameters are increasing in value and which are decreasing. This makes it considerably easier to write clear statements that draw the other two remaining corners. These statements are incorporated into this final version of the program.

■ Add the remaining statements and try out the final program. It creates the design shown in Fig. 4-2.

```
program lotsaboxes;
  var
     count, countup, countdown : integer;
begin
  for count := 1 to 50 do
     begin
        countup := count * 2;
        countdown := 200 - count * 2;
        framerect(0, 0, countup, countup );
        framerect(countdown, countdown, 200,  200);
        framerect(countdown, 0, 200, 0, countup);          {new}
        framerect(0, countdown, countup, 200)  {new statement}
     end
end.
```

Let's examine a few more programs that use FOR loops.

■ Enter and execute this program:

```
program movearound;
  var
    count, countup, countdown : integer;
begin
  for count := 0 to 50 do
    begin
      countup := count * 4;
      countdown := 200 - countup;
      moveto(countup, 0);
      lineto(200, countup);
      lineto(countdown, 200);
      lineto(0, countdown);
      lineto(countup, 0)
    end
end.
```

This program uses multiplication to allow the end points of the lines to change by 4s. The value of the loop counter is multiplied by 4 and stored in the variable COUNTUP. Again, the results of calculations are stored in variables to simplify the parameters for the graphics statements.

Two new graphics statements have been introduced: MOVETO and LINETO. MOVETO locates the drawing pointer at the specified point, but no drawing is performed. LINETO also moves the pointer, but it draws a line between the previous point and the new one.

Since the first value of COUNT is 0, the first value of COUNTUP is also 0; MOVEAROUND starts by positioning the drawing pointer to point, 0 in the Drawing window. Line drawing takes place within the loop. The first time through the loop, COUNT has the value of 0, COUNTDOWN has a value of 200, and these four lines are drawn:

0,0 to 200,0	200,200 to 0 ,200
200,0 to 200 ,200	0,200 back to 0,0

Each time through the loop, the line endpoints move around the window 4 more points. The final result is shown in Fig. 4-3.

■ As a final example, enter and execute this simple program. It reveals an interesting way to control a FOR loop:

```
program abcs;
  var
    ch : char;
begin
  for ch := 'a' to 'z' do
    writeln(ch)
end.
```

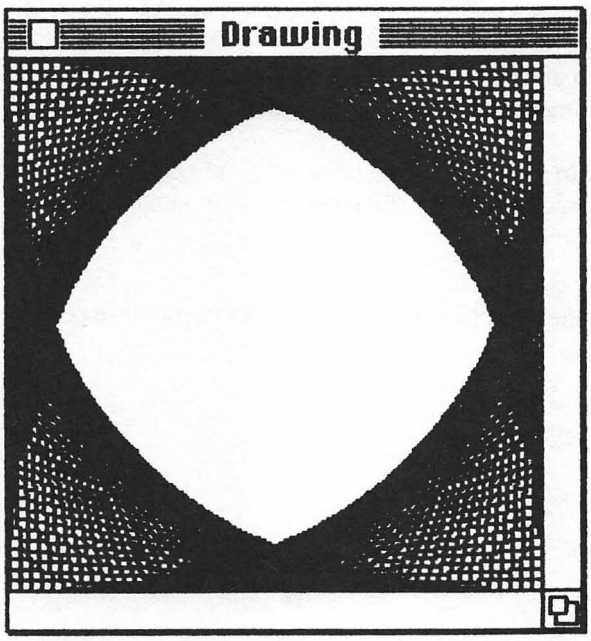


Fig. 4-3. The drawing created by MOVEAROUND.

What the program does is probably not too surprising. However, we see from this simple program that a loop can count by characters as well as by integers. The program could also have been written to work from 'Z' DOWNTO 'A'. Later, you will see that there are many ways to control loops in addition to integers.

A FOR STATEMENT BUG

There may come a time when you will be tempted to perform some calculation that will alter the value of the loop counter. For example, it would appear that the following program might instruct Pascal to count by twos, since the loop counter is multiplied by two each time.

■ When you enter the program and execute it, you will produce one of the more interesting error messages you are ever likely to see.

```
program countby2;
  var
    COUNTER : integer;
begin
  for COUNTER := 1 to 5 do
    begin
      COUNTER := COUNTER * 2;
      writeln(COUNTER)
    end
end.
```

I do not think that Pascal could have indicated its displeasure any more clearly, but what did we do to provoke such a display of outrage?

Only the loop control statement is allowed to adjust the value of a FOR loop counter variable. When we tried to change its value with an assignment statement, we confused matters unalterably, and Pascal called things off.

And yet, there are times when we would like to do just what we tried in this program. For those times, Pascal provides WHILE and REPEAT loops, which are the subjects of the next chapter.

YOUR PASCAL VOCABULARY

You currently know these Pascal words. The ones that were introduced in this chapter are printed in bold face type.

Reserved Words

PROGRAM	BEGIN	END	VAR
DO			

Statement Types

Assignment (:=)	**Compound**		
FOR..TO	**FOR..DOWNTO**		

Data Types

BOOLEAN	CHAR	INTEGER	REAL
STRING			

Procedures

READ	READLN		
WRITE	WRITELN		
NOTE			
FRAMERECT	PAINTRECT	FRAMEOVAL	PAINTOVAL
INVERTRECT		INVERTOVAL	
MOVETO	**LINETO**	PENPAT	PENSIZE

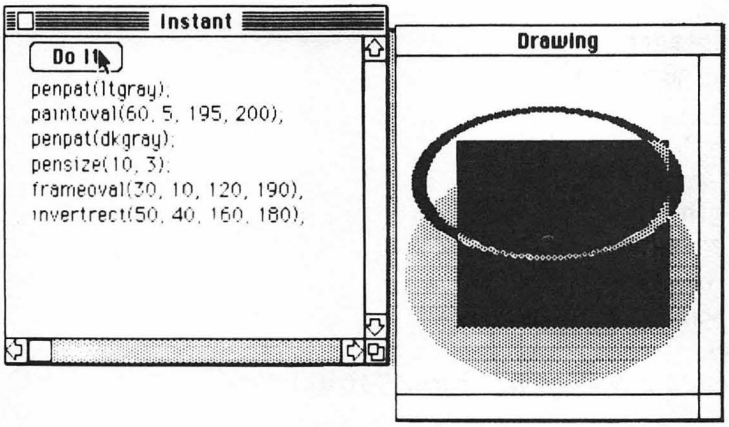
Operations

+	-	*	/
DIV	MOD		

Functions

ROUND	TRUNC

Chapter 5



Control Statements Part 2: WHILE and REPEAT Statements

FOR statements are very useful, but they do have their limitations. In this chapter, we will see how WHILE and REPEAT statements work to do things we cannot easily do with FOR loops.

TOPICS COVERED IN THIS CHAPTER

- Looping with the WHILE statement
- Initializing variables
- Reading the location of the mouse using GETMOUSE
- Using BUTTON to determine if the mouse button has been pressed
- Using nested loops
- Looping with REPEAT statements
- Using the MacPascal Observe window
- Using stops to halt program execution
- Correcting off-by-one errors
- Comparison of the WHILE and REPEAT loops
- Changing the pen drawing style with PENMODE
- Using PATXOR to permit drawing and erasure of shapes without destroying the background

WHAT FOR STATEMENTS CAN'T DO

FOR loops are very useful, but they have their limitations. To illustrate one such limitation, let's develop a program that asks for several numbers and then prints the average of the numbers. This program, which asks for and adds five numbers, is fairly simple.

■ Enter the program and try it.

```
program Average;
  var
    i : integer;
    sum, num : real;
begin
  sum := 0;
  for i := 1 to 5 do
    begin
      write('Type a number: ');
      readln(num);
      sum := sum + num
    end;
  writeln('The average is: ', sum / 5:10:4)
end.
```

This FOR statement simply counts from 1 to 5. Each time through the loop, the program asks for a number. This is added to the variable SUM at the end of the loop. We have not seen a statement like "SUM := SUM + NUM" before, and it calls for some examination.

In English this statement says, "Take the current value stored in the variable identified by SUM and add it to the value in NUM. Then store the result back in SUM." (Thus, SUM represents two different values within the same assignment statement. On the left side of the assignment operator, SUM represents the variable as it will be *after* the statement is completed. On the right side of the operator, SUM represents the *current* value of the variable.) By performing this action, a running total of the numbers is kept in SUM.

During the first time through the loop, this presents a hitch. What is the current value of SUM? Unless we inform it, Pascal does not know. So before the loop started, we *initialized* SUM by assigning it a value of 0. Unless a variable has been assigned a value by an assignment or a READLN statement, its value is *undefined,* which means it could be anything or nothing; Pascal has no rule for stating what the value of an undefined variable is. Often unpredictable actions by programs may be tracked back to the fact that variables were not initialized. Try the program as shown. Then execute it again after removing the "SUM := 0" statement, entering the same numbers both times. The result should be obviously incorrect. This will confirm the need to initialize the variable to 0 in order to obtain correct results.

After testing the program with and without variable initialization, restore it to its original condition.

How can we modify the program so that it will accept different quantities of inputs? One way to do this would be to ask the user how many numbers will be entered.

■ Make these changes to the program:

```
program Average;
  var
    i, n : integer;                    {new variable}
    sum, num : real;
begin
  sum := 0;
```

```
  write('How many numbers do you wish to add? ');    {new line}
  readln(n);                                         {new line}
  for i := 1 to n do                                 {change 5 to n}
    begin
      write('Type a number: ');
      readln(num);
      sum := sum + num
    end;
  writeln('The average is: ', sum / n:10:4)          {change 5 to n}
end.
```

This method works well when the user of the program knows exactly how many numbers will be entered prior to program execution. But, suppose you have several hundred different numbers to enter. Are you going to count them by hand before you start? Doesn't that open the possibility of miscounting? And anyway, isn't counting what computers are good for? Why should we do the computer's work for it?

THE WHILE STATEMENT

It would be convenient to have a way to continue number entry until the user typed in a final value. One method of doing this is to use the WHILE statement.

- Here is an example for you to try:

```
program Average_two;
  var
    count : integer;
    sum, num : real;
begin
  sum := 0;
  count := 0;
  write('Type a number: ');
  readln(num);
  while num <> 0 do
  begin
    sum := sum + num;
    count := count + 1;
    write('Type a number: ');
    readln(num)
  end;
  writeln('The average is: ', sum / count : 10 : 4)
end.
```

Immediately following WHILE is a condition, in this case "NUM < > 0". This is an example of a Boolean expression; that is, an expression that produces only the values of TRUE or FALSE. We encountered Boolean expressions in Chapter 3, but this is the first instance of a practical use.

The WHILE statement has the following general form:

while *Boolean expression* **do** *statement*

If the Boolean expression is TRUE, Pascal will perform the statement that follows. This can be a single statement or a compound statement, as in the sample program. Let's examine the operation of this program step by step.

1. The variables SUM and COUNT are initialized to zero.
2. A message is displayed and the user is asked to type a number. This number is stored by READLN into the variable NUM.
3. WHILE tests to see whether or not NUM is equal to zero. If any value other than zero was typed, Pascal will perform the compound statement. Let's assume that NUM has the value of 5.
4. The value of NUM is added to SUM, and COUNT is incremented by 1. SUM will keep a running total of the values entered, while COUNT is used to tally the number of times the WHILE loop is executed.
5. A new number is requested. When the user types it, the number is stored in NUM.
6. Since the end of the compound statement has been reached, Pascal loops back to the beginning of the WHILE statement.
7. The test is again performed. If the test is TRUE, then the loop is performed again, starting at step 4. If the test is FALSE, that is, if NUM = 0, then Pascal exits the WHILE loop and proceeds to the next statement.
8. After the WHILE statement is completed, the average of the entered numbers is found simply by dividing SUM by COUNT.

■ Confirm that this program will work for varying quantities of inputs. What happens if the first number you type is zero? Since the WHILE loop has not been executed, COUNT remains equal to 0. The error message results because it simply doesn't make sense to divide by zero. Whenever division is performed by a program, care should be taken to protect against division by zero.

Below is another program that could not be written using a FOR loop. The effect of the program is to let you draw on the graphics screen by moving the mouse.

■ Create and execute the program. Whenever the mouse pointer is within the Drawing window, a line will be drawn. To end the program, press the mouse button.

```
program freehand;
 var
  x, y : integer;
begin
 while not button do
  begin
   getmouse(x, y);
   lineto(x, y)
  end
end.
```

This program introduces some new Pascal features:

- BUTTON is a Boolean function that outputs TRUE if the mouse button is pressed and FALSE otherwise.

- GETMOUSE(X,Y) is a procedure that determines the coordinates of the mouse and stores them in the variables X and Y.

A WHILE loop such as this will repeat forever if you like. With each repetition, Pascal retrieves new mouse coordinates and draws a line to that location. The loop terminates when you press the button.

Notice, incidentally, that the mouse cursor changes shape while the program is running. It is no longer the insertion pointer or the arrow pointer; instead it has a + shape. This indicates that Pascal is trying to locate the mouse cursor on the Drawing screen. Macintosh Pascal uses several different cursors; you have seen the arrow cursor, the text cursor, and now the graphics cursor.

With this program, you have no control over where your drawing starts. The curve always starts from the upper left corner of the Drawing window. You would have a little more control if Pascal waited for you to press the button before starting to draw. You could then position the cursor within the Drawing window, press the button, draw for awhile, and then release the button to stop.

This plan of action could be described like this:

1. While the button is not being pressed do nothing
2. While the button is down, draw a line from the last mouse location to the current one.

■ Here is a new version of the program that performs these two actions. Try it out.

```
program freehand2;
  var
    x, y : integer;
begin
  while not button do
  ;
    getmouse(x, y);
    moveto(x, y);
  while button do
     begin
       getmouse(x, y);
       lineto(x, y)
     end
end.
```

The second WHILE loop has not changed; it accomplishes the action in statement two. To accomplish the action in statement one the first WHILE statement has been added. It simply performs the null statement until the button is pressed. I told you the null statement was useful. Sometimes it is handy to be able to do nothing! After the first loop is exited, MOVETO is used to move the starting point for drawing to the position of the mouse.

This program allows you to draw one curve, starting and ending anywhere in the Drawing window. But you say you would like to draw several curves, each with a different beginning and end? We could do this if the two WHILE loops in the current program themselves were repeated. Then, when the button was released in the second loop, the program would just turn its interest to the first loop again and would wait for a button press.

To do this, we must introduce the *nested* loop, the loop within a loop. The statement to be executed

by a loop can be any correctly formed Pascal statement. This means that a loop can repeat another loop. A WHILE statement can repeat another WHILE statement or a FOR statement.

■ Study a version of the program that uses a nested loop shown in Fig. 5-1.

To make the structure of the program clearer, I have added a bracket to point to each loop. WHILE loops 2 and 3 are identical to the loops in the previous version. The brackets, however, make it clear that loops 2 and 3 have been placed within the compound statement of WHILE loop 1.

Statement 1 uses a trick that forces the loop to repeat itself forever. There are only two Boolean values: TRUE and FALSE. Since TRUE can never be FALSE, WHILE loop 1 will never end because of a program event. A loop that cannot terminate is often called an *infinite loop.*

To stop such a loop, you must take special action. With the mouse, pull down the Pause menu. This menu has only one option: Halt. Choose Halt and the program will stop execution. A hand will appear in the left column of the Program window showing the program line that was executing when Halt was chosen.

When you next choose Go after a Halt, one of two things will happen. If you have made no changes to the program, execution will resume where it left off. If you have done any editing to the program, or if you have chosen Reset in the Run menu, Go will cause the program to start from the beginning.

Forcing the user to end a program by choosing Halt is probably not the best strategy. A much nicer way would be to have the program halt whenever the button was clicked while the mouse was outside of the Drawing window. To program this feature, we will use the REPEAT..UNTIL statement.

```
program freehand3;
 var
  x, y : integer;
begin
 while true do
  begin
   while not button do
    ;
   getmouse(x, y);
   moveto(x, y);
   while button do
    begin
     getmouse(x, y);
     lineto(x, y)
    end
  end
end.
```

2

3

1

Fig. 5-1. A drawing program using nested WHILE loops.

THE REPEAT..UNTIL STATEMENT

REPEAT statements work similarly to WHILE statements. They perform some action until a condition is met. Let's adapt the averaging program from the last section so that it uses a REPEAT statement.

■ Try out this averaging program, which uses REPEAT..UNTIL:

```
program average_three;
  var
    count : integer;
    sum, num : real;
begin
  sum := 0;
  count := 0;
  repeat
    write('Type a number: ');
    readln(num);
    sum := sum + num;
    count := count + 1;
  until num = 0;
  writeln('The average is: ', sum / count : 10 : 4)
end.
```

When you execute this program it will appear to operate in the same manner as the earlier version that used the WHILE loop. However, you will find that the averages it produces are not correct. When you enter 5, 4, and 3, for example, the program should print 4.0000, but instead prints 3.0000. In order to understand the cause of this abnormality, we must first understand fully how the REPEAT loop works.

A REPEAT loop repeats all statements that appear between REPEAT and UNTIL. Unlike WHILE, REPEAT does not require compound statements to be enclosed by a BEGIN-END combination. This function is performed adequately by REPEAT and UNTIL, which must appear in pairs. The form of the statement is:

repeat *statement(s)* **until** *condition*

After Pascal encounters a REPEAT, it proceeds to execute the statements that follow. This continues down to the UNTIL. The condition following UNTIL is then examined. If it is FALSE, then Pascal loops back to the REPEAT. If the condition is TRUE, Pascal goes on to the statement that follows UNTIL.

It should now be possible to determine why this program produces incorrect results. Let us examine the operation of the program for several loops and determine what the values of SUM and COUNT are at the end of each loop. To do this, we will take advantage of two diagnostic tools provided by MacPascal: the Observe window and program stops.

The Observe window permits us to monitor the values of expressions during program execution.

■ Open the Observe window by choosing Observe in the Windows menu. Then arrange the windows on your screen so that you can see the Text, Program, and Observe windows.

When you choose the Observe window, the insertion pointer will appear in the right hand column of boxes, beside the legend "Enter an expression."

■ Type "num" and press Return. The word "num" will remain in the box, and the cursor will move down a line.

■ Type "sum" in the second box and "count" in the third box.

■ Finally, stretch the Observe window downward a bit so that you can see the lines for all three variables.

Next, we will install the program stops.

■ Choose Stops In from the Run menu. Notice that a new box appears at the left edge of the Program window. At the bottom of the box is a stop sign. When Pascal encounters a stop sign in this column, it will pause execution in the line opposite the sign.

■ Move the mouse pointer into this column. When it enters the box, the pointer will turn into a stop sign. Position the pointer on the same line as the UNTIL clause of the program and click. This will deposit the stop. In the future you may deposit several stop signs, but for now you need only one.

Your screen should now resemble the screen in Fig. 5-2.

Now run the program. When the program asks you to type a number, respond by typing 5. After pressing Return, examine the features of your Mac screen closely. Notice that a pointing hand has appeared on top of the stop sign beside UNTIL. This indicates that execution has halted on this line.

Your Text and Observe windows will look like the "First Input" example in Fig. 5-3. Note the values of the variables. You should have been able to anticipate them without any problem.

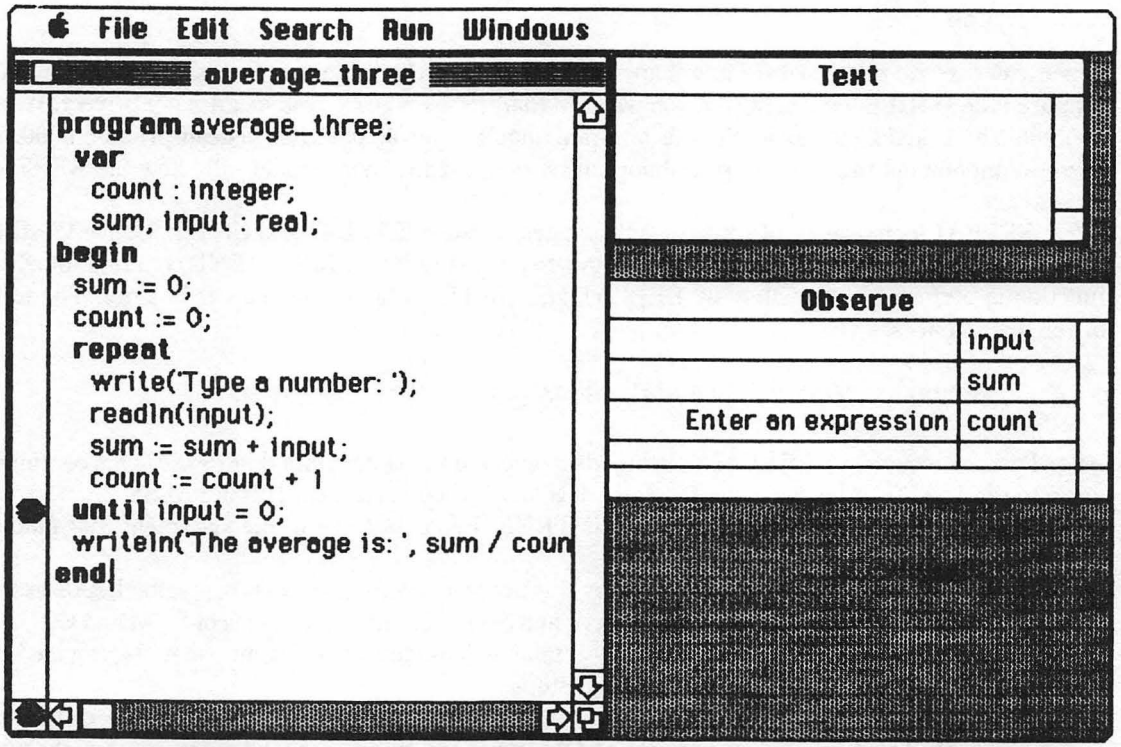
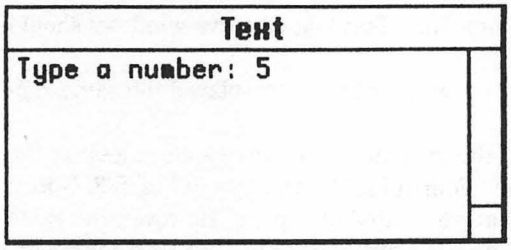
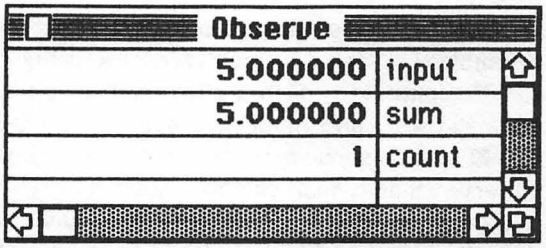


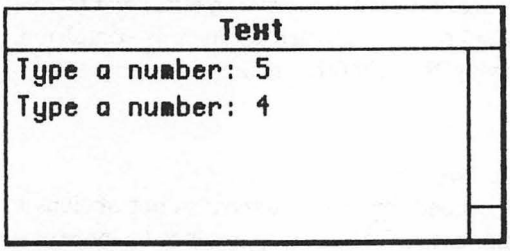
Fig. 5-2. The screen arranged for debugging the AVERAGE_THREE program.

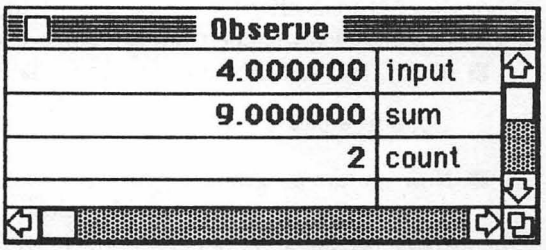
First Input: 5



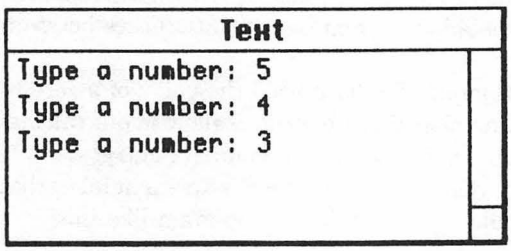


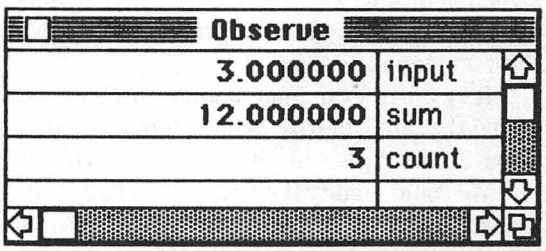
Second Input: 4



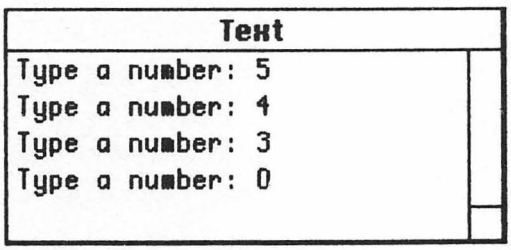


Third Input: 3





Fourth Input: 0



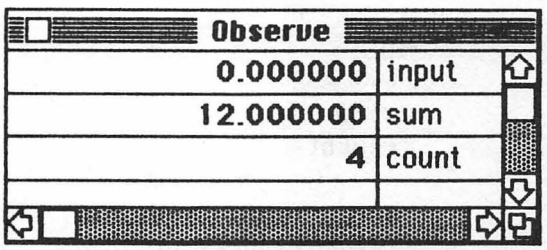


Fig. 5-3. Steps in debugging AVERAGE_THREE.

■ To make the program continue choose GO again (or type Command-G, which I find more convenient.)

■ For the second number, type a 4. Check the results. Your Text and Observe windows should resemble the "Second Input" windows in Fig. 5-3.

■ Continue by choosing Run and enter a 3. Your windows should have displayed the same contents as the "Third Input" examples in Fig. 5-3.

■ Continue one more time, and type a zero to end the program. Then take a close look at the Observe window. Your windows should now resemble the "Fourth Input" windows in Fig. 5-3. Notice that the value in SUM has not changed, which is just what we wanted to happen. However, the value of COUNT is now 4, one greater than we need to determine the average of the first three numbers. No wonder the average is incorrect!

Errors like this are so common in computer programming, that they have a name: *off-by-one-errors.* This one occurs because the program does not know enough to skip the step that increments COUNT. Although zero has no effect on the value of SUM, adding one certainly does have an effect on COUNT. Since this will always happen, we can correct for it by subtracting 1 when the loop is completed.

■ To do this, add this new line immediately following the UNTIL clause:

```
count := count - 1;
```

■ Now try the program. It will now print a correct average.

This is not a particularly desirable solution since the need for this statement is not obvious to someone who is examining the program. Generally speaking, the WHILE loop produced a more easily understood version of the program. Part of good programming style is to make programs easy to understand.

However, you will often find that a given process can be programmed using either a WHILE construction or a REPEAT..UNTIL. In the sample programs, I have been keeping the WHILE and the REPEAT..UNTIL versions similar, so that it would be possible to point out the differences between the two loop structures.

It is worth examining another problem with this program. We have used the entry of a zero to signal that all data have been entered. What if zero is a number that we want to allow in our calculations. After all, leaving it out was an arbitrary decision. How else can we end data entry?

We could modify the program so that it would look for another value. If 9999 were a number that we would not want the program to accept, then we could try to modify the program like this:

```
program average_three;
  var
    count : integer;
    sum, num : real;
begin
  sum := 0;
  count := 0;
  repeat
    write('Type a number: ');
    readln(num);
    sum := sum + num;
    count := count + 1;
```

```
    until num = 9999;
    count := count - 1;                                    {new line}
    writeln('The average is: ', sum / count : 10 : 4)
end.
```

■ Try it. Does 9999 work in the same way as zero? Resoundingly we must exclaim,"NO!"

The first version of this program took advantage of the fact that the final value of zero would not affect the value stored in SUM. Therefore, we could afford to add it to SUM. Certainly, we had to correct the value of COUNT by subtracting 1, but that simple solution appeared to work.

Using a different stop value makes the strategy blow up. We must look for a more sophisticated solution to the entire problem.

Examine the structure of the program. Notice that the new value is read into NUM before NUM is added to SUM. If we are going to keep a running total in SUM, it makes a sense to have values to total. But this arrangement causes the problem we are having in trying to end the program.

■ Let's switch the order so that the REPEAT loop looks like this:

```
begin
  sum := 0;
  count := 0;
  num := 0;
  repeat
    sum := sum + num;
    count := count + 1
    write('Type a number: ');
    readln(num);
  until num = 9999;
```

If NUM is initialized to zero, this will work. In the first pass through the loop, zero is added to SUM, which does no harm. In the last pass through the loop, the stop value is tested in the UNTIL clause, which causes the loop to terminate before the erroneous value is added to SUM. We still have the problem with COUNT being incremented one too many times. This time the extra addition happens the first time through the loop. But we know how to correct that.

However, things are not always this predictable. We do not always know what correction factor should be applied when an extra pass is made through a loop. Let's see if we can fix the program another way.

■ Here is one solution:

```
program average_three;
  var
    count : integer;
    sum, input : real;
begin
  sum := 0;
  count := 0;
  write('Type a number: ');
```

```
    readln(input);
    repeat
      sum := sum + input;
      count := count + 1;
      write('Type a number: ');
      readln(input);
    until input = 9999;
    writeln('The average is: ', sum / count : 10 : 4)
end.
```

Our solution involves reading a value both before and after SUM is calculated. This may seem inefficient, but it is a technique that is used quite frequently with REPEAT loops. And, in fact, we did something similar with the WHILE version of the program introduced earlier, although we made no special note of the fact at the time.

This new program works if we allow it to have one quirk: if the first entry is 9999, the program will not stop immediately. This is true for a very simple reason: REPEAT loops *always* execute once. The test for the REPEAT is at the end of the loop, while the test for the WHILE loop is at the beginning. In fact, this is the primary difference between the two types of loops. Since the REPEAT loop will execute at least once, we would probably conclude that the WHILE version of this program is preferable.

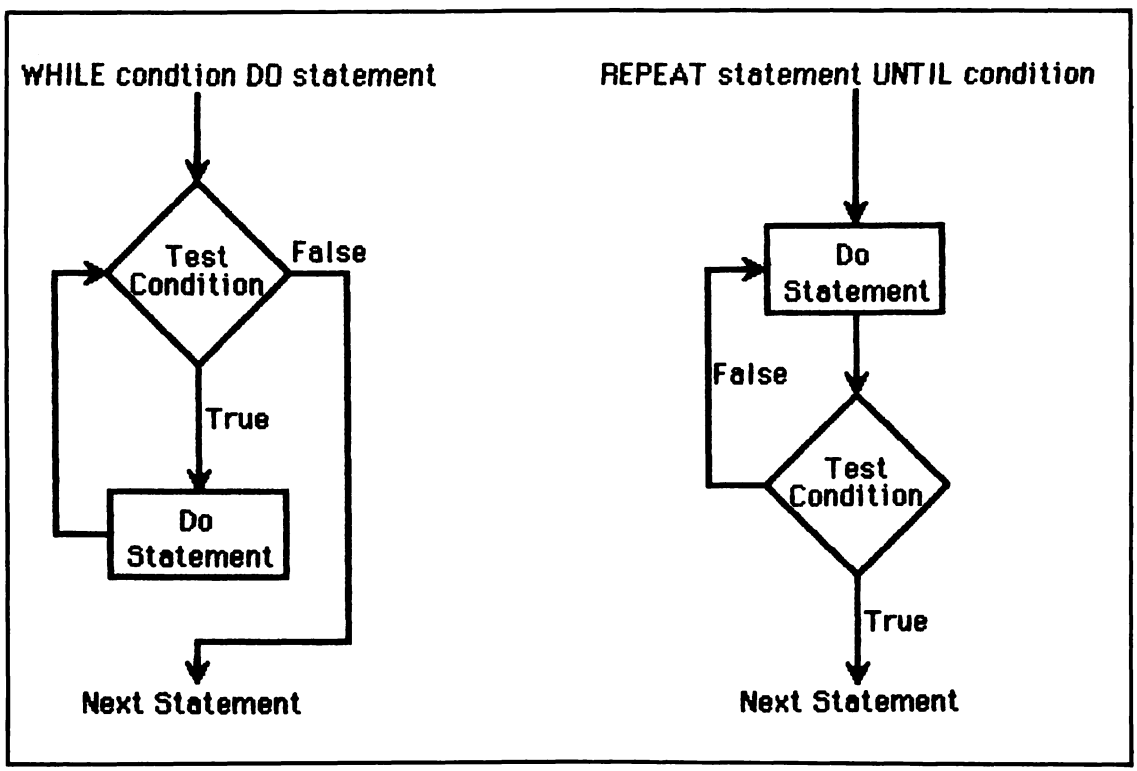


Fig. 5-4. A comparison of the flow of control in WHILE and REPEAT statements.

COMPARING WHILE AND REPEAT LOOPS

All of which brings us to the need to do a careful comparison of WHILE and REPEAT loops. Figure 5-4 shows block diagrams of the functions of these two loop types. Pay special attention to the locations of the conditional tests in the loops. In the WHILE loop, the test precedes the statements to be executed. In the REPEAT..UNTIL loop, the test follows the statements. We have already seen that this difference has important consequences.

While it is often possible to perform a task using either type of loop, it usually happens that the programs must be subtly altered. For example, here are two programs that count from 1 to 10:

```
program repeat_counting;
 var
   count : integer;
begin
 count := 1;
 repeat
   writeln(count);
   count := count + 1
 until count > 10
end.
```

```
program while_counting;
 var
   count : integer;
begin
 count := 1;
 while count <= 10 do
   begin
     writeln(count);
     count := count + 1
   end
end.
```

In this case, it is the conditional part that must be altered:

- In the WHILE loop, the condition is used to indicate when the statement should *continue.*
- In the REPEAT..UNTIL loop, the condition is used to indicate when the statement should *terminate.*

In fact, the conditional part of the statement will almost always be different when the same task is to be accomplished by different loop types.

Another significant difference is the characteristic of REPEAT loops that we noticed earlier. Generally, the WHILE statement will be used if there are conditions under which the loop should not execute even once.

By using a REPEAT loop, we can easily modify the FREEHAND program so that it will terminate if the button is clicked outside of the Drawing window.

■ Try one last program version of the FREEHAND program, which introduces this feature:

```
program freehand4;
  var
    x, y : integer;
begin
  repeat
    while not button do
    ;
    getmouse(x, y);
    moveto(x, y);
```

```
    while button do
      begin
        getmouse(x, y);
        lineto(x, y)
      end
  until (x < 0)
end.
```

This program simply replaces the outer WHILE loop with a REPEAT loop. This REPEAT loop terminates if the test is performed when the mouse is located anywhere left of the Drawing window.

■ Try it. Run the program and draw a few curves. Then move the mouse to the left of the Drawing window and click.

The click causes the first WHILE loop to terminate. The second WHILE loop does nothing because the button is not held down. Then, the UNTIL clause finds that the mouse is to the left of the Drawing window, and the program terminates.

This final version of FREEHAND illustrates that loop types can be mixed in nested loops. REPEAT, WHILE, and FOR loops may be freely combined depending on the task at hand.

Here is another graphics program, which is largely a variation on FREEHAND. However, it serves to introduce some new graphics options.

```
program Rectangles;
  var
    newx, newy, oldx, oldy : integer;
begin
  repeat
    while not button do
    ;
    getmouse(oldx, oldy);
    penmode(patxor);
    repeat
      getmouse(newx, newy);
      framerect(oldy, oldx, newy, newx);
      framerect(oldy, oldx, newy, newx);
    until not button;
    penmode(patcopy);
    framerect(oldy, oldx, newy, newx)
  until oldx < 0
end.
```

■ When you try the program, notice that it will draw rectangles from the location where you press the button to the location where you release it. This will continue until you click the button to the left of the Drawing window.

Rectangles will be drawn only when the end point is below and to the right of the starting point. This is due to the way FRAMERECT works, not to a problem with the program. We do not yet have the tools needed to solve this problem.

Before explaining the workings of the program, let's examine the new procedure PENMODE, which determines *how* the pen will draw. PENMODE does not control the pattern, which is determined by PENPAT, but the method of drawing each individual dot that makes up the image. Normally, the PENMODE is set to PATCOPY, which simply draws the requested shape in black. When PENMODE is set to PATXOR, however, drawing is accomplished by inverting each dot that is covered by the pattern. That is, black dots are changed to white, and, white dots are changed to black. (The effect is similar to INVERTRECT. In fact, when PENMODE is set to PATXOR, a rectangle drawn by FILLRECT will be drawn in the same way as the rectangles normally drawn by INVERTRECT.) An interesting thing happens when PENMODE is set to PATXOR. If we draw the same identical shape twice in the same location, the second drawing will return the Drawing window to its original state. It will be as if nothing was ever drawn!

Now, let's examine the program. Steps 1 through 6 repeat until the button is released to the left of the Drawing window:

1. The WHILE loop waits for the button to be pressed.
2. The program reads the mouse position with GETMOUSE, storing the coordinates in OLDX and OLDY.
3. The PENMODE is set to PATXOR.
4. Steps A through C are repeated. The loop continues until the button is released:

   A. The mouse position is stored in NEWX and NEWY.
   B. A rectangle is drawn with top at OLDY, left at OLDX, bottom at NEWY, and right at NEWX.
   C. The same rectangle is drawn again. Because the PENMODE is set to PATXOR, this erases the first drawing of the rectangle. This has the effect of having each rectangle flicker as it is being drawn.

5. The PENMODE is changed to patcopy.
6. The rectangle is drawn again, but this time it is drawn normally, in solid black.

YOUR PASCAL VOCABULARY

You now know these Pascal words. New words are printed in bold face type.

Reserved Words

PROGRAM	BEGIN	END	VAR
DO			

Statement Types

Assignment (:=)	Compound		
FOR..TO	FOR..DOWNTO	WHILE	REPEAT..UNTIL

Data Types

BOOLEAN	CHAR	INTEGER	REAL
STRING			

Procedures

READ	READLN		
WRITE	WRITELN		
NOTE			
FRAMERECT	PAINTRECT	FRAMEOVAL	PAINTOVAL
INVERTRECT		INVERTOVAL	
PENPAT	PENSIZE	**PENMODE**	
BUTTON	**GETMOUSE**		

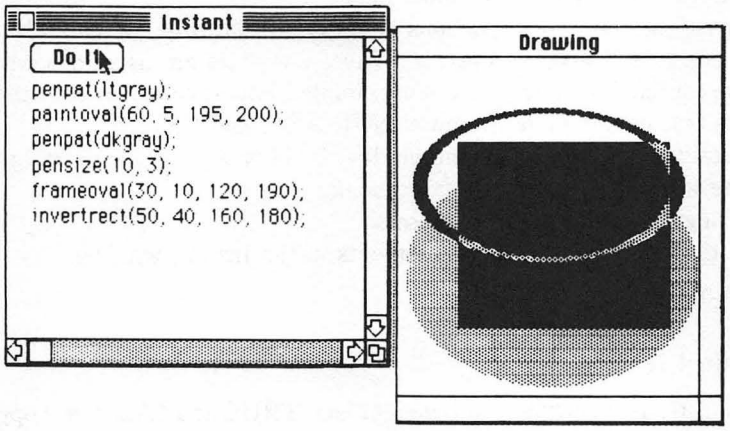
Operations

+	-	*	/
DIV	MOD		

Functions

ROUND	TRUNC

Chapter 6



Control Statements Part 3: Branching with IF..THEN and CASE

There are many situations in which a program may be called upon to make a decision. For example:

- When you are making a bank deposit to an automatic teller, the bank computer asks whether you wish to have the money placed in your checking or your savings account. Depending on your response, the computer deposits to the appropriate account.
- You are writing a program that calculates the results of a political poll. With each response, your program must increment the appropriate variables.
- You are calculating letter grades for a class. Depending on the average of each student's number grades, the computer must determine the appropriate letter grade to assign.

None of these situations can be handled simply by controlling a loop. They call the program to make one or several related decisions, often depending on complex sets of conditions. We will look at these levels of decision making in this chapter, starting from the simple and proceeding to some that are more complicated. But first we must take a closer look at the core of Pascal's decision making capability, the Boolean expression.

TOPICS COVERED IN THIS CHAPTER

- Boolean expressions
- Procedence of operators
- Working with simple and complex IF..THEN statements
- Semicolons in IF.. THEN statements
- Correcting a common bug with nested IF..THEN..ELSE statements
- The CASE statement

BOOLEAN EXPRESSIONS

Boolean expressions have only two possible values: TRUE and FALSE. We have encountered these expressions before in the conditional portions of loops. At that time, we used the Boolean *relational* operators, which are: > , < , > =,<=, and < > . These operators are used to compare data items. To this point, we have confined ourselves to making simple comparisons. However, there are three other Boolean operators that enable us to do more: NOT, AND, and OR.

■ Enter Pascal and set up the Instant window. I will be asking you to try out several examples as we sort out the workings of Boolean expressions.

NOT simply negates any Boolean expression.

■ Try executing these WRITELN statements in the Instant window:

```
writeln(not true)

writeln(not false)
```

These examples illustrate an obvious but essential fact: TRUE and FALSE are opposite values. Now, let's see what happens when we negate more complex Boolean expressions.

■ Try this example:

```
writeln(not 5 > 4)
```

The Boolean expression "5 > 4" outputs a value of TRUE to the WRITELN statement. Since the expression is Boolean, we should be able to negate it by applying NOT to it.

■ Execute this statement:

```
writeln(5 > 4)
```

We would like to negate the TRUE expression "5 > 4", and this seems to be the straightforward way to do it. However, this statement produces an error message the indicates a type mismatch.

■ Here is a version of the statement that will work. Try it out, and then we will determine what caused the failure of the first statement.

```
writeln(not(5>4))
```

This produces the correct response, which is FALSE. Obviously, the new parentheses have corrected matters.

The problem has to do with the *precedence* of operations, and we have encountered a similar situation before. When we were calculating numeric averages, we had to group the numbers within parentheses like this:

```
writeln((number1 + number2 + number3) / 3)
```

If parentheses were not used to contain the addition, the division was carried out on NUMBER3 alone, not on the sum of the three numbers. This occurs because division takes precedence over addition. The only tool we have available to override this is the use of parentheses.

It turns out that Boolean operations obey rules of precedence also. Since Boolean operations may be combined with numeric operations, let's examine these rules. There are four levels of operator priority:

HIGHEST	**NOT**	Unary operator
SECOND	*, /, **DIV, MOD, AND**	"Multiplying" operators

THIRD	+, -, OR	"Adding" operators
LOWEST	=, <>, >, <, >=, <=	Relational operators

By grouping these operators by priority, and by employing three rules, the interpretation of any expression can be made unambiguous. These are the rules Pascal applies when processing expressions:

1. When an operand appears between two operators of different precedence, the operation with the greater precedence is carried out first.
2. When an operand appears between two operators of the same precedence, the operation to the left is carried out first.
3. Parenthetical expressions are always evaluated before they are acted on by outside operands.

This gives us the information we need to determine why "NOT (5 > 4)" works while "NOT 5 > 4" produces an error message.

In the second versions, 5 is located between two operators of unequal precedence. Since NOT takes precedence over >, Pascal first attempted to evaluate the expression "NOT 5", NOT expects its parameter to be a Boolean value, however 5 is an integer. A type mismatch is the consequence.

Now the parenthesized version is easy to understand. It groups the relational expression so that it will be resolved first. > will accept two parameters of compatible data type; for example, two integers, two characters, or two strings. And, > produces a Boolean value. It is the Boolean value from > that is passed on to be negated by NOT.

NOT was described as a *unary* operator, meaning that it operates on only one operand. The other operators require two operands. We have considered all of these operators except AND and OR. These are the operators that enable us to make complex decisions. Without AND, we could not determine if a number lay between two values. OR allows us to determine if one of several conditions are met.

AND evaluates two expressions and outputs TRUE only if both expressions are true.

■ Try these two statements:

```
writeln((5 > 4) and (2 * 3 = 6))

writeln((5 > 4) and (2 * 3 = 7))
```

The first statement will print TRUE since both expressions are true. The second will print FALSE since the second expression is false.

Examine the use of parentheses in these statements closely. Notice that the inner sets of parentheses cannot be removed, since AND has a higher precedence than the two relational operators. If the parentheses are removed, Pascal attempts to use 4 and 2 as the operands of AND, which will operate only on Boolean values.

To determine if a number falls within a certain range, we must know whether the number is less than the upper limit *and* greater than the lower limit of the range. To determine if the value of variable NUMBER falls between 1 and 10, the following conditions are applied:

```
writeln((number > 1) and (number < 10))
```

AND will return the value TRUE only if both conditions are met. Notice that values of 1 or of 10 will return FALSE since we did not use >= and <=. If we wish to include 1 and 10 in the range of numbers that will be accepted, we must modify the statement like this:

```
writeln((number >= 1) and (number <= 10))
```

OR is akin to AND in that it works with two Boolean operands. In the case of OR, if either of the operands is TRUE, then OR will return TRUE. For example, the following expression will print TRUE even though one of the conditions is false.

■ Execute this statement in the Instant window:

```
writeln((5 < 3) or (6 > = 2))
```

To determine if a character stored in CH is a vowel, we might use this statement:

```
writeln((ch = 'A') or (ch = 'E') or (ch = 'I') or (ch = 'O') or (ch = 'U'))
```

This expression will be TRUE if any one of the comparisons is TRUE. As you can see, a great many expressions may appear within a logical expression. Again, the parentheses are essential if Pascal is to accept the statement.

Before leaving this discussion, let us examine a very complex Boolean expression and determine how Pascal evaluates it. We will begin with this statement:

```
writeln((6 > 2 * 2) or (5 * 2 = 13 - 3) and not (7+2 = 9))
```

■ Execute the statement in the Observe window. This will demonstrate Pascal's final evaluation of the expression. Now, we will examine the steps by which Pascal reached its conclusion.

Working with the parentheses first, the arithmetic operators take precedence over the relational ones. The multiplications, addition, and subtraction are carried out first, simplifying the statement to this:

```
writeln((6 > 4) or (10 = 10) and not (9 = 9))
```

Pascal continues to work inside the parentheses, reducing the statement to this form:

```
writeln((true) or (true) and not (true))
```

The parentheses can now be removed:

```
writeln(true or true and not true)
```

Now the operator of highest priority is NOT, and the statement becomes:

```
writeln(true or true and false)
```

AND is evaluated next:

```
writeln(true or false)
```

The final value of the original Boolean expression, therefore, is TRUE.

THE IF..THEN STATEMENT

Most decisions made in Pascal revolve around the IF..THEN statement. In its simplest form, the statement works like this:

```
if (12 = 3 * 4) then writeln('They are equal')
```

■ Try this statement in the Instant window. Substitute different values into the expression, or change the relational operator. The statement following THEN will only be carried out if the condition evaluates as TRUE. We can easily put this to work in a program.

■ Enter this program and try it out.

```
program food;
  var
    answer : string;
begin
  writeln('Do you like pickle jello?');
  readln(answer);
  if answer = 'no' then
    writeln('I''m with you!')
end.
```

Suppose, however, that the person being questioned has a rare affinity for pickle jello? Can we provide an appropriate response in that case? One possible approach would be to use < > to test for any answer that is not 'no'.

■ Edit the program to include a second IF..THEN statement.

```
if answer = 'no' then
  writeln(' I''m with you.');          {add semicolon}
if answer <> 'no' then                 {new statement}
  writeln('How weird can you get?')
```

■ Try the program to determine that it responds properly for each response.

Having one of two actions to decide between is an extremely common situation. We can solve it by including two IF..THEN statements, but this gets to be a bit wordy since it forces us to perform each test twice. To make things simpler, and to emphasize the either or situation that such decisions present, Pascal allows for an optional ELSE clause in the IF statement. The function of the IF..THEN..ELSE statement can be made pretty clear simply by writing it into our FOOD program.

■ Edit the program, modifying the second IF..THEN statement like this:

```
program food;
  var
    answer : string;
begin
  writeln('Do you like pickle jello?');
  readln(answer);
  if answer = 'no' then
    writeln('I''m with you. ')                {remove semicolon}
  else                                        {change from if..then}
    writeln('How weird can you get?')
end.
```

■ Try this version. It will operate just like the version that contained two IF..THEN statements. However, you probably find it easier to read. Further, the ELSE makes it instantly obvious that only one of the two available actions will be carried out.

Here we must again pay attention to the proper use of semicolons in Pascal. Notice that there are no semicolons in the IF..THEN..ELSE statement of this program. This is a single statement. If you were to place a semicolon ahead of the ELSE, Pascal would get upset. So, either of the following structures may be used in an IF statement:

if *condition* **then** *statement*

if *condition* **then** *statement* **else** *statement*

Figure 6-1 presents block diagrams of the two varieties of IF statements. Looking at these diagrams, it is easy to understand why IF statements are often called *branching* statements.

Conditional statements can also make decisions of a more complicated nature. Suppose that you were writing a guidebook program. You would probably require a way to convert numeric scores to letter grades. Let's create such a program, which will assign letter grades based on the following criteria:

Numeric Score	Letter Grade
90 or greater	A
80 or greater but less than 90	B
70 or greater but less than 80	C
60 or greater but less than 70	D
less than 60	F

The first step will be the creation of a simple program that assigns A grades.

■ Enter this short program. Then run and test it with scores of 85 and 95.

```
program grades;
 var
  score : integer;
begin
 write('Type a number score: ');
 readln(score);
 if score >= 90 then
  writeln('A')
end.
```

This program shouldn't surprise you at all. Grades of 90 and above will cause the program to write an 'A'. Lower grades will be ignored. Next, we will add a statement to process B grades.

■ Add the indicated semicolon and the indicated lines before the end of the program:

```
 if score >= 90 then
  writeln('A');                    {add semicolon}
 if score >= 80 then               {new statement}
  writeln('B')
end.
```

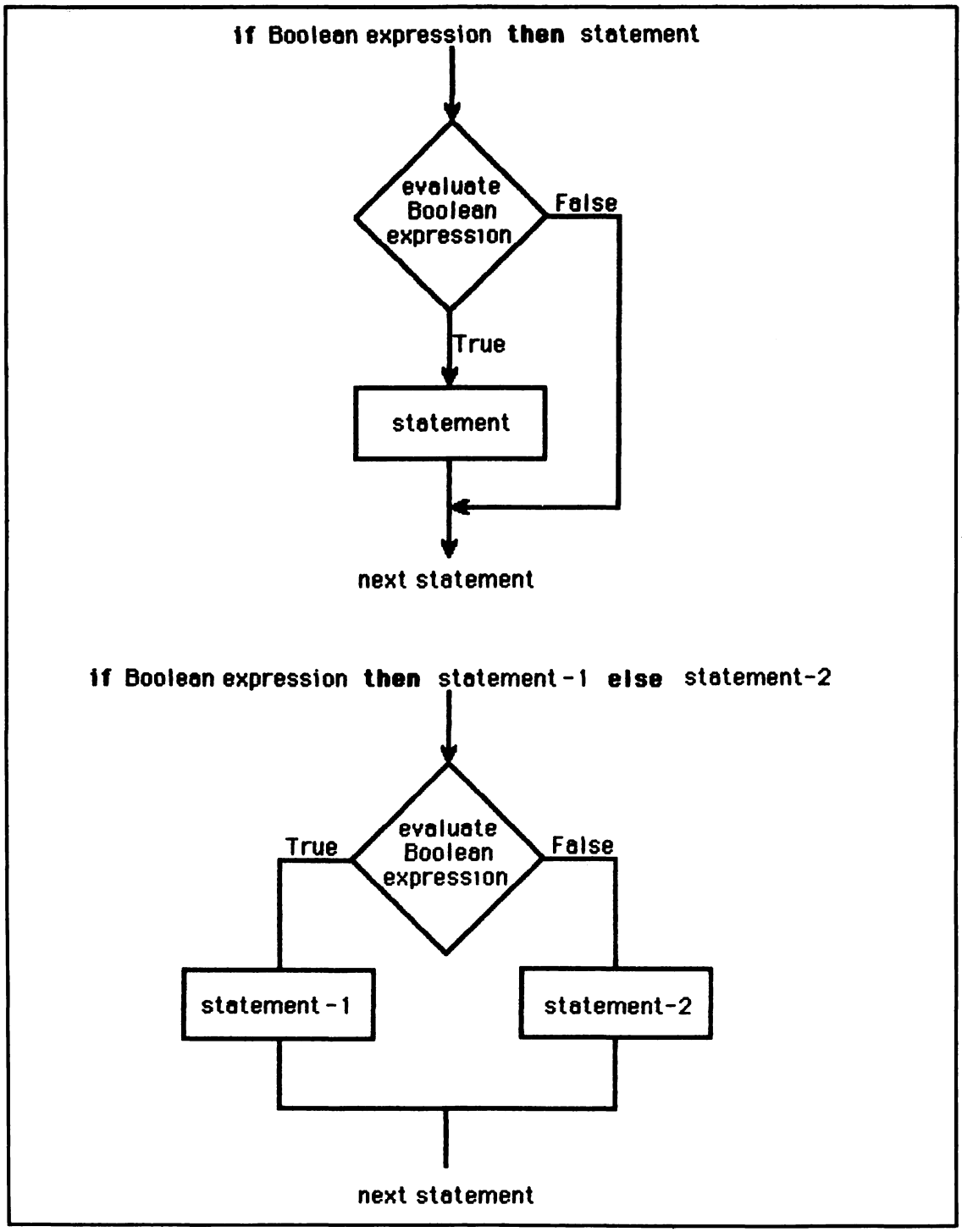


Fig. 6-1. The flow of control in IF..THEN and IF..THEN..ELSE statements.

■ Now try the program with inputs of 95 and 85. What are the results? Why did a score of 95 produce both an A and a B grade?

Such situations cause difficulties for some beginning programmers. They reason that, since the first IF statement tested TRUE and executed its dependent statement, the rest of the program will be ignored. However, there is a principle at work here, one so obvious that it is almost never stated. I call it the *principle of sequential execution.* In most programming situations, when a statement has completed execution the statement immediately following it will be executed. We have taken this completely for granted up to this point, but many beginners do not expect it to be working in programs such as the present one.

There is nothing in this program that has revoked the principle of sequential execution. After the first IF statement has executed, there is nothing to prevent the second IF from also executing. Since SCORE has a value of 95, the first statement will naturally print an "A". But then, as we have just observed, the next statement in the program must be executed. Since SCORE is also greater than 85, a "B" is also printed.

What we wish to set up is an either-or situation, and I hope you are now shouting, "This looks like a job for IF..THEN..ELSE." Indeed it is.

■ Here are the necessary modifications. Type them in and test the program with scores of 85 and 95:

```
if score >= 90 then
 writeln('A')                          {remove semicolon}
else if score >= 80 then               {add the else}
 writeln('B')
```

You will find out that this program processes numeric scores of 80 and above appropriately. It will take much now to complete the gradebook program. We can easily extend the ELSE statements.

■ Complete the program by adding the indicated lines:

```
program grades1;
  var
    score : integer;
begin
  write('Type a number score: ');
  readln(score);
  if score >= 90 then
    writeln('A')
  else if score >= 80 then
    writeln('B')
  else if score >= 70 then             {new lines begin}
    writeln('C')
  else if score >= 60 then
    writeln('D')
  else writeln ('F')                   {new lines end}
end.
```

The program should now function for all scores. Notice that it was not necessary to use any test in

the line that printed 'F' since by then all other possibilities had been eliminated.

A note on a recurring theme: since all of those lines are parts of a single IF statement, there are no semicolons anywhere in them. Any semicolon would mark the end of the statement and would cause either abnormal operation or an error.

There are many times when you will want an IF statement to initiate more than one action for a given branch. For example, let's improve the GRADE program so that it keeps a count of the times each letter grade occurs. After all of the scores are entered, the program will print a report detailing the frequency with which each grade occurred. To do this, the THEN statements must initiate two actions. The program must continue to print out the individual letter grades, just as it does now. Also, when a letter grade is determined, a counter for that grade must be incremented. This is easily done by using a compound statement. For an A grade, the statement to accomplish these tasks looks like this:

```
if score >= 90 then
  begin
    writeln('A');
    asum := asum + 1
  end
```

We used the compound statement to group statements within loops in the last two chapters. Again we see that a compound statement may appear anywhere a simple statement is acceptable. Both the WRITELN statement and the assignment statement will be performed whenever SCORE > = 90. You can see that a semicolon is acceptable inside of a compound statement even though the compound statement is within an IF statement. In fact, this situation requires the semicolon. However, as far as the IF statement is concerned the compound statement is a single statement, and it may not be followed by a semicolon.

Since we will be entering an indefinite number of scores, the situation obviously calls for a loop to handle the repetition. A good choice would appear to be a REPEAT loop that will continue to accept scores until an unacceptable score is entered. We used such a loop in the last chapter.

Finally, a great many more variables will be required: one to keep track of the frequency for each letter grade. The result is, by far, the longest program you have yet seen. If you examine it carefully, however, you will see that it is built up of the same essential structures as other programs you have encountered.

■ Enter the complete program and experiment by entering a series of number grades between 0 and 100. To stop entering grades and bring up the summary report, respond with '999' or some other out of range value when you are asked to type a grade.

```
program grades2;
  var
    score, asum, bsum, csum, dsum, fsum : integer;
begin
  showtext;
  asum := 0;
  bsum := 0;
  csum := 0;
  dsum := 0;
  fsum := 0;
```

```
repeat
  write('Type a numeric score: ');
  readln(score);
  if (score > 100) or (score < 0) then
    writeln('Ending score entry. Here are the totals:')
  else if (score >= 90) then
    begin
      writeln('A');
      asum := asum + 1
    end
  else if score >= 80 then
    begin
      writeln('B');
      bsum := bsum + 1
    end
  else if score >= 70 then
    begin
      writeln('C');
      csum := csum + 1
    end
  else if score >= 60 then
    begin
      writeln('D');
      dsum := dsum + 1
    end
  else
    begin
      writeln('F');
      fsum := fsum + 1
    end;
until(score < 0) or (score > 100);
writeln(asum, ' Grades of A');
writeln(bsum, ' Grades of B');
writeln(csum, ' Grades of C');
writeln(dsum, ' Grades of D');
writeln(fsum, ' Grades of F');
end.
```

When you are satisfied that the program works as it should, identify the various structures in the program. Be sure that you can identify:

- The REPEAT loop. There are a lot of lines between its beginning and end.
- The method of terminating the loop.
- The new IF..THEN..ELSE that eliminates scores greater than 100 from consideration.

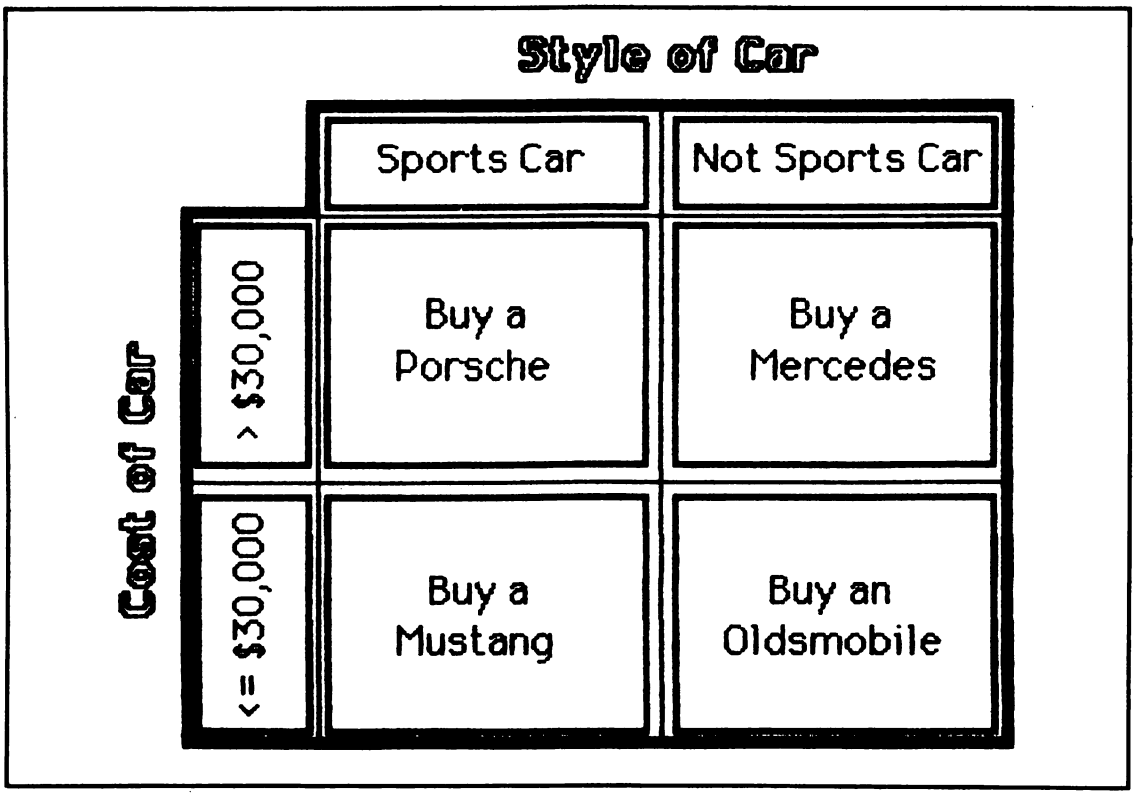


Fig. 6-2. The decision table for car selection.

- The structure of the nested IF..THEN..ELSE statements. It has not been changed much from the earlier grade program. It has simply been expanded.
- The mechanism for incrementing each of the grade counters.

If you have successfully identified these structures, you will realize that, although the program is quite large, it is not very complicated.

This is not a particularly elegant program, and you are probably wondering if there isn't a simpler way to accomplish the same thing. In fact there are several ways to simplify things. One technique involves the CASE statement, which will be introduced later in the chapter.

By combining IF..THEN statements differently, still more complex determinations can be made. Let's investigate a situation in which a decision is made based on the answers to two questions. The program will recommend a car purchase, based on two conditions: cost (over or under $30,000) and style (sports car or not). The logic for this decision is presented in the table in Fig. 6-2.

■ Let's build part of the program first. Type in this program, which considers the cost variable.

```
program car1;
  var
    response : char;
    highcost, sportscar : boolean;
```

```
begin
  showtext;
  write('Do you want to spend over $30,000? (answer y or n)');
  read(response);
  if response = 'y' then
    highcost := true
  else
    highcost := false;
  if highcost then
    writeln('Buy a Porsche.')
end.
```

The question portion of the program accepts a yes-no response. Actually, since READ is used with a Char variable, only one character is accepted, and the responses are simply "y" or "n". The IF..THEN..ELSE statement translates the character into a Boolean value, which is stored in the variable HIGHCOST.

■ This method of converting answers to Boolean values has several advantages. It requires only a simple response from the user of the program. Also, since the name of the Boolean variable was carefully chosen, the conditional statement is very easy to read. Compare these versions of the IF statement:

```
if response = 'y' then writeln('Buy a Porsche.')

if highcost then writeln('Buy a Porsche.')
```

The first statement only makes sense if we look to the earlier portion of the program to determine what is meant by a response of 'y'. The second version is much easier to interpret. Since questions may often be separated from the conditional statements that act on the responses, the approach used in this program is very useful. In the next chapter, we will see a way to further simplify the construction of programs that use this method of managing input data.

A difficulty with the program as it stands is that any entry other than a 'y' will be interpreted as a 'no' response. This includes an upper case 'Y', which will not test as being equal to a lower case 'y'. With interactive programs, we must be very careful to allow for erroneous entries by users. We will learn some methods of input error checking later on. We will also add an OR statement to future versions of the program that will take care of the case problem.

The sports car question requires a second question sequence. Notice the OR statement, which is added to solve the problem just discussed.

■ Add these new lines to your program immediately after the line "highcost := false;".

```
writeln('Do you want a sports car? (answer y or n) ');
readln(response);
if (response = 'y') or (response = 'Y') then
  sportscar := true
else
  sportscar := false;
```

These lines also store a Boolean value into the variable SPORTSCAR. This done, we can proceed to write the rest of the program. We will do this in two different ways to illustrate two different techniques of handling multiple branching. The first approach simply uses four IF..THEN statements, one for each of the four possible decision outcomes.

```
program car1;
  var
    response : char;
    highcost, sportscar : boolean;
begin
  showtext;
  write('Do you want to spend over $30,000? (answer y or n)');
  readln(response);
  if (response = 'y') or (response = 'Y') then
    highcost := true
  else
    highcost := false;
  writeln('Do you want a sports car? (answer y or n) ');
  readln(response);
  if (response = 'y') or (response = 'Y') then
    sportscar := true
  else
    sportscar := false;
  if highcost and sportscar then              {modify this line}
    writeln('Buy a Porsche.');
  if highcost and not sportscar then          {first new line}
    writeln('Buy a Mercedes.');
  if not highcost and sportscar then
    writeln('Buy a Mustang.');
  if not highcost and not sportscar then
    writeln('Buy an Oldsmobile.')             {last new line}
end.
```

■ Add the new lines and try the program out. Refer to Fig. 6-1, and experiment with all four possible combinations of answers to the questions. In each case, an appropriate response should be produced.

In producing a program like this, it is necessary to make sure that each of the IF statements takes effect only in the proper circumstances. For the purposes of this program each statement applied the AND operator to two Boolean expressions. NOT was used as required to produce the desired response. For any given combination of answers, only one of the "Buy" sentences will be printed.

There is another way of achieving the same results. It is somewhat more difficult to understand, and it presents some peculiar problems, but this second method is more readily adapted to more complex decision making. This method involves nested IF..THEN..ELSE statements.

■ Here is the second version of the program. Replace the last four IF..THEN statements with the following:

```
    if highcost then
      if sportscar then
        writeln('Buy a Porsche.')
      else
        writeln('Buy a Mercedes.')
    else if sportscar then
      writeln('Buy a Mustang.')
    else
      writeln('Buy an Oldsmobile.')
end.
```

■ Since we have created a new version of the program, change the identifier in the program header:

```
program car2;
```

■ Run this version several times, again entering each of the four possible pairs of answers. Are the answers appropriate in each case? They should agree with the logic table presented earlier.

As a first step in explaining the conditional statements, I am going to change the indentation scheme and label the statements. Indentation does not change the way a program works, but proper indentation can help clarify the structure of a program. There is no standard for indentation in Pascal, so the developers of Macintosh Pascal were free to devise one of their own. MacPascal performs this indentation automatically, which is usually very convenient. In this case, however, the automatic indentation system does not clearly display the structure of the statements. A clearer organization is shown in Fig. 6-3.

This situation is not unlike the nested loops that were encountered in the last few chapters. It is a single statement, as you should gather from the absence of semicolons. The primary IF..THEN..ELSE statement is statement 1 in the figure. Examination of this statement reveals that the statements performed by the THEN part and the ELSE parts of statement 1 are themselves

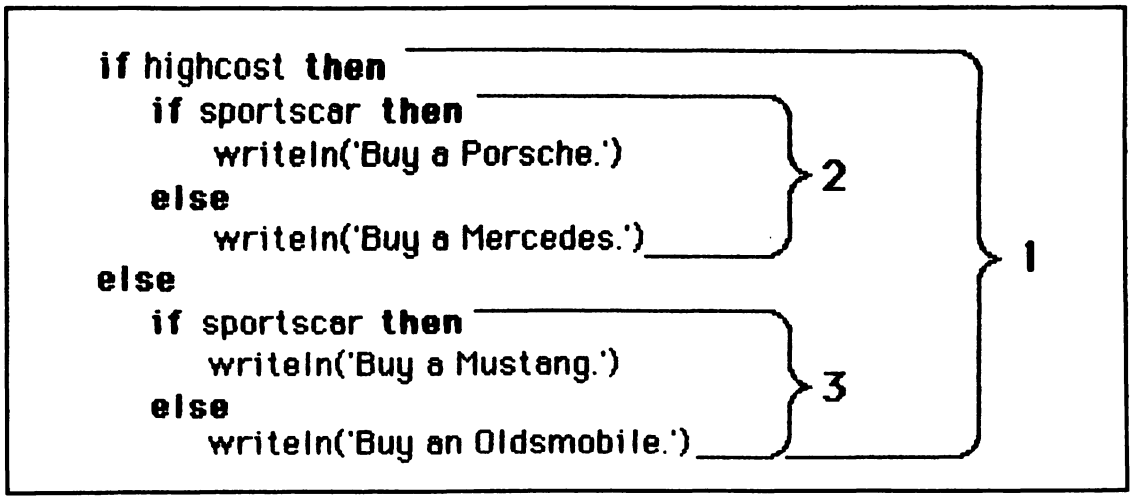


Fig. 6-3. The nesting of the IF statements in the CAR2 program.

IF..THEN..ELSE statements. Because statement 1 encloses statements 2 and 3, we can refer to statement 1 as the *outer* statement, and to statements 2 and 3 as *inner* statements.

If HIGHCOST is TRUE, then statement 2 is executed. If HIGHCOST is FALSE, statement 3 is executed. Between the two inner statements, four actions are possible. Each action may occur only in one set of circumstances. For example, "Buy a Mustang." will be printed only if HIGHCOST is FALSE and SPORTSCAR is TRUE. Before you continue, be sure you understand how and when each of the four possible responses is produced. The modified indentation scheme should help you considerably.

A NESTED IF STATEMENT BUG

Nested IF statements have a few tricks in store.

■ To illustrate, remove the following lines from the program:

```
else
  writeln('Buy a Mercedes.')
```

■ First, examine the program and attempt to anticipate how it will respond to each of the possible answer combinations. When you are satisfied with your predictions, execute the program, trying each of the possible answers. You will probably find that your predictions were wrong on one or more counts. We have some explaining to do.

Only when HIGHCOST and SPORTSCAR are both TRUE does the program respond as before. When HIGHCOST is FALSE, nothing was printed at all. And, perhaps most perplexing, when HIGHCOST is TRUE and SPORTSCAR is FALSE the program advised you to buy an Oldsmobile. This option used to be selected only when HIGHCOST was FALSE.

These problems are all caused by an ambiguity in the interpretation of IF..THEN..ELSE statements. To explain the situation, let's reformat the statements a couple of different ways. Chances are that you thought of them about as they are shown in Fig. 6-4.

With this interpretation, we have simply removed the ELSE portion of statement 2. The ELSE that controls statement 3 is still seen to be the ELSE clause of statement 1, and nothing in the operation of statement 3 should be affected. If Pascal interpreted the statements like this, when HIGHCOST

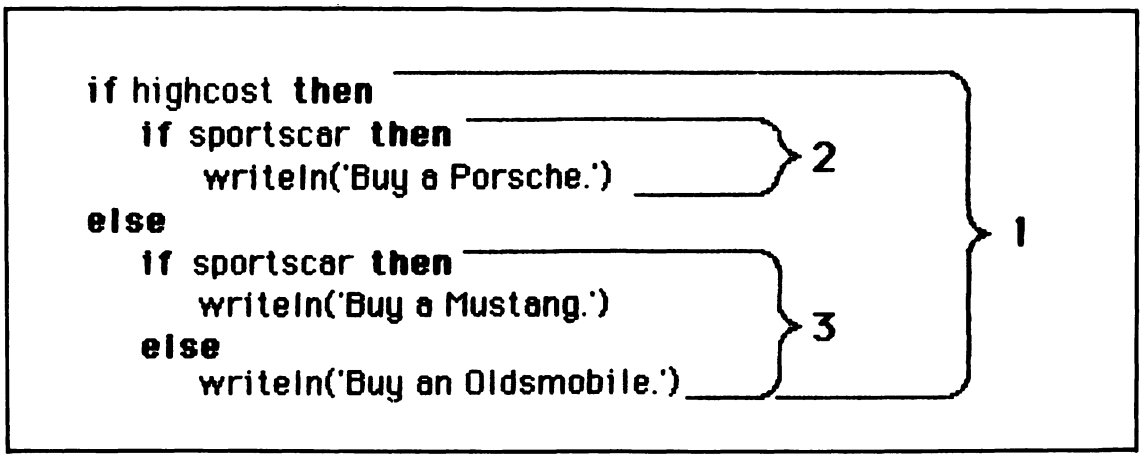


Fig. 6-4. One way of interpreting the nested IF statements.

is FALSE the ELSE portion of the statement would execute and the Mustang and Oldsmobile choices would print just as they did in the earlier version of the program. Also, if we asked for a high cost car that was not a sports car, nothing at all would be printed, since there is no ELSE portion in the second IF statement.

But here is another interpretation of the statement. In this case, the ELSE that controls statement 3 has become the ELSE clause for statement 2. This interpretation is diagrammed in Fig. 6-5.

We must determine which IF statement the first ELSE is associated with. If it is associated with statement 1, then the diagram in Fig. 6-4 is appropriate. If it is associated with statement 2, then the diagram in Fig. 6-5 is correct.

Examine Fig. 6-4 closely. How would the statements depicted here interpret a HIGHCOST value of TRUE and a SPORTSCAR value of FALSE?

1. Since HIGHCOST is TRUE, the THEN branch of statement 1 would be taken, which leads to the execution of statement 2.
2. In statement 2, SPORTSCAR is FALSE, so the THEN branch cannot be taken. Pascal would look for an ELSE branch.
3. Since there is no ELSE branch, nothing would be printed.

But this is not what is happening. Instead, the program is printing "Buy an Oldsmobile."

Also, for the structure presented in Fig. 6-4, when HIGHCOST is FALSE, statement 3 should operate as it did earlier, producing the Mustang or the Oldsmobile responses as appropriate.

Now, examine Fig. 6-5. Does it reflect the results our program is producing? If the program is working as Fig. 6-5 would suggest, the following events should take place when HIGHCOST is TRUE and SPORTSCAR is FALSE:

1. Since HIGHCOST is TRUE, statement 1 takes the THEN branch. This causes statement 2 to be executed.
2. Since SPORTSCAR is FALSE, the ELSE branch of statement 2 is taken. This results in the execution of statement 3.
3. Statement 3 determines that SPORTSCAR is FALSE. Therefore, the ELSE branch is executed and the program prints, "Buy an Oldsmobile."

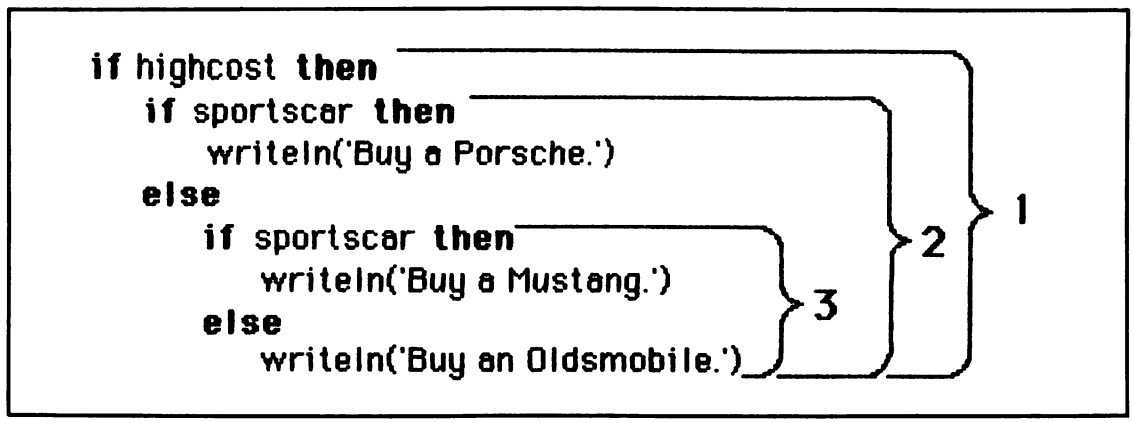


Fig. 6-5. Another interpretation of the nested IF statements.

This, in fact, is what the program is doing. Close examination of Fig. 6-5 also reveals that nothing would be printed if HIGHCOST were FALSE, since there is no ELSE branch for statement 1. So Fig. 6-5 seems to be the appropriate interpretation of the statement.

It would appear that statement 2 has priority where this ELSE is concerned. And, in fact, this is the case. Pascal expects every IF statement to have both a THEN and an ELSE branch unless something happens to end the statement. Further, the most recent IF has priority in this regard. Specifically, in a situation involving nested IF statements, the innermost statement has first claim on an ELSE. In Fig. 6-5, we observe that IF statement 3 is nested inside statement 2.

Suppose that we wish to force Pascal to adhere to the interpretation shown in Fig. 6-4? How can we associate the ELSE of statement 3 with the outermost IF? Somehow, we must inform Pascal that statement 2 is completed and does not require the ELSE clause. Since semicolons are often used to separate things, lets try using one after statement 2.

■ Edit the statement like this:

```
if highcost then
  if sportscar then
    writeln('Buy a Porsche.');                {add semicolon}
```

Unfortunately, when you do this, Pascal displays the ELSE in outline text, indicating a syntax error:

```
if highcost then
  if sportscar then
    writeln('Buy a Porsche.');
  else if sportscar then
    writeln('Buy a Mustang.')
  else
    writeln('Buy an Oldsmobile.')
```

This doesn't work. In fact, we can state flatly that: *Pascal will never allow a semicolon to appear immediately before a THEN or an ELSE. And a semicolon will only be allowed after THEN or ELSE if it marks the end of a completed IF statement.* Since we cannot place a semicolon before the ELSE, some other means of marking the end of the statement must be found.

Semicolons are not the only tools we have for marking the ends of statements. We can also use an END. Since an END must be matched with a BEGIN, let's try enclosing all of statement 2 like this:

```
if highcost then
  begin                                   {insert begin}
    if sportscar then
      writeln('Buy a Porsche.');
  end                                     {insert end}
else if sportscar then
  writeln('Buy a Mustang.')
else
  writeln('Buy an Oldsmobile.')
```

■ Enter and test these modifications. Does the program work as it should? All responses should work, except that when HIGHCOST is TRUE and SPORTSCAR is FALSE nothing will be printed.

Now, let's conclude this discussion and try to state some general principles.

An IF statement remains active until one of three things happens:

1. an ELSE branch completes it,
2. an END is used appropriately, or
3. a semicolon separates it from a subsequent statement

Since semicolons cannot appear before THEN or ELSE, only methods 1 and 2 can work within a nested IF statement. So, the general rule for untangling nested IF..THEN..ELSE statements in this:

An ELSE branch will always belong to the nearest preceding IF that is still active.

Therefore, whenever an IF..THEN statement is included in the THEN branch of an IF..THEN..ELSE statement, you should enclose the complete inner IF..THEN statement with a BEGIN-END pair. This will prevent possible confusion.

THE CASE STATEMENT

Pascal provides an alternate method of controlling multiple branching. IF..THEN expressions rely on the evaluation of Boolean expressions to determine their branching behavior. CASE statements may be used in situations where the value of other expression types can be used to determine branching behavior. For example, here is a simple program that uses values of a CHAR type variable to control branching:

```
program calculate;
  var
    n1, n2 : integer;
    operator : char;
begin
  write('First number: ');
  readln(n1);
  write('Operation(+, -, *, or /): ');
  readln(operator);
  write('Second number: ');
  readln(n2);
  case operator of
    '+' :
      writeln('Sum is', n1 + n2);
    '-' :
      writeln('Difference is', n1 - n2);
    '*' :
      writeln('Product is', n1 * n2);
    '/' :
      writeln('Quotient is', n1 / n2)
  end
end.
```

The WRITELN and READLN statements at the beginning of the program serve to store two integer values and a character value into appropriate variables. The value of the character variable OPERATOR is then used by the CASE statement to select an appropriate WRITELN statement for execution.

■ Enter the program and enter the following responses:

```
5+3          14/2          3*6
```

Confirm that the program works as described.

■ Then enter

```
5=3
```

There is no entry for '=' in the CASE statement. How does the program react?

The CASE statement takes this general form:

case *expression* **of** *case list* **end**

In our program the expression is the variable named by OPERATOR. The case list contains the various branches that may be taken. Each item in the case consists of a control value and a statement to be performed. Here is one of the entries from the case list:

```
'-':
  writeln('Difference is', n1 - n2);
```

The '-' is a possible value of OPERATOR. When the expression following CASE has this value, the action following the colon will be executed. If necessary, many cases may appear in the case list. Each one ends with a semicolon.

The control value in each case is a constant. Expressions cannot appear here. This is a restriction on the utility of CASE statements.

Another restriction is that the expression and the case constants must be of an ordinal type. So far, we have worked with three ordinal types: Integer, Char, and Boolean. If we would like to select cases based on a nonordinal type (such as Real or String) we must somehow translate the data to an ordinal form.

An interesting feature is the use of END to terminate the CASE statement. END turns out to be a multipurpose terminator in Pascal. So, you will not necessarily find a BEGIN for each END. This can lead to some confusion, so just as with semicolons, you must be sure to learn how END is used in each situation. Because END can be used in several situations, programmers frequently include a comment to indicate that an END is associated with a CASE statement.

What happened when you typed something other than +, –, *, or / where an operator was expected? You should have received the bug message, "Case expression didn't match case constant." If you entered '=' as an operator, Pascal would not be able to find a case with '=' as its case constant. Whenever you use a CASE statement, you must do one of two things. You may check to be sure that an unexpected value is never given to the statement (the only option in strictly standard Pascal), or you may establish a case for exceptional values. This can be done through inclusion of an OTHERWISE clause.

■ Add this OTHERWISE clause to your program.

```
case operator of
  '+' :
    writeln('Sum is', n1 + n2);
  '-' :
    writeln('Difference is', n1 - n2);
  '*' :
    writeln('Product is', n1 * n2);
  '/' :
    writeln('Quotient is', n1 / n2);          {add semicolon}
  otherwise                                   {new line}
    writeln('I don't recognize ', operator)   {new line}
end  {case statement}
```

■ Now try the program. Confirm that incorrect data produces the error message in the program instead of Pascal's bug message.

Let's go back to the gradebook program we wrote earlier. It is evident that IF statements can be a very long winded way to control a situation involving multiple branches. Can we use the CASE statement to simplify things?

The gradebook program must consider a great many possible numeric values. The IF statement version could use Boolean expressions to determine if a score fell within a given range. However, Boolean expressions can have only two values, while we have five possible letter grades. Further, we have 101 possible numeric scores. Must we establish 101 cases, one for each score?

We can simplify things greatly by using DIV, along with a new feature of the CASE statement. First, we will write the program so that it reduces the full range of possible scores down to just ten possibilities. The trick is to use DIV to perform an integer division with a divisor of 10. Here is a table that shows the relationships between the possible numeric scores, the results when they are processed by DIV, and the associated letter grades:

Range of SCORE					SCORE DIV 10	Letter Grade
90	≤	SCORE	≤	100	9,10	A
80	≤	SCORE	<	90	8	B
70	≤	SCORE	<	80	7	C
60	≤	SCORE	<	70	6	D
0	≤	SCORE	<	60	0,1,2,3,4,5	F

DIV thus reduces all values of SCORE down to just ten possibilities. These ten values can fit very neatly into a CASE statement. We have not seen it yet, but CASE allows more than one case constant to appear in a case. This allows us to greatly simplify the gradebook program. Here is a gradebook program based on CASE.

```
program grades3;
  var
    score : integer;
begin
```

```
    write('Type a number score: ');
    readln(score);
    score := score div 10;
    case score of
      10, 9 :
        writeln('The letter grade is: A');
      8 :
        writeln('The letter grade is: B');
      7 :
        writeln('The letter grade is: C');
      6 :
        writeln('The letter grade is: D');
      5, 4, 3, 2, 1, 0 :
        writeln('The letter grade is: F');
      otherwise
        writeln('That is not a score between 0 and 100')
    end  {case statement}
end.
```

■ Enter this program and confirm that it produces the same results as the first gradebook we developed.

As mentioned, this program associates more than one value with certain cases. When multiple values are required, they are separated by commas.

I think you will agree that this program is simpler than the earlier version that used IF statements. There is also no difficulty with making sure that nested IF statements are properly set up and punctuated.

As one last refinement, let's add the feature we included earlier that permits the program to count the frequency with which each letter grade occurs. Compare this program to GRADES2, which performed the same actions using IF statements.

```
program grades4;
  var
    acount, bcount, ccount, dcount, fcount, score: integer;
  begin
  acount := 0;
  bcount := 0;
  ccount := 0;
  dcount := 0;
  fcount := 0;
  repeat
    write('Type a number score: ');
    readln(score);
    score := score div 10;
    case score of
```

```
        10, 9 :
          begin
            writeln('The letter grade is A');
            acount := acount + 1
          end;
        8 :
          begin
            writeln('The letter grade is B');
            bcount := bcount + 1
          end;
        7 :
          begin
            writeln('The letter grade is C');
            ccount := ccount + 1
          end;
        6 :
          begin
            writeln('The letter grade is D');
            dcount := dcount + 1
          end;
        5, 4, 3, 2, 1, 0 :
          begin
            writeln('The letter grade is F');
            fcount := fcount + 1
          end;
        otherwise
          writeln;
      end  {case statement}
      until (score < 0) or (score > 10);
      writeln(acount, ' Grades of A');
      writeln(bcount, ' Grades of B');
      writeln(ccount, ' Grades of C');
      writeln(dcount, ' Grades of D');
      writeln(fcount, ' Grades of F')
    end.
```

YOUR PASCAL VOCABULARY

You now know the following Pascal words. Words introduced in this chapter are printed in bold face type.

Reserved Words

PROGRAM BEGIN END VAR

Statement Types

Assignment (:=)	Compound		
FOR..TO	FOR..DOWNTO	WHILE	REPEAT..UNTIL
IF..THEN	**IF..THEN..ELSE**	**CASE**	

Data Types

BOOLEAN	CHAR	INTEGER	REAL
STRING			

Procedures

READ	READLN		
WRITE	WRITELN		
NOTE			
FRAMERECT	PAINTRECT	FRAMEOVAL	PAINTOVAL
INVERTRECT		INVERTOVAL	
PENPAT	PENSIZE	PENMODE	
BUTTON	GETMOUSE		

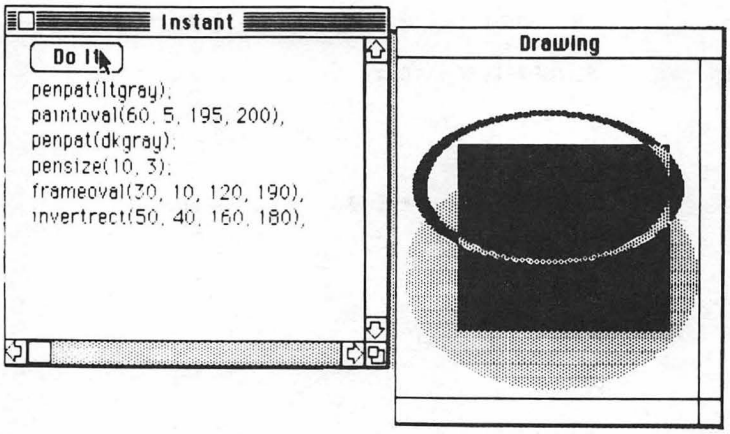
Operations

+	-	*	/
DIV	MOD		
>	>=	<	<=
<>	=		
NOT	**AND**	**OR**	

Functions

ROUND	TRUNC

Chapter 7



Defining Procedures and Functions

Macintosh Pascal possesses a huge repertoire of procedures that we can call on. Ultimately, everything we do in Pascal involves arranging those procedures and controlling them appropriately, so that the desired task will be performed. Quite often, we find that we have developed a program structure that is of sufficient interest that we would like to use it several places in a program, or, we would like to have a convenient way to move the structure from one program to another.

In fact, it would be nice to be able to add our own commands to Pascal's vocabulary, for example, a command to sort a series of words into alphabetical order, a command to find the volume of a cylinder, or a command to draw a graphic shape that MacPascal does not already know how to draw. To do this, we must learn how to create our own Pascal procedures.

TOPICS COVERED IN THIS CHAPTER

- Using the SIN and COS functions
- Converting angle measurements to radians
- Program declaration of constants
- Defining procedures
- Adding parameters to procedure definitions
- Defining functions
- Simple uses of sets
- Generation of random numbers
- The scope of variables
- Formal and variable parameters

DRAWING A ROTATED SQUARE

If we wish to draw a rotated square, none of MacPascal's built-in rectangle procedures will help us in the least. They will only draw vertical and horizontal lines. We will have to draw each side of the square separately using one of the line drawing procedures. Let's create a program first that will draw one square at a specified rotation.

First, we must examine the calculations that determine the end points of an angled line. Figure 7-1 illustrates such a line. It begins at coordinates (0,0), is rotated at an angle of 30 degrees, and has a length of 100. Our job is to determine the coordinates that describe the other end of the line.

The dotted lines in Fig. 7-1 illustrate that the angled line may be thought of as the hypotenuse of a right triangle. By applying calculations using appropriate trigonometric ratios, we can determine the dimensions of the two sides of the triangle. These dimensions correspond to the x and y coordinates for the unknown end of the line.

I will not explain why the formulas I use will produce the correct results. Such an explanation would require a chapter by itself. If you are curious, consult any good text on trigonometry. Otherwise, you will find that you can use the formulas quite well without understanding exactly why they work.

(If this approach bothers you, consider that we use many things in our lives without understanding how they work. Almost certainly, you do not know what goes on inside your Macintosh, and you are probably almost as naive about your television or even your car. But that does not stop you from using these things. We can use a mathematical formula or a television as a "black box." We know what goes in and what comes out, but we need not understand what happens inside. If you are still curious about the whys and wherefores of the formulas, marvelous! Please do learn all about them. But, for the rest of you, just relax and use them.)

The formula for the new x coordinate is: current x+COS (ANGLE) * LENGTH.
The formula for the new y coordinate is: current y+SIN (ANGLE) * LENGTH.

Pascal provides sine and cosine functions, so it is no problem to carry out these calculations. But there is one catch. Pascal expects angles to be expressed in *radians*, a measure that is based on the value of π. Conversion from degrees to radians is fairly simple. 360 degrees is equivalent to 2π ra-

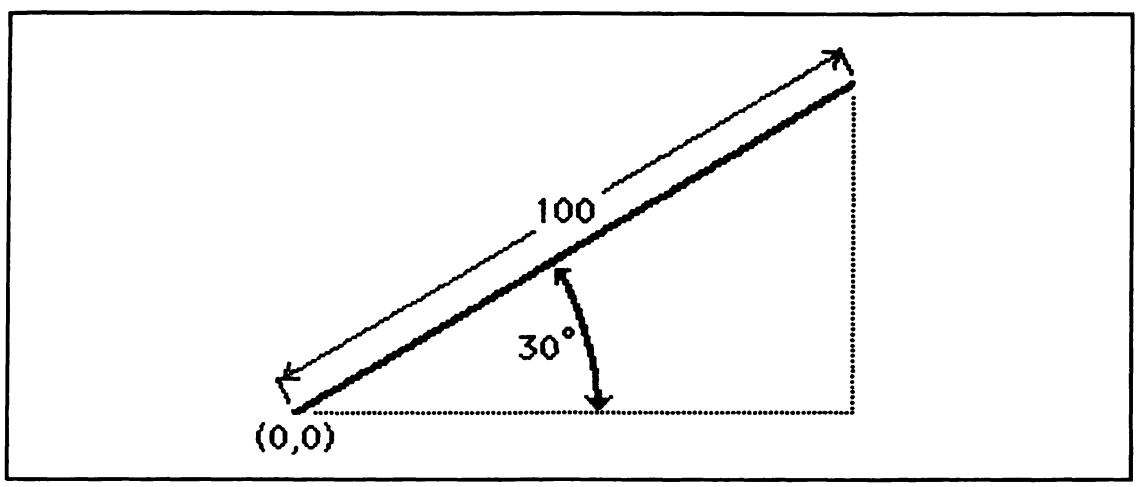


Fig. 7-1. An angled line and the right triangle associated with it.

dians. That is, 360 degrees = 2 * 3.14159 radians. Again, don't worry too much about radians. The formulas are simple, and we will discover techniques in this chapter that make the conversion quite painless.

To determine the radians equivalent for a measurement in degrees

1. Calculate the fractional part of a circle that the degrees measurement reprsents. For example, 45 degrees is one eighth of a circle. This ratio is determined by dividing the measurement by 360. 45 divided by 360 is .125 which is one eighth.
2. Multiply the fraction just calculated by 2π. Since a full circle is 2π radians, an eighth of a circle is

   .125 * 2 * π,

   which is

   .125 * 2 * 3.14159.

   Thus, 45 degrees is equivalent to .78540 radians.

Combining the formula for converting degrees to radians with the formula for determining the new endpoint coordinates, the complete Pascal expressions to determine the unknown x and y coordinate are:

```
newx := currentx + round(cos ( rotation / 360 * 2 * 3.14159) * length)

newy := currenty + round(sin ( rotation / 360 * 2 * 3.14159) * length)
```

The rounding is done so that the results may be used by procedures that require integer parameters. Again, if you do not follow the explanation of these expressions, please do not be alarmed. Before long, you will perform conversions between degrees and radians almost painlessly. For now, all you need do is substitute the desired values into the variables ROTATION and LENGTH.

Let's apply our newfound knowledge to draw a single line, at an angle of 30 degrees. We will build on this program later to enable it to draw a complete square. Our approach will be to use the LINE procedure, which works a bit differently from the LINETO procedure we have used in the past for drawing lines.

LINETO(X,Y) draws a line from the present drawing location to the point described by (X,Y). Lines drawn by LINETO can have different lengths and angles, depending on the relative locations of the starting and ending points.

LINE(X,Y) uses X and Y to describe a point relative to the present drawing location. It says, "Draw a line from the current location to a point that is X points to the right and Y points below the current location." For a given set of parameters, the line drawn by LINE will always have the same length and angle. Only the location will change, as determined by the starting point. This is the procedure to use for our purposes, since it will let us draw the same shape square anywhere in the Drawing window, using the same procedure calls.

Here is a program that draws one angled line:

```
program anglesquare;
  var
    change1, change2, rotation, length : integer;
begin
  showdrawing;
  rotation := 30;
  length := 50;
  change1 := round(cos( rotation / 360 * 2 * 3.14159) * length);
  change2 := round(sin( rotation / 360 *2 * 3.14159) * length);
  moveto(100, 100);
  line(change1, change2);
end.
```

■ Enter this program, execute it, and observe the results. We know that the starting point for the line was at the center of the Drawing window, MOVETO set the drawing point to 100,100. Therefore, the line angled down, where the line in Fig. 7-1 angled up. Does this mean that our formulas were wrong?

The line angled down because Macintosh Pascal unfortunately inverts the normal coordinate system. Normally, y coordinates increase in value as we move upward on the coordinate grid, and negative coordinates extend toward the bottom of the coordinate system. On the Macintosh screen, y coordinate values increase as we move *downward*. So, everything is working properly with the program, even though the results are upside down. We could control this inversion, but in this case the program will work fine in spite of it.

We can simplify the formulas by introducing a program-defined constant. Constants are similar to variables in that a value is assigned to an identifier. The difference is that, once assigned, the value of a constant cannot be changed. There are several reasons for using constants. The present reason is program clarity. Here is the same program, using a CONST block:

```
program anglesquare2;
  const
    pi = 3.14159;
  var
    change1, change2, rotation, length : integer;
begin
  showdrawing;
  rotation := 30;
  length := 50;
  change1 := round(cos( rotation / 360 * 2 * pi) * length);
  change2 := round(sin( rotation / 360 *2 * pi) * length);
  moveto(100, 100);
  line(change1, change2);
end.
```

A CONST block is used to assign values to constant identifiers. Following this declaration, the identifier may be used anywhere a value might appear in the program. The CONST block must appear before the VAR block in the declaration part of a program.

This results in a small but worthwhile improvement in program readability, since we need no longer

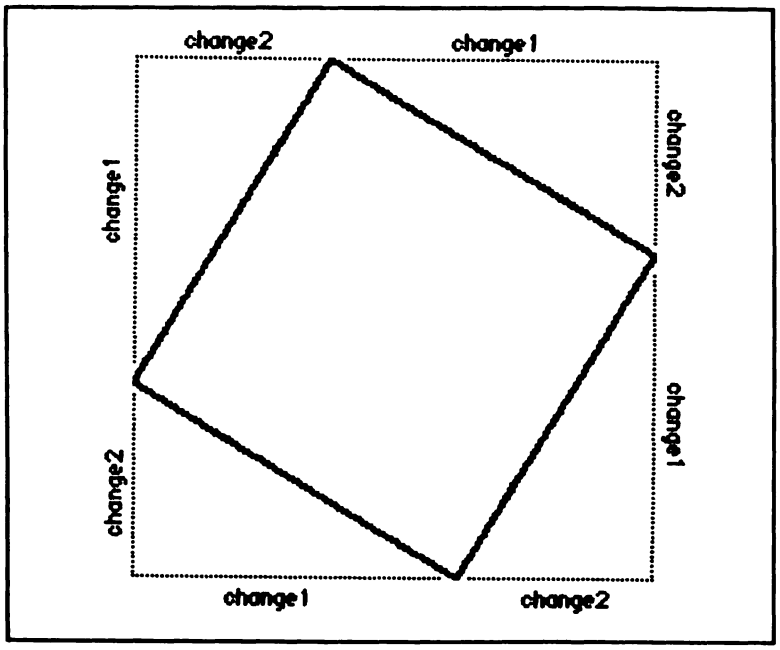


Fig. 7-2. The dimensions for the completed square.

interpret the number 3.14159. Of course, the value of PI is fairly familiar, and the interpretation is minor. This is no always the case, however, and constants properly used can make programs much more readily understandable.

Figure 7-2 shows the rest of the square. Dotted lines mark the right triangles that are associated with the four sides of the square we wish to draw. Notice that the values we have calculated and stored in CHANGE1 and CHANGE2 may also be used in various combinations to describe all four sides of the square. It is a simple matter to complete the square.

```
program anglesquare3;
  const
    pi = 3.14159;
  var
    change1, change2, rotation, length : integer;
begin
  showdrawing;
  rotation := 30;
  length := 50;
  change1 := round(cos( rotation / 360 * 2 * pi) * length);
  change2 := round(sin( rotation / 360 *2 * pi) * length);
  moveto(100, 100);
  line(change1, change2);
  line(-change2, change1);
  line(-change1, -change2);
  line(change2, -change1)
end.
```

A positive parameter in LINE causes the end point of the line to move either right or down. To move the end point left or up, it is necessary to negate a parameter. You will find that this program will draw squares of any orientation or size simply by varying the values of ROTATION and SIZE.

Suppose that we wished to draw several such squares at different locations, having varying sizes and rotations. How could we modify the program to do this? How would you create such a program right now to draw the following four squares in the same Drawing window?

1. At 50,25 with a size of 50 and a rotation of 30,
2. At 150,50 with a size of 25 and a rotation of 45,
3. At 60,50 with a size of 40 and a rotation of 10, and
4. At 150,150 with a size of 75 and a rotation of 120.

About the only strategy you would have available would be to repeat almost all of the statements for each square, using different values for SIZE and ROTATION. (You could use a loop, and read in the values for the variables from the keyboard, but I would like the program to be completely automatic, so we won't allow that strategy.)

The problem would be trivial if you had a procedure that drew such rotated squares. You would then draw angled squares in about the same way you have drawn squares with FRAMERECT. Suppose you had a ROTSQUARE procedure to perform the four LINE statements each time. You could simplify the program to something like this:

```
program foursquares;
  var
    rotation, size : integer;
begin
  showdrawing;

  moveto(50,25)
  size := 50;
  rotation := 30;
  rotsquares;

  moveto(150,50)
  size := 25;
  rotation := 45;
  rotsquares;

  moveto(60,50)
  size := 40;
  rotation := 10;
  rotsquares;

  moveto(150,150)
  size := 75;
  rotation := 120;
  rotsquares
end.
```

Notice the blank lines which I have added to break up the program into functional units. Pascal will ignore blank lines, so we can include them freely.

If you read the introduction to the chapter, you are probably ahead of me. Yes, we can create a procedure ROTSQUARES. And, when we are done the program above will be simplified so that it requires only four procedure calls to perform all of that work.

DEFINING PROCEDURES

Pascal procedures look very much like Pascal programs. In fact, a procedure can be considered as a miniprogram or a subprogram controlled by a main program. Here is a first-version procedure that we will use to draw angled squares:

```
procedure rotsquare;
  const
    pi = 3.14159;
  var
    change1, change2 : integer;
begin
  change1 := round(size * cos(rotation / 360 * 2 * pi)
  change2 := round(size * sin(rotation / 360 * 2 * pi)
  line(change1, change2);
  line(-change2, change1);
  line(-change1, -change2);
  line(change2, -change1)
end;
```

Taken by itself, a procedure looks very much like a program. It contains a statement section, bounded by BEGIN and END. It can contain constant and variable declaration parts. In fact, it can contain just about anything that a program can contain. However, a procedure is intended to be used by a program. By defining a procedure, we add to the program's vocabulary.

An important distinction between programs and procedures is the final punctuation. Procedures are terminated with a semicolon. Only a program is terminated with a period.

Procedures fit into programs between the declaration part and the statement part:

```
program foursquares;
  var
    rotation, size: integer;

  procedure rotsquare;
    const
      pi = 3.14159;
    var
      change1, change2: integer;
  begin
    change1 := round(size * cos(rotation / 360 * 2 * pi));
    change2 := round(size * sin(rotation / 360 * 2 * pi));
```

```
    line(change1, change2);
    line(-change2, change1);
    line(-change1, -change2);
    line(change2, -change1)
  end;

begin { main program }
  showdrawing;

  moveto(50, 25);
  size := 50;
  rotation := 30;
  rotsquare;

  moveto(150, 50);
  size := 25;
  rotation := 45;
  rotsquare;

  moveto(60, 50);
  size := 40;
  rotation := 10;
  rotsquare;

  moveto(150, 150);
  size := 75;
  rotation := 120;
  rotsquare
end.
```

■ When you enter and execute the program, you will find that it does everything we wanted it to do, at a considerable savings in size. But it can be made still simpler by eliminating some of the assignment statements in the main program. In fact, we can reduce the main program to just five lines by placing even more of the statements in the procedure definition.

Currently, ROTSQUARES gets its information about size and rotation from values assigned to variables in the main statement part. To simplify the main program, it would be nice if we could reduce the number of assignment statements.

The problem we face is how to pass information to the procedure about the size and the rotation of the square without using assignment statements? Actually, we have been doing something similar all along when we have used parameters to modify the actions of built-in procedures. The statement FRAMERECT(10,10,50,100) is passing four values to the FRAMERECT procedure, without requiring assignment statements. We can modify our procedure to give it an identical capability.

Here is the new version of the procedure:

```
procedure rotsquare (x, y, size, rotation : integer);
  const
    pi = 3.14159;
  var
    change1, change2: integer;
begin
  moveto(x,y);
  change1 := round(size * cos(rotation / 360 * 2 * pi));
  change2 := round(size * sin(rotation / 360 * 2 * pi));
  line(change1, change2);
  line(-change2, change1);
  line(-change1, -change2);
  line(change2, -change1)
end;
```

The really new part is the section in parentheses following ROTSQUARES. This section contains the parameter definitions for the procedure. The parameters are variables, which may be used only within the procedure itself. This procedure has four parameters: X, Y, SIZE, and ROTATION. The order is important, for it indicates the order in which we must place the data in the parameter portion of the procedure call. All four parameters are declared to be of type Integer. Later we will see examples of procedures that use other types of parameters. Within the procedure, these parameter identifiers function as variables, just as if they had been declared in a VAR block.

Since parameters are now required, a procedure call will look something like this:

```
rotsquare(75, 100, 50, 35)
```

This would produce a square with the starting corner at an x coordinate of 75 and a y coordinate of 100. The size would be 50, and the rotation would be 35. Using parameters, the statement part of the program becomes quite simple indeed. Notice that SIZE and ROTATION need no longer appear in the VAR block, since they are declared as parameters in the procedure.

Here is a final version of FOURSQUARES.

```
program foursquares2;
  var
    change1, change2 : integer;

  procedure rotsquare (x, y, size, rotation : integer);
    const
      pi = 3.14159;
    var
      change1, change2 : integer;
  begin
    moveto(x,y);
    change1 := round(size * cos(rotation / 360 * 2 * pi));
    change2 := round(size * sin(rotation / 360 * 2 * pi));
    line(change1, change2);
```

```
    line(-change2, change1);
    line(-change1, -change2);
    line(change2, -change1)
  end;

begin { main program }
  showdrawing;
  rotsquare(50, 25, 50, 30);
  rotsquare(150, 50, 25, 45);
  rotsquare(60, 50, 40, 10);
  rotsquare(150, 150, 75, 120)
end.
```

The more times a procedure is used, the more it will save in terms of program size. But small program size is not the only or even the most important reason for defining procedures.

Properly used, defined procedures make the operation of a program more clear. Compare the latest version of the program with the program ANGLESQUARE, which we started out with. Suppose we had added the statements to draw the four squares, applying the same approach that we used in ANGLESQUARE? Would the program be as easy to understand? Would it be clear that the purpose of the program was to draw four squares at different locations? I trust that you will agree that this last version of FOURSQUARES as by far the easier one to read and understand. And, it would be the easier to modify if we wished to change the locations, numbers, or characteristics of the squares.

Yet another benefit offered by procedures is easy transportability. If you need another procedure like ROTSQUARES in a program you are writing, you can simply copy the text for the procedure from one program to the other, using the program editor of MacPascal.

Now that you have a working ROTSQUARES procedure, it will be very simple indeed for you to create this new program:

```
program spisquares;
  var
    i, j, size : integer;

  procedure rotsquare (x, y, size, rotation : integer);
    const
      pi = 3.14159;
    var
      change1, change2 : integer;
    begin
      moveto(x,y);
      change1 := round(size * cos(rotation / 360 * 2 * pi));
      change2 := round(size * sin(rotation / 360 * 2 * pi));
      line(change1, change2);
      line(-change2, change1);
      line(-change1, -change2);
```

```
    line(change2, -change1)
  end;

begin  { main }
  showdrawing;
  for i := 0 to 11 do
    begin
      j := 30 * i;
      size := 10 * i;
      rotsquare(100, 100, size, j);
    end
end.
```

DEFINING PASCAL FUNCTIONS

Earlier in the chapter, I promised that we would make it easy to convert degrees to radians. In fact, we will now proceed to develop a Pascal word that makes the conversion automatic. This Pascal word will refer to a user-defined *function*. Let's pause a moment to recall some functions we know, and to contrast them with procedures.

Several of the built-in procedures we have used are WRITELN, READ, FRAMERECT, NOTE, and MOVETO. All procedures have in common this feature: they perform some action. Optionally, a procedure may accept some data in the forms of parameters. But the important thing is that procedures perform a task.

We have also encountered several built-in functions, including TRUNC, ROUND, SIN, and COS. It is the purpose of all functions to perform some operation and to output a result. TRUNC inputs a real number and outputs the integer portion. SIN inputs a number and outputs the sine of that number. The common thread joining all functions is that they produce some value for use in the program.

Perhaps the clearest working distinction between a function and a procedure is the fact that a function is never used alone. It is always used by something else that requires the value that it outputs. Procedures, as we know, can do quite well on their own. It is perfectly all right to have a statement like:

```
writeln(round(5.123))
```

But we can never have a statement that says simply:

```
round(5.123)
```

For our purposes, we require a function that accepts a value in degrees and outputs a value in radians. The general formula has already been introduced. All that remains is to place the formula into proper form for a function:

```
function radians (degrees : real) : real;
  const
    pi = 3.14159;
  begin
    radians := 2 * pi * degrees / 360
  end;
```

Using this function, if an angle measurement in degrees is stored in ANGLE, the sine of the angle could be found using this statement:

```
writeln(sin ( radians ( angle )))
```

This is more clear and direct than

```
writeln ( sin( (2 * 3.14159 * angle / 360)))
```

Another advantage is that the word "radians" clearly tells us what is going on. Clarity in programming is an advantage that cannot be overemphasized.

Let's take the RADIANS procedure apart. While it looks superficially like a procedure, we must make note of some differences. First, let's take a look at the heading:

```
function radians ( degrees : real) : real;
```

This line accomplishes several things:

- It names the function RADIANS
- It declares the parameter DEGREE and states that its type is real,
- It declares that the output, the value produced by the function, will be real.

It is this last point that clearly distinguishes functions from procedures. A function always outputs a value, and the type of that value is declared along with the parameters in the function heading.

An essential event in a function is always the assignment of the output value to the function identifier, as in this statement in RADIANS:

```
radians := 2 * 3.14159 * degrees / 360
```

The function identifier has a dual identity. Within the function, it usually behaves as a variable.

Outside of the function itself, the identifier behaves just like a built-in Pascal function identifier. It makes no sense to say

```
round(3.456) := 2 + 1
```

since ROUND is not a variable in this instance and cannot be assigned a value. Similarly, we cannot state

```
radians(45) := 3.1415926 / 4
```

Functions appear in the same area of the program as procedures: after any declarations and ahead of the main program. There is one catch. If one function or procedure calls another, *the calling function or procedure must appear after the one that is being called.*

First, let's look at a program that works properly. Here is a version of SPISQUARES that uses the RADIANS procedure:

```
program spisquares2;
  var
    i, j, size : integer;

  function radians (degrees : real) : real;
  begin
     radians := 2 * 3.14159 * degrees / 360
  end;
```

```
procedure rotsquare (x, y, size, rotation : integer);
  var
    change1, change2 : integer;
begin
  moveto(x, y);
  change1 := round(size * cos(radians(rotation)));
  change2 := round(size * sin(radians(rotation)));
  line(change1, change2);
  line(-change2, change1);
  line(-change1, -change2);
  line(change2, -change1)
end;


begin  { main }
  showdrawing;
  for i := 0 to 11 do
    begin
      j := 30 * i;
      size := 10 * i;
      rotsquare(100, 100, size, j);
    end
end.
```

■ Try the program as shown to confirm that everything is working properly.

■ Move the function RADIANS so that it appears after the procedure and before the main statement part of the program. How does Pascal respond when you try to execute the program?

As Pascal prepares to execute a program, it scans the program, starting from the top. Among the things Pascal does during this process is to learn the definitions of any new words you have included. If a spot is reached that calls on a word that has not yet been defined, Pascal gets confused and generates an error message. Technically we would say that Pascal does not permit any *forward references* to identifiers.

To illustrate the diversity of possible functions, we will develop a couple more. The first will be a Boolean function that will make it much easier to obtain and process yes-no answers such as the ones required by the CAR programs in the last chapter. This function will request a yes or no answer and translate it into a Boolean value:

```
function yes_answer : boolean;
  var
    response: char;
begin
  write ('Answer y or n: ');
  read(response);
  yes_answer := response = 'y'
end;
```

The assignment statement in this function is interesting. If it confuses you at first, recall that ":=" is used to assign values while "=" makes comparisons. Before a value can be assigned to YES_ANSWER, Pascal must evaluate the expression to the right of the assignment operator. The expression "response = 'y'" will output either TRUE or FALSE, which can then be assigned to YES_ANSWER.

It is worth noting that YES_ANSWER has no parameters. In this case, none are required.

This procedure can simplify keyboard input considerably. Here is one last version of the CAR program from Chapter 6. Notice the savings within the main program and the reduction in duplication. The more yes-no responses the program calls for, the more valuable the function becomes.

```
program car3;
  var
     highcost, sportscar : boolean;      {remove RESPONSE variable}

  function yes_answer : boolean;                    {add function}
     var
       response: char;
  begin
     write ('Answer y or n: ');
     read(response);
     writeln;
     yes_answer := response = 'y'
  end;

begin
  showtext;
  write('Do you want to spend more than $30,000? ');
  highcost := yes_answer;                          {add function call}
  writeln('Do you want a sports car? ');
  sportscar := yes_answer;                         {add function call}
  if highcost then
     if sportscar then
       writeln('Buy a Porsche.')
     else
       writeln('Buy a Mercedes.')
  else if sportscar then
       writeln('Buy a Mustang.')
     else
       writeln('Buy an Oldsmobile.')
end.
```

The function still has a serious shortcoming. Whenever a program requests a user response, it should have some plan concerning how to handle unanticipated responses. You may be a mistake-free typist, but I certainly am not and neither are the majority of people who will use your programs. What

if someone misses the y key and types u? Equally possible is that your user might type an uppercase Y, which is not equivalent to the lowercase character. In each case, the procedure will react as if a "no" response had been entered.

One solution would be to use a more complicated test. Using the tools we have on hand, the most direct route would be to use several OR operators to test for all acceptable answers. If the character that was typed is not an acceptable one, the function should keep requesting characters until an acceptable one is entered. Here is a REPEAT..UNTIL loop that will continue to request responses until an acceptable one is entered:

```
repeat
  write('Answer y or n: ');
  read(input);
  writeln
until (input = 'y') or (input = 'Y') or (input = 'n') or (input = 'N');
```

This works well. The loop will not terminate until one of the four acceptable responses is typed. But I would like to show you an easier way of writing such a statement by introducing the Pascal *set*. The set is another Pascal data type, but we need not yet worry about the details. When a set is used, the UNTIL clause in that last REPEAT loop can be simplified to this:

```
until input in ['y', 'Y', 'n', 'N'];
```

A set is a collection of values. When a set is represented, the elements are enclosed in square brackets and separated by commas. All elements in the set must belong to the same type, and the type must be *scalar* (Integer or character, for example). In the above example, ['y', 'Y', 'n', 'N'] is a constant of the type Set. Thus, the *elements* of the set are of the type Char.

IN is a relational operator that works only on sets. In the above example, IN will output TRUE if the value of INPUT is a member of the set. Proper use of IN can save you a great deal of trouble. Remember that the test value must match some value among the members of the set to produce a TRUE result. In this final version of YES_ANSWER, two IN operations are performed:

```
function yes_answer : boolean;
    var
      input : char;
  begin
    repeat
      write('Answer y or n: ');
      read(input);
      writeln
    until input in ['y', 'Y', 'n', 'N'];
    if input in ['y', 'Y'] then
      yes_answer := true
    else
      yes_answer := false
  end
```

This is a pretty bomb-proof entry function. When you are writing a program that will be used by others, you should attempt to ensure that no input will cause the program to malfunction. This is not always easy to accomplish, but it is definitely worthwhile when done.

The last function we will develop in this chapter will be one to generate random numbers, which are frequently used in computer games. Macintosh Pascal provides a built-in RANDOM procedure, but we must do some work before it will produce the results we want.

■ To demonstrate the built-in RANDOM function, try this program. You will probably want to enlarge the Text window to make room for all the numbers that will be produced.

```
program randomdemo;
  var
    i : integer;
begin
  showtext;
  for i := 1 to 50 do
    write(random);
end.
```

The numbers produced cover a broad range. In fact, RANDOM can output values that span the entire range of integer values from −32768 to 32767. But to simulate the throwing of dice, as one example of a gaming application of random numbers, we need to generate values from 1 to 6. You can see why I indicated that the RANDOM function would need some work.

First let's turn our attention to limiting the range of numbers that will be produced. While we are at it, we will begin to build a more useful random function.

For the first time, we find a use for the MOD operator. MOD was introduced earlier as a companion to DIV. Whereas DIV performs integer division, MOD determines the *remainder* of an integer division. This is just what we need. Here is the demonstration program, this time incorporating a first version of the RAND function, which we will be developing.

```
program randomdemo2;
  var
    i : integer;

  function rand (toplimit : integer) : integer;  {new function}
  begin
    rand := random mod toplimit
  end;

begin
  showtext;
  for i := 1 to 50 do
    write(rand(6));                          {change function call}
end.
```

■ Make the necessary changes and execute the program. You will see that the numbers produced are distinctly different from those seen earlier.

TOPLIMIT is assigned a value of 6 when RAND is called. MOD is then used to find the remainder of RANDOM DIV TOPLIMIT. This has eliminated the negative numbers (there are no negative remainders) and has nicely limited the range of the printed numbers, but they are not quite the numbers we need when simulating dice.

The remainder of a division will always be less than the divisor. This means that the result of RANDOM MOD 6 will never be 6. At the other end of the scale, zeros will be produced whenever RANDOM is evenly divisible by 6. So the numbers being produced range from 0 to 5, making it a simple matter to correct things just by adding 1.

■ Edit the statement part of the RAND function like this:

```
rand := random mod toplimit + 1
```

Since MOD enjoys a higher priority than +, no parentheses are required. Now the function will add one to the results of the MOD calculation.

■ Try the program again. Are the numbers correct this time?

In fact, this is exactly what we want, and we could stop here. But I would like to extend the procedure so that it can produce random numbers for any range of positive integers. For this, we will require a new parameter, LOWLIMIT, and some additional calculations. The lower limit of numbers produced could be raised simply by adding the lower limit to the results of the MOD operation. Edit the RAND function, adding the LOWLIMIT parameter to the parameter list:

```
function rand (lowlimit, toplimit : integer) : integer;
begin
  rand := random mod toplimit + 1 + lowlimit
end;
```

To use the function, you must now pass two parameters to it. Let's try to generate a set of numbers ranging from 3 to 6.

■ Edit the WRITE statement to read:

```
write(rand(3, 6));
```

■ Execute the program and observe the printed numbers. Fairly obviously, they do not fall within the desired range.

When we raised the lowest value, we pushed up all of the numbers produced. The next step is to restrict the upper limit. We do this by reducing the range of the outputs from MOD. Since we have raised the lower limit by 3, let's try to reduce the range of MOD outputs by the same amount. This is done by subtracting LOWLIMIT from TOPLIMIT.

■ Make this modification and try the program again:

```
rand := random mod (toplimit - lowlimit + 1) + lowlimit
```

The function finally does everything we want it to do, actually more than we need right now. Let's put it to work.

First, let's see how evenly distributed the random numbers are. If we are going to write games that use them, it would be nice to know that the computer rolls fair dice. We will just do a visual check, not a careful statistical analysis. Our method will be to have the computer toss a single die a fairly large number of times, keeping track of the results of the rolls. At the end, if each face of the computer die has appeared about the same number of times, we will judge the computer die to be fair.

There is nothing terribly special about the program, so let's jump right in:

```
program dicecount;
  var
    i, roll, ones, twos, threes, fours, fives, sixes : integer;

  function rand (lowlimit, toplimit : integer) : integer;
  begin
    rand := random mod (1 + toplimit - lowlimit) + lowlimit
  end;

begin
  ones := 0;
  twos := 0;
  threes := 0;
  fours := 0;
  fives := 0;
  sixes := 0;
  showtext;
  for i := 1 to 100 do
    begin
      roll := rand(1, 6);
      case roll of
        1 :
          ones := 1 + ones;
        2 :
          twos := 1 + twos;
        3 :
          threes := 1 + threes;
        4 :
          fours := 1 + fours;
        5 :
          fives := 1 + fives;
        6 :
          sixes := 1 + sixes
      end;  {of case}
    end;  {of for loop}
  writeln('Total of ones:', ones);
  writeln('Total of twos:', twos);
  writeln('Total of threes:', threes);
  writeln('Total of fours:', fours);
  writeln('Total of fives:', fives);
  writeln('Total of sixes:', sixes)
end.
```

■ Execute the program to confirm that it is operating properly. You will probably find that the counts for the six values are not particularly equal. This is the result of the relatively small number of loop iterations. As we increase the FOR statement's upper limit, things will even out.

■ Next, change the upper limit value of the FOR loop to 1000. Execute the program and note the new values of the counter variables. They should be getting closer.

■ Finally, change the upper limit to 30000. This is about as high as we can go without exceeding the maximum allowable integer value of 32767. Execute the program and wait. Notice that some time is required to complete thirty thousand passes through the loop—about six minutes. This will be the first time you have seen—or rather not seen—a program require a substantial time to execute a task.

Even if the numbers were perfectly random, you would probably never observe all six values being equal. Computer random numbers are not truly random in the sense that numbered balls pulled from a rotating hopper could be considered randomly arranged. Truly random numbers will not have a pattern, and they will be evenly distributed over the full range of their values. Computer numbers only approximate randomness. They are produced through a complex series of calculations, and for that reason they will exhibit a pattern. However, for our purposes, they serve quite well. Our experiments have shown that with a large enough sample, a fairly even distribution is produced. That is generally enough to satisfy our needs.

Be sure to save this program. We will be returning to it several times in future chapters.

MOVING DATA IN AND OUT OF PROCEDURES AND FUNCTIONS

Without going into great detail, we have already seen two methods of passing data to procedures and functions, through program variables and through parameters. There is a great deal more to this issue, and we need to go into it before moving on.

■ Here is a simple program, with a single procedure. Type it in and execute it:

```
program datademo;
  var
    a, b : integer;

  procedure printdata;
  begin
    writeln('In PRINTDATA the value of A is:', a);
    writeln('In PRINTDATA the value of B is:', b)
  end;

begin          {main program}
  a := 2;
  b := 3;
  writeln('A is:', a);
  writeln('B is:', b);
  printdata;
  writeln('A is:', a);
  writeln('B is:', b)
end.
```

First the values of A and B are printed in the main program. Then they are printed in PRINTDATA. Finally, they are printed in the main program again. Not surprisingly, you will find that the values of A and B are the same each time they are printed. What could happen to change that?

■ Insert the following statements before the WRITELN statements in PRINTDATA:

```
a := 22;
b := 33;
```

■ Execute the program. Still no surprises—A was assigned the value of 22 by PRINTDATA before execution of the second pair of WRITELN statements in the main program, so we expect A to have a value of 22 the last time it is written. The same sort of thing has happened to B.

■ Add this VAR block to the procedure PRINTDATA:

```
var
  a : integer;
```

■ Execute the program. Now you will see something unusual. The WRITELN statement in PRINTDATA shows that A has a value of 22 within the procedure. However, both sets of WRITELN statements in the main program show that A has a value of 2. The value of A in the main program was not changed by the assignment in the procedure!

The explanation is that there are two distinct variables named A in this program: one for the main program and one within the procedure PRINTDATA. Each is declared in a variable declaration, and they have nothing whatever to do with each other.

The Scopes of Program Variables

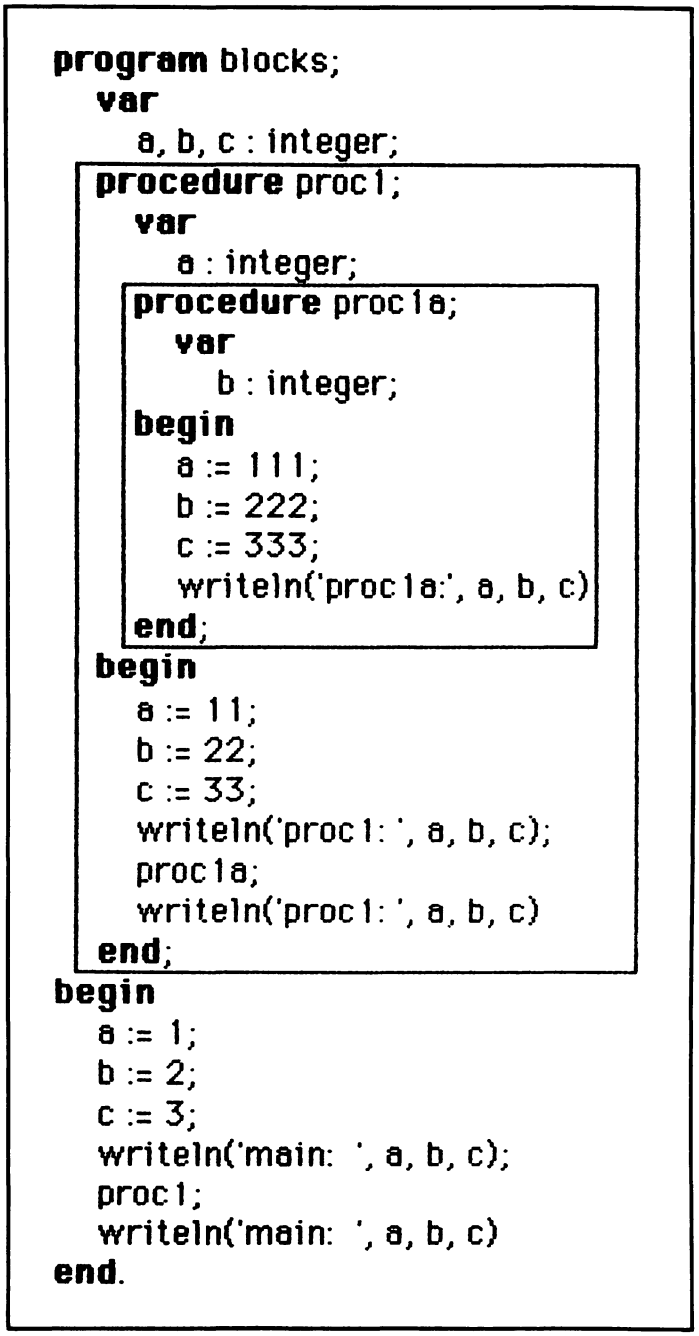
We must now enter on a discussion regarding the *scopes* of program variables. Pascal programs and their components may be illustrated as nested boxes, such as the ones shown in Fig. 7-3. Until now, the variables we have used have been equally available anywhere in the program. Any procedure, or function can use a variable declared in the main program declaration block. In the first version of DATADEMO, which did not have a variable declaration part in the procedure, the variables A and B could be written and their values could be altered anywhere in the main program or in the subroutine.

Any variable that is declared by the main program is said to be *global* in scope, which is to say that it is accessible anywhere in the program.

However, each section in the program can have its own variable environment. In Fig. 7-3, each box, more properly called a *block*, represents a variable *environment*. If a variable is declared in a block, it will generally be available in any block that is contained by that block. However, the variable will not be available in parts of the program that fall outside of that block. The scope of a variable describes which sections of the program may have access to the variable.

When a variable is declared within a procedure or a function, that variable is limited in scope. It may not be changed or referenced outside of the procedure or function that declared it. Such a variable is said to have a *local* scope. If a local variable is declared having the same name as an existing global variable, the procedure cannot access the global variable.

Examine the program that appears in Fig. 7-3. Notice that the procedure PROC1A is contained within the procedure PROC1. Earlier I stated that procedures and functions could contain just about anything that a program could contain. This property extends so far that procedures and functions can contain their own procedures and functions. Since PROC1A is contained by PROC1, it is local to that block, and cannot be accessed by the main program.



```
program blocks;
  var
    a, b, c : integer;
  procedure proc1;
    var
      a : integer;
    procedure proc1a;
      var
        b : integer;
    begin
      a := 111;
      b := 222;
      c := 333;
      writeln('proc1a:', a, b, c)
    end;
  begin
    a := 11;
    b := 22;
    c := 33;
    writeln('proc1: ', a, b, c);
    proc1a;
    writeln('proc1: ', a, b, c)
  end;
begin
  a := 1;
  b := 2;
  c := 3;
  writeln('main: ', a, b, c);
  proc1;
  writeln('main: ', a, b, c)
end.
```

Fig. 7-3. An example of a program block structure.

You might wish to type in this program and execute it as a further illustration of the properties of local variables. Can you explain the printed results?

You can see that a program can contain any number of variables that have the same name, but Pascal functions to prevent any confusion. This is exactly the phenomenon we have just observed.

The Use of Parameters

As you remember, parameters, as well as global variables can be used to transmit data to procedures and functions. Let's examine some more modifications to the program DATADEMO.

■ Add a parameter to the procedure PRINTDATA by editing the heading like this:

```
procedure printdata (b : integer);
```

■ Remove this assignment statement, which you added to PRINTDATA:

```
b := 33;
```

■ Finally, modify the procedure call in the main program to:

```
printdata(b);
```

■ Execute the program and observe the results. The same value is held by B in both the main program and in the procedure. Now we must ask whether there are one or two variables named B in the program.

■ Add this line immediately after the BEGIN in PRINTDATA:

```
b := b * 2;
```

■ When you execute the program, you will discover that the multiplication affected only the B variable that was in the procedure, which now has a value of 6. The B variable in the main program still has a value of 3.

Variables that are declared as parameters of a procedure are local to that procedure. Now with the VAR block you added to the PRINTDATA procedure, both A and B are local variables of the procedure PRINTDATA.

You may be confused since the B variable was used to pass a value to the parameter B. But it is absolutely unimportant that the names are the same. A variable of any name can pass a value to a parameter of any name, provided only that their types are compatible. The variables function only to exchange values, not to share names.

Now that I have laid down that absolute principle, I am going to tell you of the exception to it. The parameters we have been using are called *value* or *formal parameters*. But there are other parameters called *variable parameters*. This sort has a distinctly different nature.

■ To introduce a variable parameter, edit the procedure like this. Notice that all references to a variable named A have been replaced by a variable named X.

```
procedure printdata (b : integer; var x : integer);
begin
  x := x * 2;
  b := b * 2;
  writeln('In PRINTDATA the value of x is:', x);
  writeln('In PRINTDATA the value of B is:', b)
end;
```

Pascal will reformat the procedure heading like this:

```
procedure printdata (b : integer;
                     var x : integer);
```

- Also change the procedure call in the main program to:

```
printdata(b, a);
```

- Execute the program and observe the results. See anything that needs explaining? How has A come to represent a value of 4 when that value is never in any way assigned to it? If X is a parameter of PRINTDATA, as seems likely, then A passed a value of 2 to X. But if parameters are local in scope, a change to the value of X should not have affected the value of A in the main program.

Formal (value) parameters operate by passing values from one variable to another. These two variables are completely independent of each other, and changing the value of the procedure's variable has no effect on the variable that was used in the procedure call. Formal parameters may be used only to pass data to a procedure. They do not allow operations in the procedure to affect the values of global program variables; this is usually a desirable situation.

Variable parameters operate by assigning two names to the same variable. Recall that a variable is a designated place in the computer's memory where data may be stored. The variable identifier is simply a label that makes it convenient to reference that storage place. A variable parameter assigns a second name to the same variable location. In the program as it now stands, both X and A affect the same variable location. Variable parameters may thus be used to pass data to and to obtain data from a procedure.

Variable parameters are declared within the parameter list following the word VAR. A semicolon separates them from the value parameters, which they must follow. Variable and value parameters may not be mixed within the parameter list. Every declaration that follows the VAR will be considered to be a variable parameter.

Variable parameters are useful when we require a function that will output two or more values. a Pascal function can output only one value. Using variable parameters there is no limit to the number of values that may be exchanged. Usually, variable parameters are used only in procedures. However, a function can use variable parameters, in which case it would pass one value as the function output and other values would be passed in variable parameters. While this is allowable, to my mind it is awkward and potentially confusing.

We have seen one built-in procedure that uses variable parameters. GETMOUSE(X,Y) places the current x and y coordinates of the mouse in the variables X and Y, which must have been declared by the program VAR section.

Here is a brief program that uses variable parameters. It calculates the areas and circumferences of two circles:

```
program circlesizes;
  var
    diameter, circum, area : real;
    dia, circ, ar : real;

  procedure circum_area (d : real;
                         var a, c : real);
  begin
    a := 3.14159 * (d / 2) * (d / 2);
```

```
    c := 3.14159 * d
  end;

begin
  diameter := 5;
  circum_area(diameter, area, circum);
  writeln('1st circle.');
  writeln('Diameter : ', diameter : 6 : 4);
  writeln('Area:', area : 6 : 4);
  writeln('Circumference:', circum : 6 : 4);
  writeln;

  dia := 20;
  circum_area(dia, ar, circ);
  writeln('2nd circle.');
  writeln('Diameter :', dia : 6 : 4);
  writeln('Area:', ar : 6 : 4);
  writeln('Circumference:', circ : 6 : 4)
end.
```

The main program consists of two almost identical sections. Each calls CIRCUM_AREA to perform the calculations of area and circumference as determined by the diameter specified. In the first instance, the values are returned in the variables AREA and CIRCUM. In the second, the values are returned in the variables AR and CIRC. Both work perfectly.

We can now begin to appreciate some reasons why variable parameters may be preferable to global variables when we must exchange values with procedures.

Recall that variable parameters actually manipulate the variable that appears in the calling parameter list. Now notice that the parameter lists in the two procedure calls have different variable names. Within the same program, we have three sets of variable names, and yet they may all be used to work with the same variables. This is handy in several situations.

If an existing procedure in another program performs a function we need, we can import the procedure with the program editor and install it in the program we are writing. If the procedure relies on global variables to exchange data, we must make sure that the variable names within the procedure agree with the variable names used in the rest of the program. If, however, the procedure uses variable parameters, names need not be matched. Procedure transfer between programs is greatly simplified.

If a procedure performs an operation that is frequently used within a program, it may not always be convenient to always call it with the same variable names. Perhaps you already have values in the previously used names that you wish to preserve. It may be convenient to use a new set of names. Procedures using variable parameters are independent of the names of the variable names used to call them. Your flexibility is greatly increased.

The RECTANGLES program in Chapter 5 is an example of such a situation. The program needed to know both the old position and the new position of the mouse. Because GETMOUSE uses variable parameters, this was very easy to do.

One more point about variable parameters: since they function to assign a second name to a variable, the parameter in the procedure call must be a variable. It does not work to say GETMOUSE(100,50),

for example. We must say something like GETMOUSE(X,Y) where X and Y are previously declared variables.

Let's summarize the points we have been making about variable scope:

- Global variables may be referenced anywhere in the program that a local variable of the same name is not in force. Global variables must be declared in the main VAR block of the program.
- Any variable declared within a procedure or a function will be local in scope. Only that procedure or function may use that variable. If a local variable name is the same as that of a global variable, the global variable may not be accessed when the local variable is in force.
- Local variables may be declared in procedure or function VAR blocks, in parameter lists, or as the names of functions.

Three methods are available for exchanging data between the program and procedures or functions. In the following discussion, everything that applies to procedures is also true of functions.

1. Global Variables

Advantages: Global variables are easy to use when many variables are involved. Long parameter lists can get cumbersome. They work well with values that affect many procedures; for example, the value of a current interest rate, which might be used in several places within the program.

Disadvantages: Procedures are difficult to transport. There may be side effects because the variable names in the procedures are not isolated from the variable names in the program. Procedures cannot be made completely independent of the program in which they are contained.

2. Value (formal) parameters

Advantages: Value parameters provide a very high level of isolation between procedure and program. the procedure cannot accidentally alter a program variable. Procedures are highly tranportable. They are best used when data need only be passed to procedure.

Disadvantages: Value parameters cannot pass data back to the calling program. Large numbers of parameters can make procedure calls confusing.

3. Variable parameters

Advantages: Value parameters permit very easy interchange of data between the program and the procedure. Procedures are highly transportable. Value parameters allow access to global variables using local names. More than one value may be output.

Disadvantages: Only variables may be used to call procedures; constants and expressions cannot be used. Large numbers of parameters can make procedure calls confusing.

YOUR PASCAL VOCABULARY

You now know the following Pascal words. Words that were new in this chapter are printed in bold face type.

Reserved Words

PROGRAM	BEGIN	END	VAR
DO	PROCEDURE	FUNCTION	CONST

Statement Types

Assignment (:=)	Compound		
FOR..TO	FOR..DOWNTO	WHILE	REPEAT..UNTIL

IF..THEN	IF..THEN..ELSE	CASE	

Data Types

BOOLEAN	CHAR	INTEGER	REAL
STRING			

Procedures

READ	READLN		
WRITE	WRITELN		
NOTE			
FRAMERECT	PAINTRECT	FRAMEOVAL	PAINTOVAL
INVERTRECT		INVERTOVAL	
MOVETO	LINETO	LINE	
PENPAT	PENSIZE	PENMODE	
BUTTON	GETMOUSE		

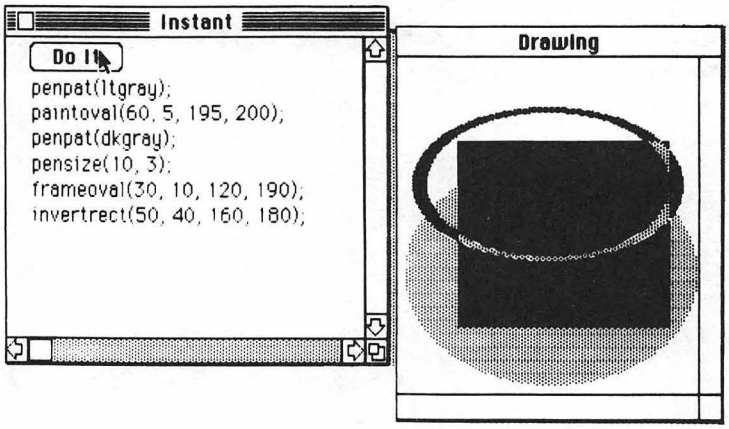
Operations

+	-	*	/
DIV	MOD		
>	>=	<	<=
<>	=		
NOT	AND	OR	IN

Functions

ROUND	TRUNC	SIN	COS
RANDOM			

Chapter 8



Data Types: Built-In and User-Defined

We have considered Pascal data types in several places, but much has been left unsaid. In this chapter, we will first look at the simple built-in Pascal data types: Real, Integer, Char, and Boolean. Following that discussion, we will learn how Pascal permits us to define our own data types that are tailored to specific program purposes.

TOPICS COVERED IN THIS CHAPTER

- Exponential representation of Real data
- Double and Extended Real data types
- How the results of evaluating an expression can exceed the capacity of a variable
- Problems with exact matching of real values
- Integers and Longintegers
- Ordinal types
- Declaring enumerated types and subranges

REAL DATA TYPES

Real numbers represent an unbroken and infinite sequence of values. For any two real numbers we can name, there is a real value that falls between them. For any real number we can think of, there is always a real number with a greater value and one with a lesser value. In short, there are no breaks in the scale of real numbers and no end-points.

Because computers are finite devices, however, there are limits to the real numbers that may be represented by Pascal. There are limits to the most positive and most negative values that may be

represented, as well as limits to the smallest fractional values that may be dealt with.

All real numbers are represented in *exponential* form. All such numbers have two parts: the decimal part, called the *mantissa*, and the *exponent*. The number 456.123 would be represented as 4.56123e2 by Pascal. The "4.56123" part is the mantissa. The "e2" represents a power of ten multiplier (exponent); in this case ten to the power of 2. To determine the decimal representation of 4.56123e2, we multiply 4.56123 by 100, which is ten to the 2nd power.

The normal practice with exponential representation is to place the decimal point after the first digit in the mantissa and to adjust the exponent accordingly. This results in a highly uniform format for real numbers, simplifying the computer's internal calculations.

To represent very small fractions negative exponents are used. 0.000123 is represented as 1.23e-4, which is 1.23 times 10 to the −4 power or 1.23 times 0.0001.

Here are some examples of decimal numbers and their exponential representations:

1.23456	1.23456e0
123456.0	1.23456e5
0.000000123	1.23e−7
−123.456	−1.23456e2
1234567890123.0	1.2345678e12

In the last example, notice that several digits from the original number were not included in the exponential form. There are limits to the sizes of mantissas and exponents that Pascal can represent. Standard real numbers are limited to seven or eight decimal digits and to exponents of −45 through 38. This is generally sufficient, but there are times when more precision is required.

In these cases, Macintosh Pascal provides the Double and the Extended real types. Here is a comparison of their limits of representation:

TYPE	RANGE	DECIMAL DIGITS
Real	1.5e-45 to 3.4e38	7-8
Double	5.0e-324 to 1.7e308	15-16
Extended	1.9e-4951 to 1.1e4932	19-20

These higher precision types are simply declared in the VAR program block instead of Real. The trade-off in selecting real types is that the high precision types require considerably more memory for storage. That is not very important now, when our programs only have a dozen or so variables, but we will soon see programs that have hundreds of variables where memory usage can begin to become a problem.

Macintosh Pascal is somewhat forgiving when data of different precision are mixed. The value of a Real type variable can always be assigned to a Double or an Extended type variable. The value of a Double type variable can always be assigned to an Extended type variable. Thus there is never a problem when moving data to a variable of higher precision.

MacPascal will also permit data to be moved from higher to lower precision variables, provided the data fit the lower precision variable. If the value of an Extended type variable has less than eight digits and an exponent between −45 and 38, the value may be assigned to a Real type variable.

In fact, this is done all the time, since all calculations with real numbers are carried out using the type Extended. This means that the results of a calculation may not fit back into a Real variable.

Here is a simple program that illustrates this problem:

```
program rangetest;
  var
    n : real;
begin
  n := 2;
  while true do
    begin
      writeln(n);
      n := 2 * n
    end
end.
```

■ When you execute this program it will begin by printing a series of real numbers. Eventually, however, it will blow up and a bug message will appear.

The key word in the error message is "overflow," and the concept behind the message is that the mathematical routines in the Macintosh have been asked to exceed the range of numbers with which they can work.

■ Change the type declaration to

```
n : double;
```

and try the program again. It will take a bit longer, but the blow-up will happen eventually.

■ Finally, change the declaration to:

```
n : extended;
```

and try the program. It will begin to crank out real numbers. If you are patient enough, it too will eventually exceed the allowable maximum value, but you will have to wait quite awhile to observe the phenomenon.

A common situation that will produce a numeric overflow is the act of dividing by zero. Since the resulting of such a calculation is infinite in value, it cannot be represented by Pascal. Any attempt to divide by zero will produce an error.

One other catch with real numbers is that exact representations of real numbers sometimes cannot be found. This can cause problems. A frequently encountered bug in programming involves testing for an exact match with a real number.

```
program matchbug;
  var
    n : real;
    i : integer;
begin
  n := 1.0;
  for i := 1 to 5 do
    begin
```

```
      n := n * 3;
      writeln(n : 8 : 8);
    end;
  repeat
    writeln(n : 8 : 8);
    n := n / 3
  until n = 1.0
end.
```

■ Enter and execute the program. Notice the numbers printed by the program. The program functions first to count upward from 1.0 by powers of 3.0. It then counts down, dividing by 3.0 until N has a value of 1.0. The program works properly, without a hitch. Take careful note of the values that are printed.

■ Now change the 3.0 in the assignment statements to 3.3, like this:

```
n := n * 3.3;
```

and

```
n := n / 3.3;
```

■ When you execute the program this time, you will find that it fails to stop. You will have to choose Halt in the Pause menu to stop it. Since you probably didn't stop it while these values were visible, here are the first twelve values that were printed:

```
3.29999995
10.88999939
35.93699646
118.59298679
391.35388184
391.35388184
118.59208679
35.93699646
10.88999939
3.29999971
0.99999994
0.30303028
```

When counting down, the program does not produce a value that is exactly equivalent to 1.0. Therefore, the REPEAT loop never terminates. The reason this occurs is that 3.3 is a value that cannot be exactly represented by the computer. 3.3 is stored as something like 3.29999995. This is one reason real numbers are not recommended for financial operations; there will often be slight errors that are intolerable in financial calculations.

To terminate this REPEAT loop, a different condition should be used. For example, by changing the condition to "N <= 1.0," the UNTIL clause will terminate the REPEAT loop as desired.

INTEGER DATA TYPES

We have used integers more frequently than real numbers. They are a bit more convenient since they may be printed without field parameters, and many situations in Pascal require them. For example, counter variables in FOR loops cannot be Real.

Integers have two advantages over real numbers: they do not suffer from the creeping imprecision of real numbers, and they require a bit less computer memory to store. However, integers are severely limited in the values that they may represent.

Standard integers may have values between −32767 and 32767. Pascal provides a standard constant MAXINT, the value of which is the greatest possible integer. The value of MAXINT changes from one version of Pascal to the next. For MacPascal, the value of MAXINT is 32767. MAXINT is a constant identifier that is predefined by Pascal. It can appear anywhere an expression is expected, for example as the parameter in a WRITELN statement:

```
writeln(maxint);
```

When larger or smaller integers are needed, we can take advantage of the MacPascal type Longint. The Longint constant that corresponds to MAXINT is MAXLONGINT, which has a value of 2,147,483,647. Long integers may be within the range of ±MAXLONGINT.

As with the high-precision real types, long integers require more memory space for storage.

Integers calculations are always carried out after conversion of the data to long integers. This means that the results of an integer calculation can become too great to fit back into a standard integer variable. We observed this phenomenon earlier with real numbers.

ORDINAL TYPES: INTEGER, CHAR, AND BOOLEAN

Real, Integer, Char, and Boolean data are all scalar in that they may be arranged along a scale of values from least to greatest. Of these, the types Integer, Char, and Boolean are ordinal. Ordinal types are a subset of the scalar types that have the following characteristics:

- The values are evenly distributed along the scale.
- The values are discrete. Consecutive values that do not have another value between them can be identified. (There is no value between "A" and "B" or between 2 and 3.)
- Every value in the type has an ordered position within all values of that type.
- The standard function ORD may be used to determine the position of a value within the range of values. The position of an Integer is the value of the Integer itself. For other ordinal types, the first value in the type has an ordinal value of zero, the second a value of 1, and so on. (Many computer numbering schemes number the first item in a series as zero. This can cause some problems when we humans must interpret the numbers, but numbering from zero is a fact of life with computers.)
- Every value except the first in a type has a value that precedes it. The preceding value can be determined with the standard function PRED.
- Every value except the last in a type has a value that succeeds it. The succeeding value can be determined with the standard function SUCC.

The values of the type Char are determined by the character set used by the host computer. Therefore, these values can differ quite widely among versions of Pascal, both in terms of the characters themselves and of the order in which the characters are ranged. A chart of Macintosh characters is

included in Appendix A. Most of these characters may be typed from the keyboard. Those that cannot may be generated through the standard function CHR. For example, CHR(65) will output the character A, which has the ordinal position of 65.

The type Boolean has only two values, TRUE and FALSE, but all of the characters of ordinal data apply.

- They are ordered: FALSE is less than TRUE.
- The ORD function applies: ORD(FALSE) is 0 and ORD(TRUE) is 1.
- FALSE has a successor value: SUCC(FALSE) is TRUE.
- TRUE has a predecessor value: PRED(TRUE) is FALSE.

The type Boolean leads us very nicely into user-defined types, which are more formally called *enumerated* types. By enumerated we mean that all of the values within the type can be listed. The values of the type Boolean are FALSE and TRUE. Let's go on to see what other sorts of enumerated types are possible.

ENUMERATED TYPES

An interesting and useful feature of Pascal is that we need not be satisfied with the data types that are provided as standard. If we want a type to represent the values of playing cards, of days of the week, or of colors we can easily define a new data type.

Types are usually declared in the TYPE definition block, which fits into programs ahead of the VAR block. If we wanted a type to represent colors, the TYPE block could look like this:

```
type
  color = (red, orange, yellow, green, blue, violet);
```

Once a new type is declared, variables may be created of that type. Assuming that the above TYPE block appears in the program, we could have the declaration:

```
var
  tint : color;
```

These declarations would permit us to use this assignment statement in a program:

```
tint := yellow;
```

Notice that YELLOW is not a string; it does not appear in quotes. Nor is YELLOW a variable that represents a value. YELLOW is itself a value.

Since a type must exist before the VAR block can assign it to a variable, the TYPE block must always precede the VAR block.

Enumerated types are ordinal. This means that the standard functions ORD, PRED, and SUCC may be applied to COLOR like this:

ORD(YELLOW) is 2
PRED(BLUE) is GREEN
SUCC(ORANGE) is YELLOW

In fact, enumerated types may be used in many situations where we have used integers.

■ We can now use a variable of type COLOR as a loop counter. Try this program.

```
program colors;
  type
    color = (red, orange, yellow, green, blue, violet);
  var
    shade : color;
begin
  showtext;
  writeln('The colors of the rainbow are:');
  for shade := red to violet do
    writeln(shade)
end.
```

I should point out that this program will not work in all versions of Pascal. Normally, Pascal does not permit us to input or output values in user-defined types, which severely limits what we can do with them. MacPascal imposes no such limitation. Therefore, we can use WRITELN to print values of the type COLOR, and we can modify the program to read in one of the loop limit values from the keyboard.

■ Modify the program like this:

```
program colors;
  type
    color = (red, orange, yellow, green, blue, violet);
  var
    shade, last : color;                          {new variable}
begin
  showtext;
  write('Type a color:');                         {new statement}
  readln(last);                                   {new statement}
  writeln('The colors of the rainbow are:');
  for shade := red to last do            {change VIOLET to LAST}
    writeln(shade)
end.
```

■ When you execute the program, be sure to type in one of the colors that were defined in the TYPE block.

There is a shorthand way to define types by including the value list in the VAR block. Using this technique, the COLORS program declaration section would look like this:

```
var
  shade, last : (red, orange, yellow, green, blue, violet);
```

The TYPE block would not be included in this situation. There are times when this approach will

not work, since a type identifier is not created in the process. For example, if we wished to define a local variable in a procedure, we would have to include the entire value list everywhere we wished to establish the type. It is much easier to define a type and assign an identifier to it at the beginning of the program. It is also, generally speaking, clearer when the functioning of the program must be interpreted.

Also, variables that are typed in this way cannot be used as parameters for functions or procedures. When parameters are declared in the function or procedure heading, they must be assigned a type. This type cannot be an enumerated list. It must be a predeclared type or a type declared in a TYPE block.

Here are some other types that might be useful:

```
day_of_week= (mon, tue, wed, thur, fri, sat, sun)

cardsuit = (club, diamond, heart, spade)
```

It might be tempting to try to represent card values like this:

```
facevalue = (2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king, ace)
```

Unfortunately, integers are already members of an ordinal type, and they cannot appear in user-defined types. If you need a type like this, you might spell out all of the card values:

```
facevalue = (two, three, four, five, six, seven, eight, nine, ten,
             jack, queen, king, ace);
```

Of course, you would be required to adhere to MacPascal practice and place this entire declaration on one screen line. This may limit the number of values that you can place into an enumerated type.

A given value may appear in only one type declaration. A TYPE block cannot contain both of the following declarations:

```
type
  color = (red, orange, yellow, green, blue, violet, black, white);
  primaries = (red, yellow, blue);
```

■ Add the PRIMARIES type declaration to the COLORS program and run the program. What error message is produced?

If we must have this PRIMARIES type, we can probably get what we want by declaring a new type that is a subrange of COLOR.

SUBRANGES OF TYPES

A *subrange type* is a subset of the values of some ordinal type that has already been defined. Here are a few examples, as they would appear in a TYPE block:

```
type
  positive_integers = 0..maxint;
  test_scores = 1..100;
  face_cards = jack..king;
```

The values in the subrange are not contained in parentheses, and they are separated by exactly two periods. The parent type must already have been defined before a subrange may be declared.

The first two examples rely on the type Integer, which is a built-in type and therefore already defined. The last subrange assumes that something like the FACEVALUE type defined earlier has been declared earlier in the TYPE block.

Subranges may also be declared in the VAR block. For example:

```
var
  testscores : 1..100;
  facecards : jack..king;
```

The subrange selected must be contiguous. Thus, we cannot solve our problem with the primary colors through a subrange unless the COLOR type declaration is modified. If we do not need to keep the values within COLOR in rainbow order, then this approach will work:

```
type
  color = (red, yellow, blue, orange, green, violet);
  primaries = red..blue;
```

We should note that the declaration of a subrange does not affect the ordinal position of the values within the subrange. The ordinal position of a value is always determined by its position in the original type, not in the subrange. Here is a program that demonstrates this fact:

```
program ordtest;
  type
    letters = (a, b, c, d, e, f, g, h);
    def = d..f;
  var
    sub : def;
begin
  showtext;
  sub := d;
  writeln(ord(sub));
  writeln(ord('d'))
end.
```

■ When you execute the program, note that the ORD value of SUB is 3, even though D is the first value in the subrange defined by DEF. Because D is in the fourth position in the type declaration of LETTERS, it has an ordinality of 3.

Just to drive home a point, note that the ordinal value d is not the same as the ordinal value of the character 'd'. The first is a value of the type LETTERS. The second is a value of the type Char. Do not confuse enumerated types with the strings or characters that they resemble.

Subranges are not really essential. We can get along quite well with the original type. As with many tools in Pascal, however, they are used to clarify the operation of the program. If we are writing a gradebook program, the declaration type SCORE, which is limited to values from 1 to 100, will announce right at the beginning of the program just what values we expect variables of that type to be assigned.

Examples of programs containing enumerated types and subranges will appear in future chapters.

YOUR PASCAL VOCABULARY

Your Pascal vocabulary now includes the following words:

Reserved Words

PROGRAM	BEGIN	END	VAR
DO	PROCEDURE	FUNCTION	CONST
TYPE			

Statement Types

Assignment (:=)	Compound		
FOR..TO	FOR..DOWNTO	WHILE	REPEAT..UNTIL
IF..THEN	IF..THEN..ELSE	CASE	

Data Types

BOOLEAN	CHAR	INTEGER	REAL
STRING	**DOUBLE**	**EXTENDED**	**LONGINT**

Procedures

READ	READLN		
WRITE	WRITELN		
NOTE			
FRAMERECT	PAINTRECT	FRAMEOVAL	PAINTOVAL
INVERTRECT		INVERTOVAL	
MOVETO	LINETO	LINE	
PENPAT	PENSIZE	PENMODE	
BUTTON	GETMOUSE		

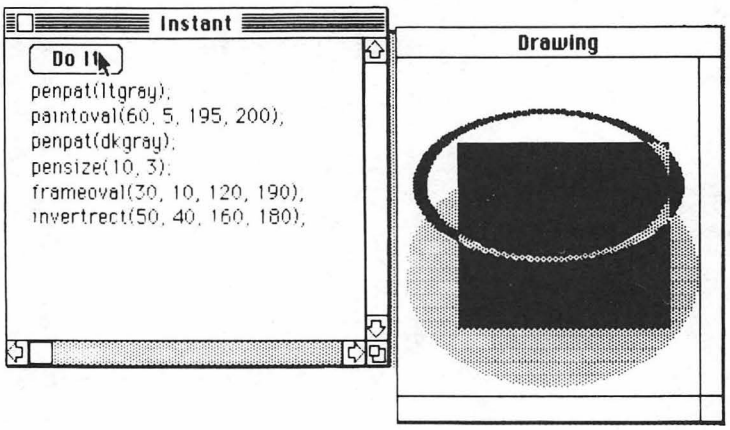
Operations

+	-	*	/
DIV	MOD		
>	>=	<	<=
<>	=		
NOT	AND	OR	IN

Functions

ROUND	TRUNC	SIN	COS
RANDOM	**ORD**	**SUCC**	**PRED**

Chapter 9



Structured Types: Arrays

In Chapter 7, we wrote a program DICECOUNT, which counted the frequency with which each of the six sides of a die appeared in a series of throws. Examine that program. Then ask yourself how you would write a program that kept track of the values produced by summing the throws of two dice. Would it be efficient to use eleven variables to keep track of each of the possible throws? What if we wished to cast three dice? This would require sixteen variables and sixteen cases in the CASE statement! If we cannot find a technique that is more powerful than that used in the first DICECOUNT program, any program to throw more than one or two dice would grow to huge proportions.

Many programming situations call on us to store large amounts of data. A really useful gradebook program might be called on to remember all of the grades for an entire class. Must we create a separate variable for each grade of each student? We would be a day just typing the VAR section.

As you might suspect, I am leading up to a more efficient way of manipulating large quantities of related variables. Using the new *type Array,* we will write a two-dice program that is considerably more compact than the first DICECOUNT program.

TOPICS COVERED IN THIS CHAPTER

- Single-dimension arrays
- Indexing arrays with the values of expressions
- Drawing bar charts
- Creating new pen patterns
- Multi-dimension arrays
- Developing a gradebook program

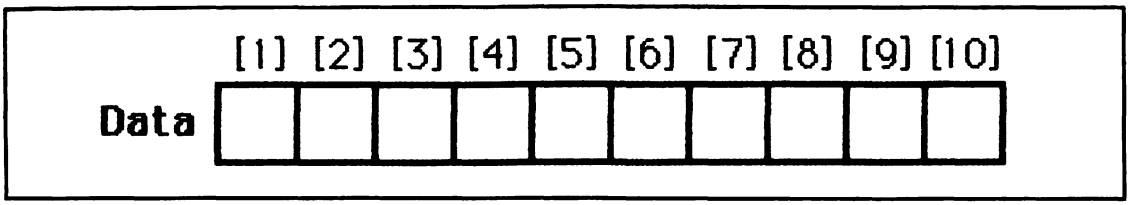


Fig. 9-1. A pictorial representation of an array.

SINGLE-DIMENSION ARRAYS

Recall that a variable is a location in memory used to store a particular type of data. We refer to the data stored in the variable by a name. Arrays allow us to refer to multiple locations using the same identifier.

If a variable may be likened to a box into which we put data, an array is a series of connected boxes. This situation is depicted in Fig. 9-1. It shows an array called DATA that has ten variable locations associated with it.

Every box is identified by the same name. However, each box has a number that makes it unique. DATA[5] refers to the fifth box in the array, for example. We can put this to work very easily in a simple program. Here is an absurdly simple one to serve as a starter:

```
program arraydemo;
  var
    data : array [1..10] of integer;
begin
  showtext;
  data[3] := 5;
  data[7] := 99;
  writeln(data[3]);
  writeln(data[7])
end.
```

The array variable is declared in the VAR block. This declaration says that DATA will be an array variable, that the array will have ten elements numbered 1 through 10, and that the elements will be of type Integer.

An array identifier can never appear without an array index, which is enclosed in square brackets. Together the identifier and the index point to a single element of the array.

The power of arrays arises because the indexing value may be derived from the value of an expression. Here is a program that first calculates the squares of the first ten integers. It then prints out those values in a table:

```
program squares;
  var
    count: integer;
    squares : array [1..10] of integer;
begin
  for count := 1 to 10 do
    squares[count] := count * count;
```

```
  for count := 10 downto 1 do
    writeln('The square of ', count, ' is ', squares[count])
end.
```

In the first assignment statement, COUNT serves both to index the array and as the basis for calculating its own square. If COUNT has a value of 5, then

```
squares[count] := count * count;
```

is equivalent to

```
squares[5] := 5 * 5;
```

As you work with arrays, you may find them very confusing since a single array identifier refers to a collection of values. The trick is always to think of a specific index value when you consider arrays. If you fully understand what is happening to one member of the array, you can usually generalize that knowledge to understand what is happening to all members of the array. This is a much easier mental leap than attempting to move directly to a general understanding of the array and then to reduce your understanding to the specifics of single elements.

If you find a particular array treatment confusing, substitute values for the variable names. After one or more such substitutions, the function of the array in the program should become clear.

Using a technique similar to that used in the previous program, the dice throwing program introduced in Chapter 7 can be considerably simplified.

```
program twodice;
  var
    i, temp : integer;
    roll : array [2..12] of integer;

  function rand (lowlimit, toplimit : integer) : integer;
    begin
      rand := random mod (1 + toplimit - lowlimit) + lowlimit
    end;

begin
  showtext;                          {1. Zero the array elements}
  for i := 2 to 12 do
    roll[i] := 0;

  for i := 1 to 1000 do              {2. Roll dice}
  begin
    temp := rand(1, 6) + rand(1, 6);
    roll[temp] := roll[temp] + 1;
  end;

  for i := 2 to 12 do          {3. Print results}
    writeln(i : 2, ' was thrown ', roll[i] : 3, ' times.')
end.
```

Notice that this version is more compact than the version in Chapter 7, even though the new one is responsible for throwing two dice. The main program is divided into three parts:

1. The array elements are prepared for counting by setting all values to zero.
2. A second loop is used to count dice throws. Each iteration, two calls to RAND are summed to determine the total dice score. The appropriate array element is then incremented.
3. A final loop prints out the results. In this loop, i is used in the WRITELN statement in two ways: to print the values of the dice throws, and to index the array when the totals of the throws are printed.

By building the program around arrays, it has become very easy to modify it for various numbers of dice.

■ Try to modify the program to throw three dice during each iteration. What changes are required?

In fact, the program may be designed so that only two values require modification, no matter how many dice we wish the program to throw at once. Let's make the necessary modifications.

■ Here is the heading of the program, along with a CONST block:

```
program multidice;
  const
    dice = 2;
    maxroll = 12;
    throws = 1000;
```

By placing these crucial values in the CONST block, it becomes very easy to modify the functioning of the program. To change the number of dice being thrown at one time, change the value of DICE to the number of dice and the value of MAXROLL to six times that number. THROWS simply reflects the number of times the dice will be thrown.

With these constants established, it seems fairly natural to use a subrange type to represent the range of possible dice total values.

■ After the CONST block, let's add a TYPE block and modify the VAR block as shown:

```
type
  rollrange = dice..maxroll;
var
  i, j : integer;
  temp : 0..maxroll;
  roll : array[rollrange] of 0..throws;
```

The range of ROLLRANGE will be determined by the values of DICE and MAXROLL. In turn, ROLLRANGE controls the range of the indexes of the array ROLL.

The index of an array is a subrange of some ordinal type. Therefore, it is possible to use a predeclared subrange to declare the index dimensions of an array. Since ROLLRANGE is defined as a subrange from 2..12, it may be used in the array type declaration as shown. Using the present constants, the declaration in the VAR block is equivalent to:

```
roll : array[2..12] of 0.. 1000;
```

To use these new declarations, the statement part of the program must be modified fairly extensively.

■ Change your program as follows, carefully checking your work:

```
begin
  showtext;
  for i := dice to maxroll do    {1. Zero the array elements}
    roll[i] := 0;

  for i := 1 to throws do        { Change 1000 to throws }
  begin
    temp := 0;
    for j := 1 to dice do        {2. Loop once for each die. Each}
      temp := temp + rand(1, 6);       {pass, add RAND to TEMP.}
    roll[temp] := roll[temp] + 1;
  end;

  for i := dice to maxroll do        {3. Print results}
  writeln(i : 2, ' was thrown ', roll[i] : 3, ' times.')
end.
```

Of course, your program must contain the function RAND if it is to work properly.

Notice how the values of the constants are used to control each of the loops. A small number of changes in the CONST block affect several places in the program.

As before, step 1 is to set all array values to zero. The integers used to determine the range of the FOR loop in the first version of the program have been changed to declared constants.

The most important single modification is the addition of a loop to throw the dice. Since we do not know how many dice will be thrown, we can no longer simply call RAND a fixed number of times. In the new version, a loop is executed once for each die that is to be thrown. During each interaction, a single call is made to RAND, and the result is added to TEMP. When this loop is complete, the sum in TEMP is used to increment the appropriate array variable. This is a common technique when loops are used to find the sums of a series of numbers.

GRAPHIC DISPLAY OF THE DICE PROGRAM RESULTS

Often the most effective way to display program data is graphically. The data in the DICE programs fall into a pattern that is not particularly obvious when presented in the form of text, but a bar graph will illustrate the pattern quite readily. Therefore, as a final touch, we will add a routine to draw a horizontal bar chart. In keeping with the spirit of easy modification, we will design the chart routines so that they can handle a reasonably large range of dice values.

Figure 9-2 shows a bar chart such as the one we want the program to draw. Before we can begin the drawing process, however, we must devise methods to determine the widths and lengths of the bars.

The width of the bars should be easy to calculate. We need to divide the height of the Drawing window, which is 200 units, into as many spaces as we require bars. We will require one bar for each element in the ROLL array. Since the range is from DICE to MAXROLL, we might try to calculate the number of elements by simple subtraction. The width of a single bar would then be determined like this:

```
barwidth := 200 div (maxroll - dice);
```

However, this would be wrong, despite its intuitive correctness. Using the present value of the constants, 18 – 3 would determine that there were 15 array elements. However, by counting the elements it may be seen that there are 16.

This is another instance of the Off-By-One-Error, one of the sneakiest bugs in programming. This bug is a problem for several reasons. In situations such as this one, it runs counter to our intuitions. If asked how many integers fall in the range of 13 to 87, we would quite naturally answer "74" since this is the difference between 13 and 87. We would, however, be "off by one." The effects of the bug are small, and the cause is often hard to detect. However, problems like our present one are encountered often, and the best defense is awareness.

The solution is to add one to the difference resulting from the subtraction:

```
barwidth := 200 div (1 + maxroll - dice);
```

This statement will correctly determine the number of array elements and divide that result into 200 to determine how wide each bar should be.

The other thing we must determine is how long the bars should be made. The values stored in ROLL can vary quite a bit. In order to make the most of the display space we have available, we would like to draw the bars as long as possible. We do not wish to have the tallest bar merely 15 units long. And we cannot draw a bar that is 300 units long. So we must devise a method of adjusting the heights of the bars depending on the data stored in the arrays.

Here is a sample of some values that might be stored in ROLL when three dice have been thrown one thousand times:

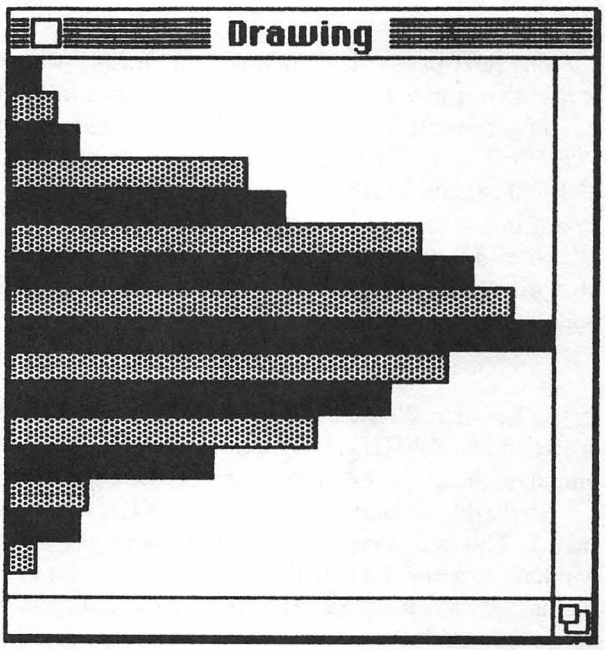


Fig. 9-2. A bar chart for the dice throwing program.

```
roll[3] = 4          roll[11] = 116
roll[4] = 14         roll[12] = 134
roll[5] = 29         roll[13] = 104
roll[6] = 45         roll[14] = 84
roll[7] = 64         roll[15] = 40
roll[8] = 92         roll[16] = 26
roll[9] = 123        roll[17] = 17
roll[10] = 111       roll[18] = 3
```

The first thing we must do is determine which is the greatest value stored in the array. This is a common enough task in programming, made quite easy by the use of arrays. The plan is simple: our program will examine each element in turn. If the value of an element is found to be the largest value encountered so far, that value is stored and compared to the succeeding elements. By always remembering the largest value from a comparison, the largest value in the array will be retained at the end of the examination.

Using a loop, this is more easily done than described:

```
j := 0;
for i := dice to maxroll do
  if roll[i] > j then
    j := roll[i];
```

J starts out with a value of zero. On each pass through the loop, J is compared to ROLL[I]. If the value of the array element is greater than the value of J, then J is assigned the value in the array element.

The first time through, I has the value of 3. Using the array values listed a few paragraphs back, ROLL[3] has the value of 4. Since J equals 0, the array value is greater than J, and J is assigned the value of 4.

The next pass, J is compared to ROLL[4], which has a value of 14. The value of J is 3, which is less than the value of the array element, and the value of J is updated to 14.

This continues until ROLL[13] is reached. At this point, J has the value of 134, the value of ROLL[12]. This is greater than the value of ROLL[13], which is 104. From this point on, the value of J will remain unchanged. At the end of the loop, J will have the value of 134, the largest value stored in the array.

Next, J is used to calculate a correction factor that will control the lengths of the bars. We will be storing the maximum bar length in the constant MAXWIDTH, and the correction factor is calculated quite simply as follows:

```
correction:= maxwidth / j;
```

If J is less than MAXWIDTH, then the correction factor will be greater than 1. For example, if J is 45 and MAXWIDTH is 200, the correction factor will be 4.44. When the bar dimensions are determined, multiplying 4.44 times 45 will cause the corresponding bar to be drawn with a length of 200.

Similarly, if J is greater than MAXWIDTH, then CORRECTION will have a fractional value less than 1. This will serve to reduce the length of a bar to 200. If J has a value of 323, CORRECTION will have a value of 0.61919. When this bar is drawn, multiplying 323 by 0.61919 will result in a bar that is 200 units long. Since the correction factor could take on fractional values, CORRECTION must be a real variable.

We are now in a position to determine all of the dimensions for a bar. Let's draw the first bar. A variable TOP will be used to keep track of the position of the top edge for each rectangle. The first bar represents the data stored in ROLL[3], the value of which is 4.

First we need the bar width. 200 DIV 16 is 12, and this is stored in BARWIDTH.

Next, we must know the length correction factor. We have previously determined that the largest value in the array is 134. The calculations then are:

```
correction := maxwidth / 134
```

which is to say that

```
correction := 200 /134.
```

Since we now know the correction factor to be 1.4954, we can determine the length of the bar for ROLL[3] to be:

```
round(correction * roll[3])
```

or

```
round(1.4954 * 4)
```

This bar, then, will be drawn with a horizontal length of 6. We can now define the parameters for the rectangle procedure that will draw this bar:

```
top = 0

left = 0

bottom = top + barwidth = 0 + 12 = 12

right = left + round(1.4954 * 4) = 0 + 6 = 6
```

The first bar will be drawn with the following parameters:

```
framerect(0, 0, 12, 6)
```

For the second bar, which represents a value of 14, the dimensions are:

```
top = 0 + barwidth = 0 + 12 = 12

left = 0

bottom = top + barwidth = 12 + 12 = 24

right = left + round(1.4954 * 14) = 0 + 21 = 21
```

And the FRAMERECT statement is:

```
framerect(12, 0, 24, 21)
```

The time has come to pull things together. Here is the complete program:

```
program multidice_graph;
  const
    dice = 3;
    maxroll = 18;
    throws = 1000;
    maxwidth = 200;
    left = 0;
  type
    rollrange = dice..maxroll;
  var
    i, j, barwidth, bottom, top, right : integer;
    temp : 0..maxroll;
    roll : array[rollrange] of 0..throws;
    correction : real;

  function rand (lowlimit, toplimit : integer) : integer;
  begin
    rand := random mod (1 + toplimit - lowlimit) + lowlimit
  end;

  begin
  showtext;
  for i := dice to maxroll do
    roll[i] := 0;

  for i := 1 to throws do
  begin
    temp := 0;
    for j := 1 to dice do
      temp := temp + rand(1, 6);
    roll[temp] := roll[temp] + 1;
  end;

  j := 0;
  for i := dice to maxroll do
  begin
    if roll[i] > j then                    {find greatest value}
      j := roll[i];
    writeln(i : 2, ' was thrown ', roll[i] : 3, ' times.')
  end;

  correction := maxwidth / j;
  barwidth := 200 div (1 + maxroll - dice);
```

```
  top := 0;
  showdrawing;

  for i := dice to maxroll do                    {draw the bars}
  begin
    bottom := top + barwidth;
    right := left + round(correction * roll[i]);
    framerect(top, left, bottom, right);
    top := bottom;
  end
end.
```

As you can see, once the basic calculations are carried out, the statements to draw the bars are pretty straightforward. Let's look at the last FOR loop. At the beginning of each iteration, BOTTOM is calculated by adding BARWIDTH to TOP. LEFT never changes, so we have quite easily taken care of two more sides of the rectangle. The right side is determined by the statement:

```
right := left + round(correction * roll[i]);
```

From there, it is a simple matter to arrange the dimensional parameters in the FRAMERECT parameter list.

The third FOR loop now serves two purposes. It is used to produce the written report of the array values, but it is also used to determine the greatest value in the array. Be alert in your programming for opportunities to make loops perform multiple duties.

Notice the extensive use of constants in the program. Later, we will learn to control the size of the drawing window. By including parameters that control the size of the bars, it will be very easy to modify the program to draw larger or smaller bars as window sizes change.

Also, in the chapter about strings we will discuss methods of labeling graphics. To do this, we will have to shorten the bars to allow room for the text.

One last thing. If you would like to dress up the chart a bit more, insert these statements just before the FRAMERECT statement:

```
if odd(i) then
  penpat(dkgray)
else
  penpat(ltgray);
paintrect(top, left, bottom, right);
penpat(black);
```

The IF..THEN..ELSE statement alternates the colors of the bars. The built-in function ODD outputs TRUE if the parameter evaluates as odd. This simple approach is an effective way to make a graph more interesting.

CHANGING THE PEN PATTERN

MacPascal uses a rather large number of built-in types when working with graphics. These types need not be defined by the user. Drawing patterns use a built-in Macintosh Pascal data type to represent the pattern data. The type is named Pattern, and is defined as follows:

```
type
  pattern = array [0..7] of 0.. 255;
```

Before embarking on a detailed explanation of the use of this type, let's run a demonstration program.

```
program penpattern;
  var
    pat : pattern;
begin
  showdrawing;
  pat[0] := 10;
  pat[1] := 4;
  pat[2] := 4;
  pat[3] := 85;
  pat[4] := 160;
  pat[5] := 64;
  pat[6] := 64;
  pat[7] := 85;
  penpat(pat);
  paintoval(50, 25, 125, 180);
end.
```

■ Try the program out to see how it works. Then we will do a little explaining.

All MacPascal patterns are based on an 8 by 8 grid. The one in Fig. 9-3 was used as the basis of the pattern in the program you just drew. Patterns you design must be arranged so that they will

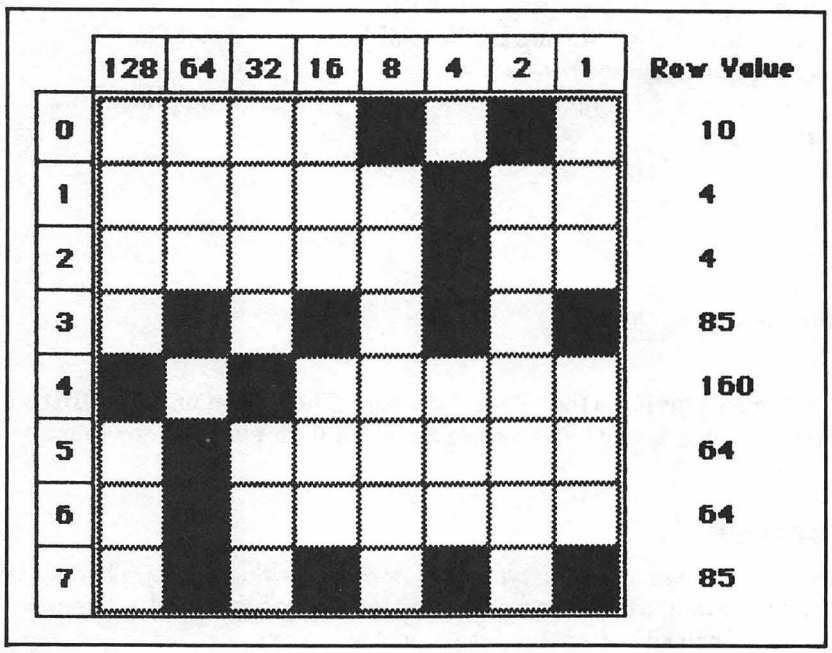


Fig. 9-3. A Macintosh pen pattern.

break up into 8 by 8 dot blocks that neatly fit together. Once a pattern is developed, it is necessary to represent the dots in numeric form so that they may be manipulated by Pascal. The process is not difficult, but it can be tedious.

Each row in the pattern will be represented by one of the eight array elements in the pattern array. So we need eight numbers for the eight rows.

Notice in Fig. 9-3 that each column is associated with a number: 1, 2, 4, 8, 16, 32, 64, or 128. To determine which number should be used to represent a given row, simply add together all of the numbers of the columns that contain dots in that row. If you add all of the possible values, you will find that the sum is 255. This explains the use of a subrange in the type definition: possible values all fall within the range 0.255.

In the first row of our example, dots are found in the 8 and in the 2 columns, so that row is represented by the number 10. Calculations for the other rows are also shown.

To establish a pattern, we must first create an array variable of the type Pattern. This, of course, is done in the VAR section of the program. Then it is a simple matter to assign the eight calculated values to the eight array elements. From then on, we can use the new pattern exactly as we have used the built-in patterns such as WHITE, LTGRAY, and DKGRAY. Examine the program to determine how the row numbers were stored into the array and how the array PAT was used to establish a new pen pattern.

This method of numerically representing a pattern is called a *bit-map*. Perhaps you recognized that the numbers associated with the columns were all powers of two. In fact, each row in the design can be represented as a base-two or *binary* number. The decimal equivalent of the binary number is used in assigning values to the pattern array elements. Here are the binary and decimal numbers used to define the pattern we are working with:

Binary Number	Decimal Equivalent
00001010	10
00000100	4
00000100	4
01010101	85
10100000	160
01000000	64
01000000	64
01010101	85

Binary digits are called *bits*. A bit can have only two values: on or off, usually represented as 1 or 0. This makes binary numbers ideal for representing dot patterns. We simply use a 1 to represent a dot and a 0 to represent the absence of a dot. The resulting numbers are then converted to decimal form for use in Pascal. When you add together the numbers for the columns, you are performing the conversion. As you can see, this can be done quite well without understanding binary numbers or bit maps. I just wanted to add a little explanation for the curious reader.

It's time to make your own pattern. If you are stuck for ideas, look at some of the patterns in the Macintosh Control Panel or in MacPaint. In MacPaint you can use the FatBits option to enlarge a pattern for examination.

MULTI-DIMENSION ARRAYS

The arrays we have used so far are called *single-dimension* arrays. Any value in the array may

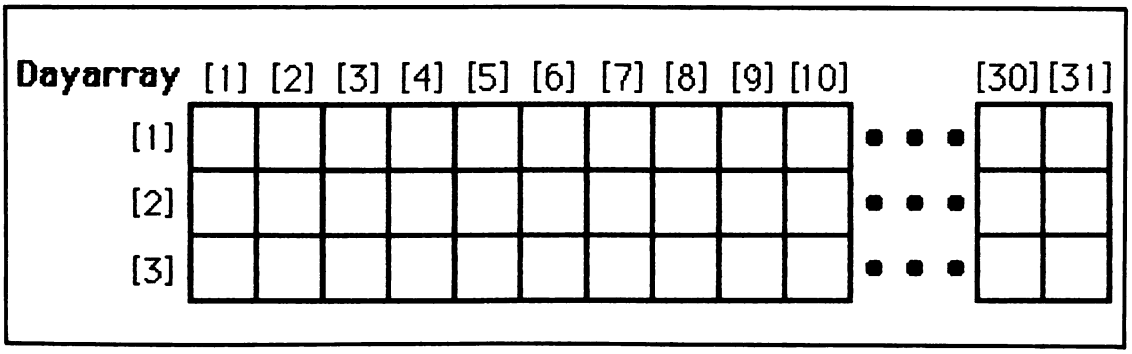


Fig. 9-4. A pictorial representation of a multi-dimension array.

be accessed through a single index. This is fine, provided we do not wish to subdivide the data within the array.

If we were writing an appointment book, we would probably wish to represent dates in a chart such as the one in Fig. 9-4. Notice that to uniquely identify a box in this diagram, we must specify two things, the number of the month and the number of the day in the month. In a program, this could be accomplished by using a multi-dimension array.

Let's assume that we wish each entry in the array to be a string that can hold a note for that day. So that we do not exceed the memory capacity of the Mac, we will design the program to keep track of only three months. Since there are 3 months and at most 31 days per month, a multi-dimension grade array could be declared like this:

```
var
  dayarray : array[1..3] of array [1..31] of string;
```

This declaration states that every element of the ARRAY[1..3] is itself an ARRAY[1..3] OF STRING. The ARRAY[1..31] corresponds to the columns in Fig. 9-3, while the ARRAY[1..3] corresponds to the rows. To fully specify a box, we must state the column and the row. For example, the box in the ninth column of the third row would be addressed as:

```
dayarray[3,9]
```

Pascal permits a shorthand form of this array declaration. The following declarations are functionally equivalent:

```
dayarray : array[1..3] of array [1..31] of string;

dayarray : array[1..3,1..31] of string;
```

We shall generally use the second form in our discussions.

The order of the arrays in the variable declaration is very important. The declaration could have been stated like this:

```
var
  dayarray : array[1..31,1..3] of string;
```

However, this would require that the column be addressed before the row. Now to address column 3 of row 9 we would use the following form:

```
dayarray[9,3]
```

The order in which the array dimensions are specified is not overwhelmingly important, since we can do everything with one form that we can do with the other. As with so many programming choices in Pascal, this one is best made with the goal of promoting clarity of the program.

In place of a numeric index, we could substitute an enumerated type. Since we are working with months, we might have declared this type:

```
type
  month = (jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec);
```

The array variable declaration would then be:

```
dayarray : array[1..31, jan..mar] of string[40];
```

The third day for month 3 would now be addressed as DAYARRAY 3], MAR. The use of an enumerated type often makes the workings of the array much clearer; multidimension arrays have a way of getting abstract and difficult to grasp. Enumerated types often make the dimensions of the array more explicit than do numeric indexes.

A GRADEBOOK PROGRAM

During the rest of the chapter, we will be developing a gradebook program. Our goal is to produce a program that will keep track of three test scores as well as a final score for each student. The final score is the average of the three test scores. The program will print out a semester final report complete with all scores, final grades, and a class average.

It will be useful to outline the things the program must accomplish. Broadly speaking, there are two tasks: entering the scores and printing the grade report. Of course, each of these must be broken down into considerably more detail. Let's examine score entry first.

The major tasks in score entry are:

A. entering and storing the student's names,
B. entering and storing the student's test scores, and
C. calculating and storing each student's final score, which is the average of his or her test scores.

Since the same three tasks are being performed for each student, we should immediately begin thinking in terms of a loop. Of the three types of loops, which is most appropriate? An IF..THEN loop would be fine if the number of students was fixed. It is probably easier, however, to use a more flexible loop since students might be absent, or we might wish to use the same program for classes of different sizes.

So we would like to allow the user to continue grade entry until all students have been processed, at which time entry of a special value would signal that data entry could cease. Actually, we can reach this goal in either of two ways. Either we can enter students while students remain to be entered, or we can enter students until all students have been entered. Each approach has its difficult features, but either will work. In a fairly arbitrary decision—remember that WHILE and REPEAT loops can often be used in the same programming situation—we will try a WHILE loop.

We can now begin to map the statement part of the program, using comments to indicate unfinished

tasks. The data entry part of the program can be diagrammed like this:

```
WHILE students remain to be entered
    A. enter and store a student's name
    B. enter and store the student's scores
    C. calculate and store the student's final score
```

Notice that we can begin to state our tasks in Pascal-like terms without getting into the details. In fact, the outline we have begun to develop will eventually be filled out to become our program.

We can now address the subtasks one at a time. Two things must be accomplished in the first subtask:

```
A. enter and store a student's name
    1. read the student's name from the keyboard
    2. store the name
```

Actually, a READ or READLN can perform both tasks, so we really do not have to break A down to subtasks.

Since we are storing several students' names, and since we have decided to use a loop, an array seems to be the logical way to store the names. Let's call the array NAMES. Next we must determine how to index the array for data storage. We will establish an index variable, tentatively called STCOUNT, without worrying about the details of it just yet. The outline so far becomes:

```
initialize the index variable STCOUNT
WHILE students remain to be entered
    A. READ a name and store it in NAMES[STCOUNT]
    B. enter and store the student's scores
    C. calculate and store the student's final score
    increment STCOUNT
```

Since we have reached the point of forming task A into a close approximation of Pascal form, we have a good indication that analysis of that step is nearly complete. We can leave A for the moment.

Turning our attention to B, we notice that three test scores are being entered. We have two things to determine: how the data are to be entered, and how they will be stored.

For entry, a loop is again suggested. Any repetitious task should cause us to examine the possibility of using a loop. Since three test scores are being entered, the intriguing idea emerges of using an enumerated type as the loop counter. Suppose that we have a type with the values (TEST1, TEST2, TEST3). A FOR loop could easily be set up like this:

```
B. FOR TEST1 TO TEST3 DO READ a test score and store it
```

Again, circumstances suggest an array for storage of scores, and it seems reasonable to make it two-dimensional. One dimension would correspond to the students. Since we are already using STCOUNT to index the NAMES array, it seems like a good idea to use the same variable to index the students in the grade array. That way, a given value of STCOUNT would point to the name of a student in NAMES and to the same student's grades in the grade array.

The other dimension of the array will be used to store the test scores. Since an enumerated type may be used to index an array, it might be useful to use variables of the same type for both loop control and to index the array. In fact, we can use the FOR loop control variable to index the scores dimension of the array. As you will see, this is an effective technique and we will plan to include this type declaration:

```
GRADES = (TEST1, TEST2, TEST3)
```

To control the FOR loop, we need a variable of type GRADES. Let's call it GDCOUNT. The FOR loop has now begun to take shape, and B now becomes:

```
B. FOR GDCOUNT := TEST1 TO TEST3 DO
       READLN GRADEARRAY[STCOUNT,GDCOUNT]
```

For a given student, the scores will be found in GRADEARRAY[STCOUNT, TEST1] through GRADEARRAY[STCOUNT, TEST3].

A first approach to step C might be:

```
C. Average := (GRADEARRAY[STCOUNT, TEST1] +
               GRADEARRAY[STCOUNT, TEST2] +
               GRADEARRAY[STCOUNT, TEST3] ) DIV 3
```

This works, but frankly I am bothered by the wordiness and repetition, which would become even worse if we decided to adapt the program to store more grades per student in the future. Since the data is stored in an array, we can use a loop to accumulate the total of the test scores. This would give step C this form:

```
C. FOR GDCOUNT := TEST1 TO TEST3
   SUM := SUM + GRADEARRAY[STCOUNT, GDCOUNT]
```

Followed by a division to determine the average:

```
AVERAGE := SUM DIV 3
```

The virtue of this approach becomes evident when we realize that this FOR loop has the same structure as the loop in B. We can make the same loop serve multiple functions. B becomes:

```
B. Initialize SUM variable
   FOR GDCOUNT := TEST1 TO TEST3 DO
     BEGIN
      READLN GRADEARRAY[STCOUNT, GDCOUNT]
      SUM := SUM + GRADEARRAY[STCOUNT, GDCOUNT]
     END
```

The only thing left for C is to calculate and store the average score:

```
C. AVERAGE := SUM DIV 3
```

This loop will expand very easily. If we wish to add more tests, we simply expand the values in the type definition for GRADES.

Where is the final grade to be stored? Do we create a new array? It would be nice if we could store it in the same array as the other grades. We can do this since the final grades will be of the same type as the test scores. Therefore, we can just add one more column to the array. Lets change the type declaration of GRADES to:

```
GRADES = (TEST1, TEST2, TEST3, FINAL)
```

Without changing anything else, this will create a new place for the final score. C becomes simply:

```
C. GRADEARRAY[STCOUNT, GDCOUNT] := SUM DIV 3
```

The only remaining task is to increment the array index variable:

```
Increment STCOUNT
```

The complete outline now looks like this:

```
initialize the index variable STCOUNT
WHILE students remain to be entered
    A. READ a name and store it in NAMES[STCOUNT]
    B. Initialize SUM variable
      FOR GDCOUNT := TEST1 TO TEST3 DO
        BEGIN
         READLN GRADEARRAY[STCOUNT, GDCOUNT]
         SUM := SUM + GRADEARRAY[STCOUNT, GDCOUNT]
        END
    C. GRADEARRAY[STCOUNT, GDCOUNT] := SUM DIV 3
    Increment STCOUNT
```

Without too much trouble, we can map out a bare-bones Pascal program to perform these tasks. At this stage, we include comments that tie the program to the points in our outline.

```
begin
  { initialize index variable }
  stcount := 1;
  while { students remain }
    begin
      {A. read and store a name }
      readln(name[stcount]);
      {B. read and store the test grades }
        sum := 0;
        for gdcount := test1 to test3 do
          begin
            readln(gradearray[stcount, gdcount]);
```

```
            sum := sum + gradearray[stcount, gdcount]
          end;
        {C. store the final score }
        gradearray[stcount, final] := sum div 3
        { increment the index variable }
        stcount := stcount + 1
      end
    until
```

Two things are missing in the program. No prompts are provided to inform the user of what he or she is expected to type. These can be easily added and will be in the final program.

More important, however, is the fact that the condition for ending the WHILE loop has not been established. How can the program recognize that the last student has been entered? If the user simply presses the Return key without typing anything else, READLN will accept an *empty* string. WHILE can easily examine NAME[STCOUNT] for an empty string and stop when one is found.

The first problem we must solve is this: the name is not entered until after the beginning of the WHILE statement. Since no value has been assigned to NAME[1], we cannot use that variable in the conditional part of the WHILE statement. The solution to this is to move the READLN so that it is before the WHILE:

```
readln(name[stcount]);
while (name[stcount] <> '') do
```

If the user types a name (or any text) then NAME[STCOUNT] will not be an empty string and the statement part of the WHILE loop will execute.

But now, the name is being entered outside of the WHILE loop. When the loop repeats, the READLN will not be repeated. We must include a second READLN inside the loop:

```
begin
  showtext;
  stcount := 1;
  readln(name[stcount]);
  while (name[stcount] <> '') do
    begin
      sum := 0;
      for gdcount := test1 to test3 do
        begin
          readln(gradearray[stcount, gdcount]);
          sum := sum + gradearray[stcount, gdcount]
        end;
      gradearray[stcount, final] := sum div 3;
      stcount := stcount + 1;
      readln(name[stcount])
    end;
```

Examine this portion of the program closely. Keep in mind that the secret to understanding arrays is to substitute values for the index variables. The first time through the loop, the value of STCOUNT is 1. Substitute 1 wherever STCOUNT appears. Then substitute TEST1, the first value

of GDCOUNT, wherever that variable appears. You will find that the array references become much easier to understand when you do this. For one thing, you are now looking at only one element of the array at a time instead of trying to understand the entire array.

As a final resort, get some paper and mark it out in columns and rows. Label the rows and columns and begin to "walk" through the REPEAT loop. What happens to the values of the index variables? Write every value down as it occurs, crossing out the old values. Do anything you can to make the values of the variables concrete, instead of abstract things inside the computer. Imagine that you are entering values in the READLN statements. Which elements of the array will these values be stored in? How does your chart look at the end of each loop?

In the future, loops and multi-dimensioned arrays will be old friends to you, and you will have no trouble forming mental pictures of how they work. For now, however, they are abstract and slippery. The only way to become comfortable is to interact with arrays and loops as much as possible. Eventually, your brain will begin to sort things out.

An important point: STCOUNT is incremented before the second READLN statement. If the rest of the loop was working with NAME[1], the next name should be stored in NAME[2]. If STCOUNT is not incremented before the READLN, then the new name will replace the name in NAME[1].

Even worse, if STCOUNT is incremented after the read, then NAME[2] does not have a value assigned. When the WHILE statement tests to see if NAME[STCOUNT] is empty, the value of the variable is undefined, meaning that there is no telling what the WHILE will do. Later, when the complete program is at hand, I will ask you to switch the positions of these two statements to observe the results. It is common when working with arrays to introduce bugs through improper indexing of the array.

Here is the entire GRADEBOOK program. Examine the type definitions. Most of these grow out of the preceding discussion. A new item is the constant CLASS_SIZE. The dimensions of arrays must, of course, be stated in the declaration part of the program. The constant CLASS_SIZE is used to control the dimension of the type STUDENTS and thus the sizes of the arrays GRADEARRAY and NAMES. We can easily allow for more students by increasing only the value of CLASS_SIZE.

Also new are the WRITE statements, used to provide user prompts before the READLN statements. Of particular interest is the use of ORD in this line:

```
write('Grade ', ord(gdcount) + 1 : 2, ': ');
```

ORD is used to print a number beside each request for a grade, as in "Grade 1".

The new constant CLASS_SIZE is used to control the dimensions of several arrays and subranges. By placing this value in a constant, it is very easy to change the number of students that the program will handle. Use of the constant also makes the purposes of the array dimensions very clear.

There are now two possible ways to end the WHILE loop. Since the constant CLASS_SIZE places an upper limit on the number of students that will be accepted, the loop must terminate if STCOUNT ever exceeds CLASS_SIZE.

■ Before reading about the second part of the program, enter and run the present version. You need not enter grades for thirty students. Three or four will be quite adequate. When you run the program, you will want to enlarge the Text window to the full size of the screen. Otherwise, the grade report will not be formatted into columns. (In Chapter 11 you will learn how to control the sizes of the Text and Drawing windows with program statements.)

```
program gradebook;
  const
    class_size = 30;
```

```
  type
    grades = (test1, test2, test3, final);
    numericscore = 1..100;
    students = 1..class_size;
  var
    stcount : students;
    gdcount : grades;
    gradearray : array[students, grades] of numericscore;
    sum : integer;
    name : array[students] of string[20];

  { Enter the student names and grades. Calculate final grades}

begin
  showtext;
  stcount := 1;
  write('Name of student ', stcount : 1, ': ');
  readln(name[stcount]);
  while (name[stcount] <> '') and (stcount <= class_size) do
    begin
      sum := 0;
      for gdcount := test1 to test3 do
        begin
          write('Grade ', ord(gdcount) + 1 : 2, ': ');
          readln(gradearray[stcount, gdcount]);
          sum := sum + gradearray[stcount, gdcount]
        end;
      gradearray[stcount, final] := sum div 3;
      stcount := stcount + 1;
      write('Name of student ', stcount : 1, ': ');
      readln(name[stcount])
    end;

  {  Print out the scores and final grades  }

  write('Name          ' : 20);          { print the report header }
  writeln('Score 1' : 9, 'Score 2' : 9, 'Score 3' : 9, 'Final' : 9);
  stcount := 1;
  sum := 0;                              { print student grades }
  while (name[stcount] <> '') and (stcount <= class_size) do
    begin
      write(name[stcount] : 20);
```

```
        for gdcount := test1 to final do
          write(gradearray[stcount, gdcount] : 9);
        writeln;
        sum := sum + gradearray[stcount, final];
        stcount := stcount + 1
      end;
    writeln;
    write('Average of final scores:' : 47);
    writeln(round(sum / (stcount - 1)) : 9)
end.
```

The last section of the program is used to print out a final grade report. The majority of this section is composed of WRITE and WRITELN statements. The heart of this section is a WHILE loop, which performs several functions:

- It prints a student name.
- It uses a FOR loop to print out the scores for the student.
- It adds each student's final score to the running total being accumulated in SUM.

After all student names and grades have been printed, the WHILE loop terminates. All that remains is to print out the final average. The two components of the class average are the total of the grades, which was accumulated in SUM, and the number of students, which is conveniently provided by STCOUNT.

Notice how similar the two parts of the program are. Each uses a main loop to process the students one-by-one, and each uses a nested loop to process the grades. Once a pattern is established for manipulating an array, that pattern will often be applied several places in a program.

In each case, the grade loop is nested inside the student loop. This is because the program was primarily concerned with all of the grades for a single student. Had the program been concerned with all of the grades for TEST1, then all of the grades for TEST2, and so forth, the student loop might have been nested inside of the grades loop.

A minor point in formatting: notice how the use of field width parameters in the WRITE and WRITELN statements made it easy to line up the columns of the heading with the columns of data. The result is a very neat report.

YOUR PASCAL VOCABULARY

You now know the following Pascal words. Words that were new in this chapter are printed in bold face type.

Reserved Words

PROGRAM	BEGIN	END	VAR
DO	PROCEDURE	FUNCTION	CONST
TYPE	**OF**		

Statement Types

Assignment (:=)	Compound		
FOR..TO	FOR..DOWNTO	WHILE	REPEAT..UNTIL
IF..THEN	IF..THEN..ELSE	CASE	

Data Types

BOOLEAN	CHAR	INTEGER	REAL
STRING	DOUBLE	EXTENDED	LONGINT
ARRAY			

Procedures

READ	READLN		
WRITE	WRITELN		
NOTE			
FRAMERECT	PAINTRECT	FRAMEOVAL	PAINTOVAL
INVERTRECT		INVERTOVAL	
MOVETO	LINETO	LINE	
PENPAT	PENSIZE	PENMODE	
BUTTON	GETMOUSE		

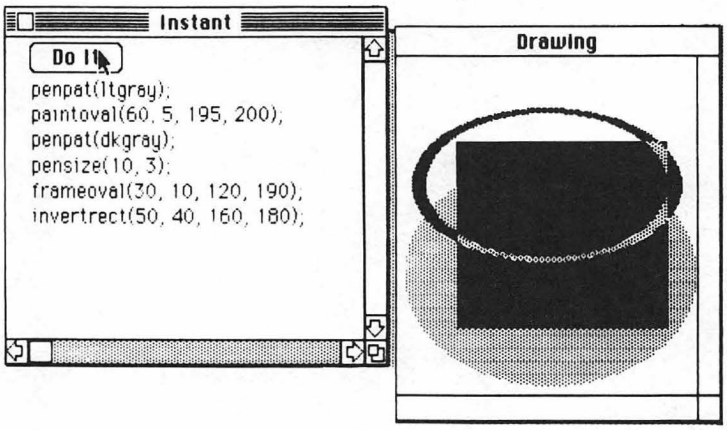
Operations

+	-	*	/
DIV	MOD		
>	>=	<	<=
<>	=		
NOT	AND	OR	IN

Functions

ROUND	TRUNC	SIN	COS
RANDOM	ORD	SUCC	PRED
ODD			

Chapter 10



Structured Types: Records

Often it is desirable to be able to manipulate data of different types as a unit. For example, if we were writing an inventory program, we might wish to keep track of the following information about every item in the inventory:

Item	Data Type
Item description	String
Stock number	String
Quantity on hand	Integer
Unit cost	Real

If all of these items were of the same data type, we could easily represent them in a two-dimensional array. However, three data types are represented. A single array won't work since all items in an array must be of the same type. We could solve the problem by using three arrays, of course, one of each type. This is not a particularly neat or efficient solution, however. And it can become positively unworkable if the number of data items grows, as is likely to happen with a real-life inventory program.

The problem of storing related but different-typed data arises quite often. An information file in a company personnel office might be required to keep track of each employee's name, age, sex, marital status, address, and so forth. A checkbook balancing program might be required to manipulate check number, payee, and amount, and to keep track of whether or not the check has cleared. The number of arrays required could grow to be quite large.

The solution to this dilemma is provided by a new data type: the record.

TOPICS COVERED IN THIS CHAPTER

- Defining record types
- Assigning values to the components of a record variable
- Using the WITH statement
- Identifying components of nested record definitions
- Points and rectangles in MacPascal graphics
- Changing the sizes of the Drawing and Text windows under program control
- Determining whether or not a point falls inside of a rectangle
- Using the mouse to select program options
- Variant records
- Some variant record types used in MacPascal graphics

AN INTRODUCTION TO PASCAL RECORDS

Here is the declaration of a record type that may be used to represent the inventory data just discussed:

```
type
  item_data = record
    description, number : string;
    on_hand : integer;
    price : real
  end;
```

This declaration appears in the TYPE block and creates a new type, named ITEM_DATA. Within this type are four *components* or *fields*, corresponding to the four data items we wish to keep track of. Each of these fields has a type and an identifier. Notice that Pascal allows more than one identifier to be associated with a type on the same line. Both DESCRIPTION and NUMBER are of type String.

To use this type, a variable must be declared. For example:

```
var
  item : item_data;
```

To store data in ITEM, a program must refer both to the variable and to the field. For example, to store the price of an item, this assignment statement might be used:

```
item.price := 1.98;
```

The variable name is followed by a period and the field identifier. Together they refer to a single record component in the record variable. DESCRIPTION, NUMBER, PRICE, or ON_HAND are not usable on their own. They must be prefaced by the variable name ITEM to be meaningful.

When the variable identifier alone is used, it refers to the entire record variable. If ITEM1 and ITEM2 are record variables of the type ITEM_DATA, it is perfectly legitimate to use this assignment statement:

```
item1 := item2;
```

The effect of this statement is to assign the value of each field in ITEM2 to the corresponding field

in ITEM1. This allows complete record variables to be passed as parameters to procedures and functions.

■ Here is a program that illustrates the use of the record type ITEM_DATA. Enter the program and run it.

```
program inv1;
  type
    item_data = record
      description, number : string;
      on_hand : integer;
      price : real
    end;
  var
    item : item_data;
begin
  showtext;
  item.description := 'shovel';
  item.number := 'hg123';
  item.on_hand := 12;
  item.price := 14.95;
  writeln;
  writeln('Item: ', item.number, '    ', item.description);
  writeln('Total inventory: $', item.on_hand * item.price : 6 : 2);
end.
```

After the price is assigned to ITEM.PRICE, the WRITELN statements will report the total value of the shovels in inventory. In this simple program, each record identifier functions simply as an independent variable. In later programs, we will see that the items in a given record may also be manipulated as groups.

Let's examine the record type declaration. The first line is

```
item_data = record
```

This line names the type and begins a record declaration. This line has the same form as other declarations in the TYPE block:

identifier = type definition

Since RECORD does not mark the end of the declaration, no punctuation appears at the end of this line.

Subsequent lines contain the record field definitions, followed by the word END.

```
item_data = record
  description, number : string;
  on_hand : integer;
  price : real
end;
```

These field declarations take the same form as variable declarations:

identifier : *type*;

The field declarations are separated by semicolons, and the declaration is completed by the word END.

Record variables may also be created in the VAR block when we do not require a type definition. We could have easily created the variable ITEM like this:

```
var
  item : record
    description, number : string;
    on_hand : integer;
    price : real
  end;
```

If ITEM is defined in this way, no TYPE block is needed. When only one variable of a given record type is required, this is an acceptable way of declaring it, but there are limitations. Record variables declared in this way may not be used to pass data to procedures or functions. A record definition cannot appear within a procedure or function parameter list. Therefore, the variables declared in parameter lists must be defined in terms of a predefined type.

Record identifiers can get pretty long and repetitious, but there is a way to simplify things: the WITH statement. Here is the statement part of the inventory program, rewritten to use WITH:

```
begin
  showtext;
  with item do
    begin
      description := 'shovel';
      number := 'hg123';
      on_hand := 12;
      price := 14.95;
      writeln;
      writeln('Item: ', number, '    ', description);
      writeln('Total inventory: $', on_hand * price : 6 : 2)
    end
end.
```

■ Substitute this for the main program in INV1. Run the modified program and confirm that both versions produce the same results.

Immediately following WITH, a record identifier is specified. Within the compound statement following WITH, only the field identifier is required to address a field in a record variable. The identifier following WITH specifies which record variable the program should manipulate. This form of the program is functionally equivalent to the earlier version. WITH does not let us do anything new; it just makes things more convenient in certain circumstances.

A record field may be declared to be of any data type. This means that a record field may itself be a record! It might be convenient to keep all of the information about a part supplier in a record. Taken by itself, the record might be defined as:

```
type
  supplier_info= record
    name : string;
    street : string;
    city : string
  end;
```

or, more simply as

```
type
  supplier_info= record
    name, street, city : string
  end;
```

Once this type declaration is made, it is very easy to include it in the definition of ITEM__DATA:

```
item_data = record
  description, number : string;
  on_hand : integer;
  price : real;
  supplier : supplier_info;
end;
```

It can be a bit complicated to address a field within a nested record. Our sample program declared the variable ITEM to be of the type ITEM__DATA. To address the PRICE field in ITEM, we must supply two bits of information: the variable name ITEM, and the field name.

To address a field in a nested record, we must supply an identifier for each level of the nesting. We must name the variable, the field in the primary record, and the field in the nested record. Therefore, to assign a value to the NAME field, the assignment statement would look like this:

```
item.supplier.name := "Jones & Company";
```

■ Here is the sample program, modified to incorporate the record definition. Enter and run the new version.

```
program inv2;
  type
    supplier_info= record
      name, street, city : string;
    end;
    item_data = record
      description, number : string;
      on_hand : integer;
      price : real;
      supplier : supplier_info
    end;
```

```
  var
    item : item_data;
begin
  showtext;
  item.description := 'shovel';
  item.number := 'hg123';
  item.on_hand := 12;
  item.price := 14.95;
  item.supplier.name := 'Jones & Company';
  item.supplier.street := '690 Main Street';
  item.supplier.city := 'Smallville, Ohio';
  writeln;
  writeln('Item: ', item.number, '    ', item.description);
  writeln('Total inventory: $', item.on_hand * item.price : 6 : 2);
  writeln('Supplier: ' :12, item.supplier.name);
  writeln(' ' : 12, item.supplier.street);
  writeln(' ' : 12, item.supplier.city)
end.
```

Let's look at another example of a record definition. To write a card game, we would need a convenient method of representing a player's hand. For starters, we might declare a type CARD:

```
card = record
  suit : (club, diamond, heart, spade);
  value : (two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, ace)
end;
```

Each of the fields is defined as an enumerated type. Once this record type exists, it is a simple matter to store the player's cards in an array. All we need is the following variable declaration:

```
hand : array[1..10] of card
```

To specify a card in a hand, we must supply the array index, as well as the field identifiers. The suit and value of the first card in the hand could be assigned like this:

```
hand[1].suit := diamond
hand[1].value := jack
```

In this way, any card in the player's hand can be identified just by changing the array index.

Notice that record fields are identified in a consistent manner, regardless of the variable types involved. The variable and field identifiers are simply strung together, separated by periods. This can get very complicated, particularly when records contain records or arrays, but the method of building up the field identifiers is always the same. One other thing to remember is that in any such field identifier, the first entry and only the first entry will be a variable name. All other entries will be field identifiers, established in record definitions.

Here are some type definitions that will further illustrate the versatility of Pascal records:

```
book = record
  title, author, publisher : string;
  year : integer;
  price : real
end;

student_data = record
  name : string;
  class : integer;
  major : string;
  grade_average : real
end;

date = record
  month : (jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec);
  day_of_month : 1..31;
  year : integer
end;
```

RECORDS AND MACPASCAL GRAPHICS

Records are used extensively with MacPascal graphics, and some special record types are predefined. Several of these record definitions are quite complicated and beyond the scope of this book. The ones I will introduce, however, will greatly expand the scope of your graphics activities.

Most graphics operations must be able to address points on the drawing screen. A point is defined as a record with vertical and horizontal components:

```
point = record
  v: integer;
  h : integer;
end;
```

It is not necessary for us to declare this type. POINT is defined for us by MacPascal.

If P1 is a variable of type POINT, then, two different values must be assigned to it before it may be used. If P1 is to represent a point with a horizontal location of 50 and a vertical location of 75, these assignment statements might be used:

```
p1.h := 50;
p1.v := 75
```

Points as this one are used by MacPascal in drawing several types of shapes. Let's examine their use in drawing rectangles.

MacPascal also defines a type named RECT, which is used in most procedures that are based on rectangles. This record definition establishes a rectangle on the basis of two diagonally opposite points:

```
rect = record
  topleft : point;
  botright : point
end;
```

■ Here is a program that uses this definition of a rectangle. Enter and run it.

```
program rec;
  var
    r : rect;
begin
  r.topleft.h := 10;
  r.topleft.v := 15;
  r.botright.h := 100;
  r.botright.v := 125;
  framerect(r)
end.
```

The parameter list of FRAMERECT is obviously constructed differently from what we are accustomed to. Always before, we have included four values describing the top, left, bottom, and right sides. Before the end of the chapter, we will discover that there are multiple ways of passing parameters to some rectangle procedures.

In this program, only one parameter appears in the parameter field for FRAMERECT. Since R is defined to be of type RECT, however, it represents not one but four values. These values were assigned in the four assignment statements, which we will now examine.

R is defined as a record that consists of two points: TOPLEFT and BOTRIGHT. These points are themselves defined as records. To specify one value in R, we must provide three pieces of information:

- the variable name
- the point (TOPLEFT or BOTRIGHT)
- and the dimension (V or H)

Thus, the vertical dimension of the top-left point is specified as:

```
r.topleft.v
```

The field identifiers TOPLEFT, BOTRIGHT, V, and H are not declared in the program since they are part of the preset definitions for the RECT and POINT types. Do not confuse these field identifiers with variables.

Now that we understand the type RECT, we can do something you may have wished we could do since we first started with graphics. We can change the size and location of the Drawing and Text windows under program control. The MacPascal screen is organized in points. The top left corner is point 0,0. The screen is 512 points wide and 342 points high, so the bottom right point is designated as 511,341. At the top, about 40 lines are occupied by the Mac's menu labels. Using this information, we can set the location and size of the Pascal output windows.

Because we will wish to do this often, let's write a procedure that will enlarge the Drawing win-

dow to fill the Macintosh display. To do this we will call on the procedure SETDRAWINGRECT, which sets the size of the Drawing window. This procedure expects a single parameter, which must be of the type RECT. Here is a procedure that can often be used to start out a graphics program. It sets the window size and selects the Drawing window for display.

```
procedure fulldraw;
  var
  r : rect;
begin
  with r do
    begin
      topleft.v := 40;
      topleft.h := 1;
      botright.v := 340;
      botright.h := 510;
      setdrawingrect(r);
      showdrawing;
    end;
end;
```

SETDRAWINGRECT operates just like a rectangle procedure. You can do the same thing for the Text window with the procedure SETTEXTRECT, which functions in exactly the same way.

■ Locate a graphics program you are fond of and install FULLDRAW in it. Don't forget to call FULLDRAW in the beginning of your program. When you run the program, notice where the drawing takes place. Although the window has changed size and shape, the top left corner is still 0,0 as far as the graphics procedures are concerned. This news may disturb you since I announced that the top left corner of the MacPascal screen is also designated as 0,0.

It is important to realize that several coordinate systems are used in MacPascal. The location and size of the Drawing window are defined in terms of the coordinates on the Macintosh screen. However, to make drawing convenient, the Drawing window has its own coordinate system, which is used by all graphics procedures. This *local* coordinate system is used only by the Drawing window. If the local coordinates were not available, we would have to modify the parameters of a procedure every time the Drawing window was moved. Normally, the upper left corner of the Drawing window is always 0,0 no matter how we move the window around or change its shape.

Notice that the dimensions of the Drawing window include the borders at the top, right, and bottom sides. If you wish to eliminate a border, simply expand that side so the border will be forced off of the screen.

IMPROVING A DRAWING PROGRAM

I would like to go back to an earlier program to correct some deficiencies. In Chapter 5 we created a program named RECTANGLES that drew rectangles under control of the mouse. Using the FULLDRAW procedure, we will be able to draw using the entire Macintosh screen. In the earlier version, we had to draw all rectangles starting from their top left corners. Using our new knowledge about point and rectangle types, we will be able to start at either the top left or the bottom right cor-

ner, moving the mouse in any direction to draw a rectangle. Finally, we will provide a box which can be clicked to quit the program.

MacPascal rectangle procedures expect the first point in the parameter list to be above and left of the second point. If this is not the case, no rectangle is drawn. We could have gotten around this in the earlier version of the program by using a few IF . . THEN statements and swapping points as required. However, a much simpler solution is available.

The procedure PT2RECT will examine two points and return the rectangle that the points describe. To demonstrate, we can modify the rectangle program from earlier in this chapter. The first step is to modify the points so that TOPLEFT is below and to the right of BOTRIGHT.

■ Change the constants in the assignment statements of REC, and try the program again.

```
program rec;
  var
    r : rect;
begin
  r.topleft.h := 100;
  r.topleft.v := 125;
  r.botright.h := 10;
  r.botright.v := 15;
  framerect(r)
end.
```

Since I warned you, you are probably not surprised that nothing was drawn. Now, through some simple modifications, we can make the program draw any rectangle, regardless of the relative positions of the points.

PT2RECT accepts two points and returns a rectangle. To utilize the procedure, we will create a second variable of type RECT to receive the returned value. Here is the modified program:

```
program rec;
  var
    r, r1 : rect;                    {add R1}
begin
  r.topleft.h := 100;
  r.topleft.v := 125;
  r.botright.h := 10;
  r.botright.v := 15;
  pt2rect(r.topleft, r.botright, r1); {new line}
  framerect(r1)                      {change R to R1}
end.
```

The assignment statements are unchanged. After the values have been assigned to the various fields in R, the points may be used as parameters of PT2RECT. An interesting thing is happening here. We see that by changing the definition, we can refer to either the entire rectangle in R, to the points that specify the rectangle, or to the coordinates that specify the points. Here are the various possibilities:

Rectangle	Point	Coordinates
R	R.TOPLEFT	R.TOPLEFT.V
		R.TOPLEFT.H
	R.BOTRIGHT	R.BOTRIGHT.V
		R.BOTRIGHT.H

This ability to work with record components at multiple levels is one of the keys to the versatility of records.

In the program, coordinates for the desired rectangle are first assigned to the coordinate components of the record R. Then, the program uses the point fields of R as parameters for PT2RECT, which requires three parameters. The first two parameters are the points that will determine the dimensions of the rectangle. The third is a variable of type RECT. After execution of PT2RECT, the variable will hold the appropriate values for drawing a rectangle specified by the points.

Try the program, editing the values in the assignment statements to draw several different rectangles. Notice that any values may be assigned to the four points, regardless of their positions in the Drawing window.

In addition to improving rectangle drawing, we would like to provide a box that may be clicked to quit the program. Whenever you use the Macintosh, you are asked to click boxes to make choices. The modification about to be introduced will illustrate something of how these boxes are created and used in programs. It depends on the built-in function PTINRECT, a Boolean function that determines whether or not a point falls within a rectangle. Using this function, we can easily tell if the mouse button has been clicked inside of a rectangle.

■ Here is the modified program. Enter it and try it out.

```
program Draw_Rectangles;
  var
    x, y : integer;
    p1, p2 : point;
    r1, quitrec : rect;

  procedure fulldraw;
    var
      r : rect;
  begin
    with r do
      begin
        topleft.v := 0;
        topleft.h := 0;
        botright.v := 358;
        botright.h := 530;
        showdrawing;
        setdrawingrect(r);
      end;
  end;      {of fulldraw}
```

```
  procedure setup;
  begin
    with quitrec do
    begin
      topleft.v := 30;
      topleft.h := 440;
      botright.v := 60;
      botright.h := 500;
    end;
    pensize(3, 3);
    framerect(quitrec);
    pensize(1, 1)
end;      {of setup}

begin
  fulldraw;
  setup;
  repeat
    while not button do
    ;
    getmouse(x, y);
    p1.v := y;
    p1.h := x;
    penmode(patxor);
    if not ptinrect(p1, quitrec) then
      begin
        repeat
          getmouse(x, y);
          p2.v := y;
          p2.h := x;
          pt2rect(p1, p2, r1);
          framerect(r1);
          framerect(r1);
        until not button;
        penmode(patcopy);
        framerect(r1)
      end;
  until ptinrect(p1, quitrec)
end.
```

FULLDRAW has been modified to expand the borders of the Drawing window completely off the screen. This presents a nice appearance to users of the program. If you write programs for use by others, you will not want them to be concerned with the details of Macintosh windows, so it is good policy to conceal them. You can reveal or manipulate programs through program statements,

making things automatic as far as the user is concerned. When the program has ended, you will have to reveal the Program window by choosing the appropriate name ("Untitled," or if you have saved the program, the name you specified when saving) in the Windows menu.

The new procedure SETUP was defined to remove clutter from the main program. Only statements directly concerned with the main action of the program remain in the statement part of the program.

The primary thing accomplished by SETUP is the assignment of values to the variable QUITREC. Following the assignment statements, a rectangle is drawn using QUITREC as the parameter.

The first new thing in the main program is the IF . . THEN statement surrounding the central REPEAT loop. The IF statement will permit the REPEAT statement to execute only if the rectangle specified by QUITREC was not clicked. The function PTINRECT, which requires a point and a rectangle as parameters, returns TRUE if the point falls within the rectangle. PTINRECT is used to determine if the rectangle was clicked.

P1 represents the point at which the mouse button was depressed to end the first WHILE loop. In this version of the program, the mouse coordinates from GETMOUSE are stored in the point P1. QUITREC represents the rectangle that was drawn by SETUP. The IF statement calls PTINRECT to determine if P1 is inside QUITREC. If the function returns FALSE, the IF statement executes the REPEAT loop to draw a rectangle.

The outermost REPEAT loop has a new terminating condition in the UNTIL clause. PTINRECT is called again to determine if the loop should terminate. If it is true that point P1 fell within QUITREC, then the loop ends. If you have used any Macintosh program, you have probably selected functions by clicking areas of the screen with the mouse. Now you know how this is accomplished in a program.

Any number of boxes could be placed on the screen, each with a different function. By storing the rectangle definitions for each of these boxes, we could determine if any of them was clicked. Using this knowledge, we could easily make a click in each box activate a different function.

VARIANT RECORDS

Records have yet one more feature that greatly amplifies their power. Returning to our hardware inventory, suppose that we have two sorts of items to contend with: screws and nails. To specify a screw size, we must record two items: its length in inches and an integer that describes its diameter. Nail sizes, however, require only a "penny" size, which is an integer. To efficiently handle both items in records, we would like to have two different record definitions.

For screws, this definition would be useful:

```
item_data = record
  description : string;
  number : string;
  on_hand : integer;
  price : real;
  length : real;
  size : integer;
end;
```

Screw lengths are usually represented in inches and fractions, for example "2 1/4". For convenience, we will use real numbers, but a working inventory program might use strings or enumerated types to represent the sizes.

On the other hand, for nails we might use this version of the record:

```
item_data = record
  description : string;
  number : string;
  on_hand : integer;
  price : real;
  size : integer;
end;
```

These definitions are not that different. They have four fields in common (DESCRIPTION, NUMBER ON—HAND, and PRICE). The other fields contain some differences, however. There is no diameter for nails at all. Also, length for screws is real, but SIZE for nails is of type integer.

Despite the differences listed, it would be convenient to store data for all fasteners in the same record type. This can be done by adding a *variant part* to the record. This is accomplished through the introduction of a CASE section. Since CASE statements require scalar types to select the cases, the new type FASTENERS must also be declared.

```
type
  fasteners = (screw, nail);

  item_data = record
    number : string;
    on_hand : integer;
    price : real;
    case description : fasteners of
      screw : (length : real; screwsize : integer);
      nail : (nailsize : integer);
  end;
```

The fields NUMBER,ON__HAND, and PRICE are defined as standard record fields. The format of the variant portion is considerably different, however.

The first line is:

```
case description : fasteners of
```

This line accomplishes two things. It begins the variant part of the record and states that the value of DESCRIPTION will be used to select the variants. In addition, it establishes DESCRIPTION as a field in the record, which has the type of FASTENERS. If ITEM is a variable of type ITEM__DATA, a program may assign a value to ITEM.DESCRIPTION. For example:

```
item.description := screw;
```

As with conventional CASE statements, the value of the case variable is used to select one of the available cases. In the variant record, the cases contain field definitions. You will find that the format for variant field definitions is the same as the format for value parameters in procedures. As in procedure parameter fields, only predefined types may be used to type the field identifiers.

The key concept behind variant records is that the value of the case variable determines which variant fields are active at a given time.

If ITEM.DESCRIPTION has the value of SCREW, then ITEM.LENGTH and ITEM.SCREWSIZE are available.

If ITEM.DESCRIPTION has the value of NAIL, then ITEM.NAILSIZE is available.

In a moment, we will look at a simple program that illustrates the use of a variant record. When you enter the program, do not be too shocked at the way MacPascal reformats the CASE portion of the record definition. With the Pascal version available while I was writing this book, the Case clause will be reformatted like this:

```
case description : fasteners of
  screw : (
    length : real;
    screwsize : integer
  );
  nail : (
    nailsize : integer
  );
```

If you compare this version to the one given earlier, you will find the contents are identical. The first version used a more conventional format, one which I prefer since it is more compact and better shows the structure of the statement. Recall that Pascal does not care how things are formatted; we normally format programs to make them easy to read and to understand, not because it makes a difference to Pascal.

In this book, I will always display the parentheses associated with a variant record on the same line as the field definition. However, you have no choice but to have your variant records reformatted by MacPascal, so be prepared. Generally speaking, if you have committed an error in your punctuation Pascal will inform you by displaying part of the statement in outline text.

■ Here is the program. Enter and run it.

```
program inv3;
  type
    fasteners = (screw, nail);
    item_data = record
      number : string;
      on_hand : integer;
      price : real;
      case description : fasteners of
        screw : (length : real; screwsize : integer);
        nail : (nailsize : integer);
    end;
  var
    item : item_data;
begin
  showtext;
  with item do
    begin
```

```
        description := screw;
        number := 'n12-2';
        on_hand := 50;
        price := 11.49;
        length := 2;
        screwsize := 10;
        writeln;
        write('Item: ', number, ' ' : 4, description, ' ' : 4);
        writeln(length : 4 : 2, screwsize);
        writeln('Total inventory: $', on_hand * price : 6 : 2)
      end       { of with }
end.
```

By now, the major features of a program such as this should be readily understandable to you. Take special note of the fields that were defined in the variant part of the record. Notice that they are used just as if they were normal record fields.

■ When you are satisfied that you understand the program, make a small change. Change the program so that DESCRIPTION is assigned the value of NAIL. Then run the program. What message is produced?

When the value of DESCRIPTION was changed, the fields SCREWSIZE and LENGTH became inactive. However, we did not alter the assignment statements that utilized them. The active field now is NAILSIZE. We may use it by removing the SCREWSIZE and LENGTH assignment statements and by changing the WITH statement as shown:

```
with item do
  begin
    description := nail;                    {change value}
    number := 'n12-2';
    on_hand := 50;
    price := 11.49;
    nailsize := 12;                         {new line}
    writeln;
    write('Item: ', number, ' ' : 4, description, ' ' : 4)
    writeln( nailsize);                     {change variables}
    writeln('Total inventory: $', on_hand * price : 6 : 2)
  end       { of with }
end.
```

There are a few more points to make about variant records:

- Even though they may reside in different variants, no two fields in the record may have the same field identifier. This is the reason the size fields were labeled as NAILSIZE and SCREWSIZE.
- The same END is used to terminate both the CASE portion of the record definition and the record definition itself. Therefore, the variant portion of the record must appear as the last part of the record definition.

- Only one variant section per record definition is permitted.

PREDEFINED VARIANT RECORD TYPES IN MACPASCAL

My primary reason for introducing variant records in this book is that they are used extensively in MacPascal graphics. Other than this, it will probably be some time before you require the power of variant records in your own programming.

We will illustrate MacPascal variant record types by returning to the type RECT. We have already seen that there are two ways to specify the size of a rectangle. The first method, introduced in the first chapter of the book, involved parameters for the top, left, bottom, and right side locations:

```
framerect(top, left, bottom, right)
```

However, if R is of type RECT, we can also call FRAMERECT like this:

```
framerect(r)
```

You already know that rectangles are defined using records. You should now begin to suspect that these are variant records, and indeed that is the case.

Here is the official MacPascal definition of the type RECT:

```
type
  rect = record
    case integer of
      0: (top : integer;
          left : integer;
          bottom : integer;
          right : integer);
      1: (topleft : point;
          botright : point);
    end;
```

Most of this definition adheres to your earlier format for a record variant. In case 0, four fields are declared: TOP, LEFT, BOTTOM, and RIGHT. Each of these is of type integer.

In case 1, two fields are declared: TOPLEFT and BOTRIGHT. These fields are of type POINT, which is, of course, also a record type. This variant, therefore, involves nested records.

If you examine the definition closely, you will notice that CASE is not followed by a case selection identifier. INTEGER is a type, not a field identifier. This contradicts our earlier observations that the value of a field identifier determines which variant fields are active.

How, then, do we determine which variant is active? In a sense, both are active as we can easily demonstrate.

■ Examine this program. Then enter and execute it:

```
program var_rec_demo;
  var
    r : rect;
begin
  with r do
```

```
    begin
      top := 50;
      left := 25;
      botright.h := 100;
      botright.v := 150;
      framerect(r);
    end
end.
```

The topleft corner of the rectangle was defined using R.TOP and R.LEFT. The bottom right corner, however, was defined using the point R.BOTRIGHT. Nothing special was done to switch from one representation to the other. Pascal did that for us.

But we can take things even further. Add these three statements after the call to FRAMERECT:

```
right := 190;
topleft.v := 10;
framerect(r)
```

A second rectangle was drawn in which two of the parameters were altered. This time, R.TOPLEFT.V was used to alter the location of the top of the rectangle, and R.RIGHT was used to change the location of the right side. Obviously, we have a great deal of freedom in defining rectangles.

There is yet another way to pass parameters to FRAMERECT. Before we knew about records, we included four values in the parameter list. Let's try it with this program.

- Remove the three lines you just added.
- Then change the FRAMERECT statement to this:

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

- When you try the program, everything will work perfectly.
- Try one last test. To see if we can use points in the parameters for FRAMERECT, change the procedure call to this:

```
framerect(topleft, botright);
```

We have two methods that work and one that doesn't. Let's try out the three forms again, using another procedure that uses a rectangle as a parameter.

- Remove the FRAMERECT statement, and substitute the following:

```
showdrawing;
setdrawingrect(r)
```

SETDRAWINGRECT is another procedure that requires a rectangle in its parameter list.

- Execute the program. The drawing window will assume the location and dimensions specified by the values in R. So far so good.
- Change the procedure call to:

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

- Try the program. Does it work? What messages does MacPascal produce?

■ Finally try the program with this parameter list:

```
setdrawingrect(topleft, botright);
```

Since points did not work for FRAMERECT, you probably did not expect them to work for SETDRAWINGRECT. But FRAMERECT accepted four side values, which were rejected by SETDRAWINGRECT.

If you examine your MacPascal manual, looking at the definitions for these procedures, you will find that they are very similar:

```
procedure framerect (r : rectangle);
procedure setdrawingrect (windowrect);
```

WINDOWRECT is defined as a value of type RECT, so the two parameter lists are essentially the same.

We already know that both procedures function properly when a parameter of type RECT is passed to them. We also know that they will not accept two points. So, we can assign values to the rectangle parameter using points, but we must use a variable of type RECT to pass the values to the procedure.

However, we have successfully used FRAMERECT and other rectangle programs, passing the procedures four parameters to specify the four sides. Why does this not work with SETDRAWINGRECT?

As a convenience, the developers of MacPascal permitted certain rectangle procedures to accept parameters of four side coordinates. If the rectangle programs would not accept these four parameters, we could not use constants in the parameter lists of the procedures. We could not have expressions in the parameter lists. Four assignment statements would be required for each call to a rectangle procedure, since four values would have to be assigned to the fields of the rectangle record variable. In short, our lives would be complicated. I would have had to teach you about records before you could have drawn a simple rectangle.

So, MacPascal allows us to cheat when using rectangle drawing procedures. In most other cases, however, if a procedure expects a variable of type RECT, that is exactly what must be provided.

YOUR PASCAL VOCABULARY

You now know the following Pascal words. New words are printed in bold face type.

Reserved Words

PROGRAM	BEGIN	END	VAR
DO	PROCEDURE	FUNCTION	CONST
TYPE	OF		

Statement Types

Assignment (:=)	Compound		
FOR..TO	FOR..DOWNTO	WHILE	REPEAT..UNTIL
IF..THEN	IF..THEN..ELSE	CASE	WITH

Data Types

BOOLEAN	CHAR	INTEGER	REAL
STRING	DOUBLE	EXTENDED	LONGINT
ARRAY	**RECORD**		

Graphics Data Types

RECT	POINT

Procedures

READ	READLN		
WRITE	WRITELN		
NOTE			
FRAMERECT	PAINTRECT	FRAMEOVAL	PAINTOVAL
INVERTRECT		INVERTOVAL	
MOVETO	LINETO	LINE	
PENPAT	PENSIZE	PENMODE	
BUTTON	GETMOUSE	**PTINRECT**	**PT2RECT**
SHOWTEXT	SHOWDRAWING	**SETTEXTRECT**	**SETDRAWINGRECT**

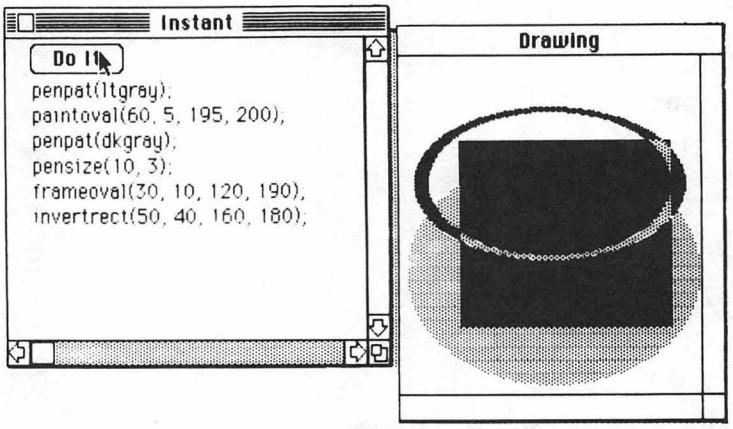
Opertions

+	-	*	/
DIV	MOD		
>	>=	<	<=
<>	=		
NOT	AND	OR	IN

Functions

ROUND	TRUNC	SIN	COS
RANDOM	ORD	SUCC	PRED

Chapter 11



Strings

Our dealings with strings have been quite superficial up to this point. However, strings are powerful tools, and there is much yet left to learn about them. In this chapter, we will explore various techniques for manipulating strings. Then we will discover how to display text in the Drawing window. This opens up the potential for displaying text in all of the Macintosh fonts and styles that you have probably admired.

TOPICS COVERED IN THIS CHAPTER

- The capacity attribute of a string variable
- The use of Boolean operators to compare strings
- Building strings with concatenation and insertion
- Taking strings apart
- Displaying and manipulating text in the Drawing window
- Formatting text printed in the Drawing window

THE CHARACTERISTICS OF STRINGS

Strings, as you know, are built up of characters. String constants are designated by enclosing the text in single quotes:

```
'This is a string.'
```

Every string has a length, which is simply the number of characters it holds. The length may be determined by the LENGTH function.

■ Run this example in the Instant window:

```
writeln(length('This is a string.'))
```

It will display 17, the number of characters in the string constant.

Strings are stored in variables of type String, of course. When a string variable is declared, Pascal determines the maximum length string that it may hold. If we do not inform Pascal differently, then the variable will be set up to hold 255 characters, the maximum size allowed by MacPascal.

Strings that long are rarely required, however, so this standard (the computer term for "standard" is *default*) length is often wasteful of memory. A string variable with a size of 255 takes up 255 characters of space in memory, even if only one character is stored in it.

Therefore, we would often like to create string variables of smaller capacity. This is done by adding a size parameter when the string is declared:

```
var
  name : string[20];
  city : string[15];
  state : string[2];
```

This VAR block creates three string variables of lengths 20, 15, and 2 respectively. This saves memory, a scarce resource in most computers and particularly in the 128K Macintosh.

Most of the relational operators Pascal provides may be used on strings much as they were on numbers and characters. Here is a program that demonstrates string comparisons:

```
program stringdemo;
  var
    string1 : string[10];
    string2 : string[20];
begin
  string1 := 'apple';
  string2 := 'applesauce';
  writeln(string1 < string2)
end.
```

■ Run the program. Then edit the program, substituting each of these operators: >, >=, <=, <>, and =. Run the program with each operator and observe the results.

■ Change the value of STRING2 to 'APPLE'. Then try the various comparison operators again.

You have discovered several things during these experiments:

- Two strings are equal when they contain exactly the same characters, in the same sequence.
- If a string follows another string alphabetically, it will be greater than the string it follows.
- The declared size of the string variable does not affect the comparison of the strings. Two strings may be identical even though their variables have different sizes.

MacPascal does not allow the string length parameter to appear in the parameter definition field of a procedure or a function. This procedure heading would not be permitted:

```
procedure xyz (s : string[40]);
```

If it is desirable to control the length of a string parameter, a type must be declared. For example, if this type declaration is in place:

```
shortstring = string[40];
```

then the procedure heading could be expressed like this:

```
procedure xyz (s : shortstring);
```

BUILDING STRINGS

When working with strings, you will constantly be needing to assemble two strings into one or to break large strings into small ones. MacPascal is rich in its ability to assemble and dissect strings.

Any number of strings may be combined into one string using the CONCAT function. The function's name is derived from the term *concatenation,* which describes the process of adding items to the ends of a series of things, like adding links to a chain. To use CONCAT, simply place the strings to be joined in the parameter list.

■ Run this example in the Instant window.

```
writeln(concat('app', 'les', 'auce')
```

It will display the string

```
applesauce
```

Notice that no spaces are included in the final string. If we wish to combine words into a sentence, one way to do it is to include space strings in the parameter list.

■ Run this statement:

```
writeln('apples ', 'and', ' oranges')
```

which will display the text

```
apples and oranges
```

If data of various types are to be combined into a single string, the STRINGOF function comes in handy. The parameter list for STRINGOF follows exactly the same rules as the parameter list for WRITELN. Any printable data type may appear.

■ For example, execute this:

```
writeln(stringof('6' : 1, ' apples cost $', 1.29 : 3 : 2))
```

which prints the string

```
6 apples cost $1.29
```

One last string-building operation is the insertion of one string into another. There are two ways to do this using the function INCLUDE and the procedure INSERT.

■ Try this statement, which uses INCLUDE:

```
writeln(include('lle', 'change', 4))
```

This statement will print the text

challenge

INCLUDE requires three parameters:

1. The string to be inserted
2. The string into which the insertion is to be made
3. An integer, which is used to determine where the first string will appear in the second string.

In this example, 'IIe' will be inserted in 'change' starting in the fourth character position. If the third parameter is greater than the length of the second string, the first string will simply be concatenated to the end of the second.

When the receiving string is to be stored in a variable, it is often more convenient to insert text with the INSERT procedure, which uses a variable parameter to directly manipulate the value of a variable. We can demonstrate string insertion easily with the STRINGDEMO program used earlier.

■ Modify the statement part of the STRINGDEMO program as shown:

```
program stringdemo;
  var
    string1 : string[10];
    string2 : string[20];
begin
  string2 := 'change';
  insert('lle', string2, 4);
  writeln(string2)
end.
```

The same thing could have been done using INCLUDE, but an assignment would have been required.

■ Substitute this statement for the INSERT statement in STRINGDEMO and try the program again:

```
string2 := include('lle', string2, 4);
```

The program will work just as it did when INSERT was used. The only difference is that INSERT was a little less wordy.

When assembling strings, the result may not exceed the size of the variable in which it is stored.

■ Change the size of STRING2 from 20 to 5 and try the last program again. How does Pascal react?

TAKING STRINGS APART

MacPascal provides an equally rich vocabulary for disassembling strings. COPY is a function that is used to extract portions of a string. Here is one example of an expression that uses COPY:

```
writeln(copy('abcdefg', 3, 2))
```

will print "cd". The parameters for COPY are:

1. The source string
2. The starting location for the copy

3. The number of characters to be copied

OMIT is a function that may be used to remove any part of a string. It requires three parameters.

■ Demonstrate its use by trying this example:

```
writeln(omit('challenge', 4, 3))
```

which prints

```
change
```

The first parameter is the string to be deleted from. The second is an integer pointing to the position of the first character to be deleted. The third is an integer telling how many characters should be removed.

These parameters are the same as those used by COPY. Both procedures start by specifying a substring in the original string. They differ in the way they manipulate the substring. COPY returns the substring itself. OMIT returns the original string after the substring has been removed. OMIT and COPY may, therefore, be considered as complementary functions. They will often be used together to accomplish a task.

Suppose we wished to remove everything after 'apple' from 'applesauce'. If we do not know how many characters are to be removed, the LENGTH function comes into play. LENGTH outputs the number of characters stored in any string.

■ Try this example:

```
writeln(omit('applesauce', 6, length('applesauce') - 6))
```

The expression LENGTH'('APPLESAUCE') −6 determines how many characters follow 'Apple', which in this case is 5.5 becomes the third parameter for OMIT, and the statement prints 'apple".

Now we would like to remove everything in 'applesauce' before 'sauce'. To do this, we need to determine the position of the 's' in the original string. For situations like this, Pascal provides the POS function.

■ Demonstrate its use by running this statement:

```
writeln(pos('sauce', 'applesauce'))
```

will print the value 6. The first parameter is the substring to be searched for, and the second is a string in which to search.

■ This may easily be used to create the length parameter for OMIT. Try this:

```
writeln(omit('applesauce', 1, pos('sauce', 'applesauce'))
```

In this example, 6 characters will be removed starting at position 1. What does Pascal print? Try it and see if you can name the error. It is our old friend the Off-By-One-Error again. POS returned the position of the "s" in "sauce", but we need to know the position of the character just before the "s." To fix the statement, we must subtract 1 from the output of POS.

■ Run this statement, which includes the correction:

```
writeln(omit('applesauce', 1, pos('sauce', 'applesauce') - 1)
```

The statement now prints the desired result, "sauce."

DELETE is a procedure that does about the same thing as the function OMIT. However, it uses a variable parameter to directly manipulate the value of the variable storing the original string. It would be informative for you to compare DELETE, OMIT, INSERT, and INCLUDE.

■ To demonstrate DELETE, we will again modify the program STRINGDEMO. Run this version.

```
program stringdemo;
  var
    string1 : string[10];
    string2 : string[20];
begin
  string2 := 'challenge';
  delete(string2, 4, 3);
  writeln(string2)
end.
```

Let's pull some of these different operations together into a program that prints the words in a string one-at-a-time. This exercise has a practical value, which we shall see later in the chapter.

To remove a word from the beginning of a string three things must be done:

1. Determine the position of the first space using the POS function.
2. Use COPY to extract all of the characters from the beginning of the string to the position of the period.
3. Remove the word from the original string.

■ A simple loop can use this process to print each word in a string individually. Enter and run this program.

```
program word_by_word;
  const
    space = ' ';
  var
    string1 : string[80];
    location : integer;
begin
  showtext;
  string1 := 'A rose is a rose is a rose. ';
  location := pos(space, string1);
  if location > 0 then
    repeat
      writeln(copy(string1, 1, location));
      delete(string1, 1, location);
      location := pos(space, string1)
    until location = 0;
  writeln(string1)
end.
```

POS returns 0 when the target string is not found. This is used by the program as the signal to end the REPEAT loop. However, when POS returns 0, STRING1 probably has one last word in it. A final WRITELN statement prints any of the string that remains.

The first time through the loop, POS outputs 2, which is assigned as the value of LOCATION. The WRITELN statement uses COPY to determine the characters of STRING1 from positions 1 through LOCATION. The copied string, the first word of STRING1, is "A". Finally, DELETE removes the first word by removing all characters through the first space from STRING1. The process is now ready to repeat.

Using a similar approach, we can reverse the words in a sentence:

```
program backwords;
  const
    space = ' ';
  var
    string1, string2 : string[80];
    location : integer;
begin
  showtext;
  string1 := 'One two buckle my shoe. ';
  string2 := '';
  location := pos(space, string1);
  if location > 0 then
    repeat
      insert(copy(string1, 1, location), string2, 1);
      delete(string1, 1, location);
      location := pos(space, string1)
    until location = 0;
  string2 := concat (string1, string2);
  writeln(string2)
end.
```

Unfortunately, there is no built-in procedure like POS to find the location of the *last* space in a string. Several approaches might be used to get around this. The one we will use stores the words from the string in a second string variable, inserting them into the second string in reverse order. This string is printed at the end of the program.

STRING2 is used to collect the words as they are removed from STRING1. These words are added to STRING2 by inserting them at the beginning. Since the words are being added to the beginning of STRING2, I could have used either INSERT or CONCAT in writing the program. I have used both in the program to illustrate the differences in the ways they are used.

As a final illustration, here is a program that reverses the order of the characters in a string.

```
program reverse;
  const
    space = ' ';
  var
    string1, string2 : string[80];
```

```
    location : integer;
begin
  showtext;
  string1 := 'One two buckle my shoe. ';
  string2 := '';
  for location := 1 to length(string1) do
    insert(copy(string1, location, 1), string2, 1);
  writeln(string1);
  writeln(string2)
end.
```

Here, all of the work is done within a simple FOR statement. In the last two programs, we removed the words from the source string after they had been copied. That got rid of the parts of the string we were finished with, letting us use POS to find the end of each word in the string.

In this program, no deletions are required. Since exactly one character will be copied from the string with each pass through the loop we do not have to examine the source string for spaces. We can just step through the string and take one character at a time. The FOR loop counter is used to determine the position at which the COPY is to be made. The length of the copied string is always one. As they are copied, the characters are inserted at the beginning of STRING2. When the loop is completed, STRING2 holds a reversed copy of STRING1.

DISPLAYING TEXT IN THE DRAWING WINDOW

It is not difficult to place text in the Drawing window. You have undoubtedly seen some of the fonts that are available on the Macintosh. All of them become available when text is displayed graphically.

The procedure WRITEDRAW will display any text that can be displayed with WRITE or WRITELN. Several procedures are available that control the font, size, and style of the displayed text. Here is a program that will be used to demonstrate all three text characteristics:

```
program GraphicText;
begin
  showdrawing;
  textfont(0);
  textface([ ]);
  textsize(0);
  moveto(5, 50);
  writedraw('Twas brillig, and the slithy toves')
end.
```

The actual text display is performed by WRITEDRAW. Since this is a graphics procedure, it is necessary to move the drawing pen to the point at which writing is supposed to start. This is done by the MOVETO statement. If the MOVETO statement were missing, no text would be displayed.

The other procedures fairly obviously manipulate text characteristics. Let's examine them one at a time.

Normally, text is drawn using 12 point Chicago type. TEXTFONT is used to select alternative

fonts. It accepts one parameter, an integer which indicates a Macintosh font. A few of the available fonts are:

0	Chicago	6	London
1,3	Geneva	7	Athens
2	New York	8	San Francisco
4	Monaco	9	Toronto
5	Venice		

You won't have all of these fonts on your MacPascal disk, but you can copy them from other disks using the Macintosh Font Mover. An excellent way to preview the fonts is to look at them in MacPaint.

TEXTSIZE determines the size of displayed text in points. For best results your disk font file should contain the appropriate size for the font you have selected. Not all fonts display well in all sizes, but you should be able to find some combinations you will be happy with. A parameter of 0 will set the size to 12 point, which is also the default size if no text size is given by your program. Other font sizes include 9, 14, 18, 24, and 32.

The final control over text display is the text style. Styles include bold face, outline, underline, and so forth. The parameter for TEXTFACE is different from any we have seen before. It is a set of the attributes that are desired. If bold type is required, the procedure call would be:

```
textface([bold])
```

Any of these constants may appear in the set: BOLD, ITALIC, UNDERLINE, OUTLINE, SHADOW, CONDENSE, and EXTEND. To combine styles, any number of constants may appear, separated by commas. To print italic, outline type, for example, use this procedure call:

```
textface([italic, outline])
```

■ Before you continue, experiment with different parameters to these text control procedures using the program Graphic Text. Of course, you are encouraged to write your own programs to put the procedures to work. Make note of any difficulties you may encounter. What happens when text prints past the right side or the bottom of the Drawing window?

When text is printed in the Text window, MacPascal does quite a bit for us. New lines are automatically begun when text exceeds the right edge of the window, moving whole words if necessary to prevent them from being broken. If printing extends past the bottom line of the window, the text in the window scrolls up so that the new text is displayed. We have very few worries when the Text window is used, but we also have a great many limitations. No alternate fonts are available in the Text window, for example.

To print text in the Drawing window, we must control the location, and we must watch out for the right and bottom margins. Quite often, this means that we will want to know how tall and how wide printed text will be. The two most important procedures for this are STRINGWIDTH and GETFONTINFO.

GETFONTINFO tells us how tall and wide the characters in a font are, as well as the amount of space that will separate characters. The most important information here is height of the character. We rarely need to worry about the width.

We must be concerned with three important dimensions of characters. These dimensions are illustrated in Fig. 11-1. All characters rest on an imaginary *base line*. The *ascent* is the height of the

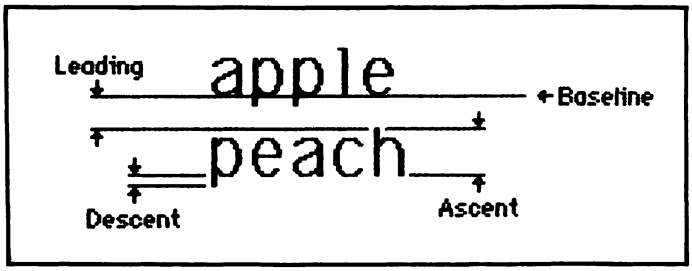


Fig. 11-1. The size characteristics of text.

tallest character above the base line. The *descent* is the distance characters such as p, q, g, and j extend below the base line. Finally, the *leading* (a term derived at a time when strips of lead metal were inserted to space rows of type) indicates the amount of space that should appear between the descenders of one line and the ascenders of the next line.

When MOVETO is used to position the pen for drawing text, the vertical dimension determines the location of the base line for the text that will be "drawn".

To use GETFONTINFO we must first declare a variable of the type FONTINFO. This type is predefined by MacPascal:

```
type
  fontinfo = record
    ascent : integer;
    descent : integer;
    widmax : integer;
    leading : integer;
  end;
```

Since we must start new lines manually, we may use this information to calculate how far we must move printing down the window when a new line is begun. Unless something unusual is desired, the sum of the ASCENT, DESCENT, and LEADING fields will indicate a pleasing spacing.

GETFONTINFO takes into account the font, the point size, and the text style that are in effect at the moment. It is used in this version of the GRAPHICTEXT program to start a new line of type.

■ Change the font or the point size and notice that the location of the new line is adjusted accordingly.

```
program GraphicText1;
  var
    info : fontinfo;
    totalheight : integer;
begin
  showdrawing;
  textfont(0);
  textface([]);
  textsize(14);
  moveto(5, 50);
  getfontinfo(info);
```

```
    totalheight := info.ascent + info.descent + info.leading;
    writedraw('Twas brillig, and the slithy toves');
    moveto(5, 50 + totalheight);
    writedraw('Did gyre and gimble in the wabe;')
end.
```

When printing a long string, how do we know when a new line must be started? Somehow, for each word to be printed, our program must determine whether or not the present line has sufficient space remaining. If the word will not fit, a new line must be started.

MacPascal provides a procedure STRINGWIDTH that determines the width of a string, taking into account the current font, text size, and style. Here is a simple procedure that uses STRINGWIDTH:

```
procedure drawword (string1 : string);
  var
    pt : point;
begin
  getpen(pt);
  if (screenwidth - pt.h) < stringwidth(string1) then
  begin
    currentline := (currentline + height);
    moveto(lmargin, currentline)
  end;
  writedraw(string1);
end;
```

The procedure uses several values that must be determined elsewhere in the program. SCREENWIDTH holds the width of the Drawing window. CURRENTLINE is the vertical location of the drawing pen at the time the procedure is called. The total height of the current font is stored in HEIGHT. Finally, LMARGIN is the horizontal location at which any new line should begin. We will see how these values were established in a little while.

The procedure's operation is quite simple. After storing the current pen location in PT, the space remaining on the line is calculated using the expression SCREENWIDTH - PT.H. If this value is less than the width of the string to be printed, as determined by STRINGWIDTH, a new line must be started. The distance to move down is determined by adding the HEIGHT of the font to the location of the CURRENTLINE. A call to MOVETO repositions the pen. After this, the string is drawn using DRAWSTRING.

To apply this procedure, let's modify the program WORD_BY_WORD, which was written earlier in the chapter. Refresh your memory about the way that program works; then examine the modifications in this new version:

```
program word_wrap;
  const
    space = ' ';
    screenwidth = 200;
  var
    string1 : string[80];
```

```
    location : integer;
    height, lmargin, currentline : integer;     {new variables}
    pt : point;                  {new}
    info : fontinfo;             {new}

  procedure drawword (string1 : string);    {new procedure}
  begin
    getpen(pt);
    if (screenwidth - pt.h) < stringwidth(string1) then
    begin
      currentline := (currentline + height);
      moveto(lmargin, currentline)
    end;
    writedraw(string1);
  end;

begin    { main program }
  showdrawing;
  textsize(12);                                 {new lines begin}
  getfontinfo(info);
  lmargin := 5;
  height := info.ascent + info.descent + info.leading;
  currentline := 5 + height;
  moveto(lmargin, currentline);
  string1 := 'Everything is funny as long as ';
  string1 := concat(string1, 'it is happening to someone else.');
  location := pos(space, string1);
  if location > 0 then
  repeat
    drawword(copy(string1, 1, location));     {change writeln}
    delete(string1, 1, location);
    location := pos(space, string1)
  until location = 0;
  drawword(string1);                            {change writeln}
end.
```

A number of lines have been added at the beginning of the main program. These are required to set up the variables used by DRAWWORD and to initialize the location of the drawing pen. After these new lines, the only change to the main program is to substitute DRAWWORD for WRITELN in two places.

■ Enter the changes and try the program. Take some time to make sure you fully understand it. Change the font size, or introduce statements to modify the font or the type style. Until you use a rather large type font, the program will work well. At some point, however, a weakness will become

evident as the last line of text disappears below the bottom of the drawing window. We will not attempt to fix this problem, but I wanted you to be aware of it. The easiest way to avoid it is to limit text display to an amount that will fit into the window. More elaborate fixes are available, but the best exceed the scope of this book.

If we wish to use this approach with non-strings, the data must be converted to string form. It is not that WRITEDRAW will not work with other data types. WRITEDRAW will print anything that can be printed with a WRITE or a WRITELN statement. The problem is that data must be in string form before STRINGWIDTH can determine the width of the text.

The easiest way to perform the type conversion is to use the function STRINGOF, which is similar to WRITE in that it will accept the same parameters. However, instead of displaying text to the screen, the text is output in string form. Change the STRING assignment standards in WORD_WRAP to read as follows:

```
string1 := stringof('It is ', (2 * 3 = 5), ' that 2 * 3 = 5. ');
string1 := stringof(string1, 'Actually, 2 * 3 = ', (2 * 3) : 2, '.');
```

Be very careful when typing these statements. I have purposely mixed things up quite a bit to illustrate the versatility of STRINGOF. In the second assignment statement, by including STRING1 in the parameter list of STRINGOF, the equivalent of a CONCAT is performed. MacPascal has so many string functions and procedures that there are frequently several ways to achieve a particular result. By using several in turn while you are learning to program, you will begin to learn which approach is best to use in a particular situation.

MIXING TEXT WITH GRAPHICS

There really is very little to learn about mixing text and graphics. It is simply a matter of using MOVETO to position the pen and then using WRITEDRAW to perform the printing. Since drawings and text are located using the same coordinate systems, this is not very difficult.

■ In Chapter 9, we created a program to draw a bar chart; MULTIDICE_GRAPH. Retrieve that program, and refresh your memory of how it works. We are going to add two labels to each bar. To the left of the bar, we will print the dice value that the bar represents. To the right, we will print the number of times that value was thrown.

First, we will make a couple of changes to the CONST block. These will reduce the lengths of the bars, allowing room for the text. The amounts of the changes were determined by experimentation. One problem with coordinating text and graphics is that the size of text is not continuously variable. We can draw the bars of the graph any width or length we desire. However, type comes only in a few point sizes. There is nothing between 9 and 12 point, for example. Also, there is no easy formula for selecting text that meets a certain size specification. You will, therefore, probably do what I did, experiment with type styles and locations until pleasing results are achieved.

■ Modify the CONST block of your program like this:

```
const
  dice = 3;
  maxroll = 18;
```

```
throws = 1000;
maxwidth = 140;          {change MAXWIDTH}
left = 20;               {change LEFT}
```

The program as presented in this chapter will use 12 point type. This works well when two or three dice are being thrown, but it will be too tall if more dice are used, since the bars would become too thin. The left end of the bar has been moved right 20 units, and the right end has moved left 40 units. This will allow for two characters on the left and four on the right.

From here, it is extremely easy to label the graph.

■ Add the indicated statements below to the final FOR loop in the program and try it.

```
for i := dice to maxroll do
begin
  bottom := top + barwidth;
  right := left + round(correction * roll[i]);
  if odd(i) then
    penpat(dkgray)
  else
    penpat(ltgray);
  paintrect(top, left, bottom, right);
  penpat(black);
  framerect(top, left, bottom, right);
  top := bottom;
  moveto(0, bottom - 1);          {new lines begin}
  writedraw(i : 2);
  moveto(right + 1, bottom - 1);
  writedraw(roll[i] : 1)          {new lines end}
end
```

In several places, it was necessary to move the text one or two units to separate an edge of the characters from the edge of a box. The base line of text will be drawn at the vertical location selected by MOVETO. Also, the left edge of the first character in text will often be at the horizontal location appearing in the last MOVETO. If we had not adjusted the drawing location of the text, it would have been drawn right next to the edge lines of the adjoining boxes.

There are several changes you might make to this program. These activities would help pull together the things you have read about in the last few chapters.

- Since the bars are now labeled, the printed report is no longer necessary. Enlarge the Drawing window to the full size of the screen.
- Introduce a CASE statement that uses the number of bars to determine the text size that will be used to label them. You will want to use the largest font that will fit the bars. To select the proper font for each case, you will have to use GETFONTINFO to determine the heights of several fonts.

- Add text down the left side of the screen telling what the numbers represent. You could do this with a large number of WRITEDRAW statements. However, a more interesting solution would be to use a procedure derived from the program WORD_BY_WORD. This procedure could draw one character at a time from a string, proceeding down the edge of the screen.

YOUR PASCAL VOCABULARY

You now know these Pascal words. New ones are printed in bold face type.

Reserved Words

PROGRAM	BEGIN	END	VAR
OF	PROCEDURE	FUNCTION	CONST
TYPE	DO		

Statement Types

Assignment (:=)	Compound		
FOR..TO	FOR..DOWNTO	WHILE	REPEAT..UNTIL
IF..THEN	IF..THEN..ELSE	CASE	WITH

Data Types

BOOLEAN	CHAR	INTEGER	REAL
STRING	DOUBLE	EXTENDED	LONGINT
ARRAY	RECORD		

Graphics Data Types

RECT	POINT	FONTINFO

Procedures

READ	READLN		
WRITE	WRITELN		
NOTE			
FRAMERECT	PAINTRECT	FRAMEOVAL	PAINTOVAL
INVERTRECT		INVERTOVAL	
MOVETO	LINETO	LINE	
PENPAT	PENSIZE	PENMODE	
BUTTON	GETMOUSE		

SHOWTEXT	SHOWDRAWING	SETDRAWINGRECT	SETTEXTRECT
CONCAT	**INCLUDE**	**INSERT**	**STRINGOF**
COPY	**OMIT**	**DELETE**	**POS**
TEXTFONT	**TEXTFACE**	**TEXTSIZE**	**WRITEDRAW**

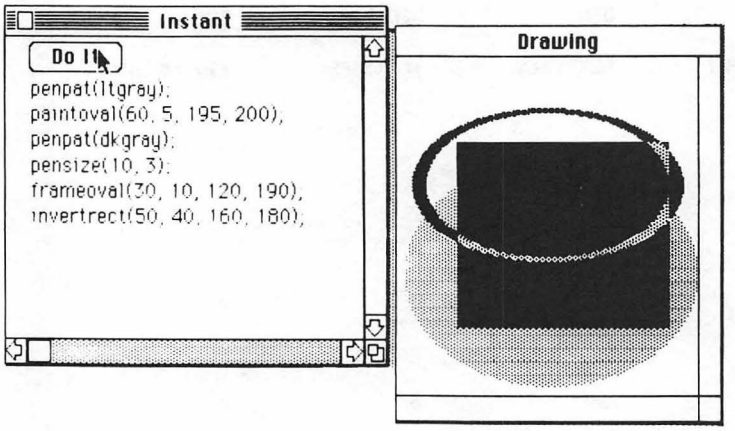
Operations

+	-	*	/
DIV	MOD		
>	>=	<	<=
<>	=		
NOT	AND	OR	IN

Functions

ROUND	TRUNC	SIN	COS
RANDOM	ORD	SUCC	PRED

Chapter 12



Structured Types: Files

In the previous chapters, you have learned only two ways to enter data into programs: through assignment statements and through READ statements. Data entered with READ statements disappear when program execution is completed. This is a real disadvantage since it does not let us keep permanent records. Imagine a grade book program that forgot all of the grade records when it ended; it would have very little practical value.

Assignment statements do allow us to store data with programs since the data can be made a part of the statement. However, assignment statements also have a number of limitations. They cannot be modified by the program. Therefore, new information cannot be added to the program except by editing the program. Also, assignment statements are very cumbersome when large quantities of data are concerned. One thousand data items require one thousand assignment statements.

Files get around these limitations by storing data in a permanent storage device, in our case on the Macintosh's magnetic disks. Files stored on disks, like programs, are retained when the computer is turned off. Also, any number of programs can access the information in a file, allowing data to be exchanged. Data in files may be changed, added to, or deleted. All of the problems listed in the previous paragraphs may be solved using files.

TOPICS COVERED IN THIS CHAPTER

- Declaration of file variables
- Opening sequential files with REWRITE and writing data to files
- Opening sequential files with RESET and reading data from files
- How WRITE and READ affect the file
- The end-of-file marker and the EOF function

- Closing files
- Copying and modifying sequential files
- Opening random access files with OPEN
- Locating and changing components in random access files
- Using text files
- The end-of-line marker and the EOLN function
- Reading and writing different data types to text files
- How READ and READLN function when used with text files
- The standard files INPUT and OUTPUT
- Sending output to a printer
- Directly controlling the file buffer: the buffer variable, GET, and PUT

AN INTRODUCTION TO FILES

In Pascal, File is a structured data type. To use files, variables must be declared appropriately. For example:

```
type
  f : file of integer;
```

Files are similar to arrays in that a given file can store only one type of data. F is a file variable that will be used with Integer data.

Now that a file variable has been created, the file may be opened. To open a file, preparing it to store new data, the REWRITE procedure is used. For example:

```
rewrite(f, myfile);
```

This procedure call initializes a disk data file named MYFILE. REWRITE tells Pascal that the variable F is to be used to store data into that file. This done with a variation of the WRITE statement. If F is typed as a FILE OF INTEGER, this statement will store the value 123 in the file MYFILE:

```
write(f, 123);
```

You might confuse this with WRITE statements you have seen before, where multiple data elements were separated by commas. The value of F is not written to the Text window. Since F is a variable of type File, Pascal treats this WRITE statement differently. F is used to identify the file to be written to, and 123 is the value that is stored.

Incidentally, don't try to use WRITELN to store data in files—more on this later in the chapter.

Here is a program that will store string values into a file:

```
program writefile;
  var
    f : file of string[20];
    x : string[20];
begin
  rewrite(f, 'testfile');
  repeat
```

```
    write('Talk to me: ');
    readln(x);
    if length(x) > 0 then
      write(f, x);
  until length(x) = 0
end.
```

■ Create the program and try it, entering three or four words, such as "apple", "orange", and "banana." To end the program, simply press the Return key in response to the input prompt. Notice how LENGTH provides a convenient way to determine if an empty string was entered. Also notice that the disk drive starts up periodically while the program is executing. At these times, the information you are entering is being stored on the disk.

Before examining the method of getting information back out of the file, let's look at your disk contents.

■ Quit Pascal and examine the icons in the Pascal window. A new icon named "testfile" has been added. The icon shape indicates that the file is a document. It is here that the strings you typed have been stored.

Of course, files would be of little value if we could not retrieve the information that was stored in them. Let's look at a program that will do that.

■ Restart Pascal and enter this program:

```
program readfile;
  var
    f : file of string[20];
    x : string[20];
begin
  reset(f, 'testfile');
  while not eof(f) do
    begin
      read(f, x);
      writeln(x)
    end
end.
```

■ Before any explanation is made, run the program. The strings you entered with the WRITEFILE program will be retrieved and displayed by READFILE.

HOW FILES WORK

There are a number of metaphors for data files. The one I like is to picture a file as a long strip of paper that is behind a moving window. The window prevents us from seeing more than a small piece of the paper. To see other sections on the strip, the window must be moved across the paper to reveal new sections.

Using this analogy, a file of strings can be pictured like this:

```
apple [orange] banana eof
```

Each item is a *component* of the file. However, only one component is visible, just as if we were viewing the file through a window. In this case, the window is over the second component, the string 'orange'. This is shown by enclosing the component with a box. At the end-of-file component Pascal keeps an end-of-file marker, shown as "eof."

REWRITE prepares a file to receive data, and it always clears out the file. When a new file is created by REWRITE, it contains only the eof marker. Data is placed in the file using WRITE, which always inserts new data just ahead of the eof marker. If the first string you typed was 'apple', the file would look like this after the WRITE statement was executed:

apple[eof]

Notice that the window can still see only the eof marker. During writing, the eof marker has little significance. The important thing to notice is that the window is always located after the last component of the file. REWRITE only permits us to add items to the file; we cannot retrieve them. When a file is opened with REWRITE, it is called a *write-only* file.

To retrieve information from the file, the file must be opened with RESET. Suppose that the strings 'apple', 'orange', and 'banana' were stored in the file. If the file is opened with RESET, it can be pictured like this:

[apple]orange banana eof

Notice that the window is over the first component of the file. This is the component that we can retrieve with READ. Executing READ(F,X) has the following effect on the file:

apple [orange] banana eof

X now has the value of the string that was just read. Also, the window has moved on to the next component. Each READ retrieves the value in the window and moves the window to the next component.

This can continue until the last component has been read. At this time, the file can be pictured like this:

apple orange banana [eof]

Eof is not a component of the file and cannot be read. Attempting to do so will produce the error message, "An attempt was made to access data beyond the end of a file." Since it is so important to be able to tell when the end of a file has been reached, Pascal provides the function EOF. This function was used in READFILE to detect an end-of-file condition for the WHILE loop. EOF requires a file variable as a parameter. For a file referenced by the file variable F, EOF(F) will be false whenever the window of the file is over a component. If the window is at the end of the file, however, EOF(F) will be true.

When a file is opened with REWRITE, EOF will always be true. This is of little significance, since we cannot read from a file opened with REWRITE.

The first time a program opens a file, a file variable must be associated with the name of a disk file. REWRITE always has the same effect: it creates an empty disk file, ready to receive data. If a disk file already exists with same filename that appears in the REWRITE parameters, the file is cleared out and opened. Otherwise a new file will be created. After a file is opened with REWRITE, WRITE may be used to place data in the file. READ, however, will be rejected since the file is opened for write-only.

RESET can only open files that have already been created by REWRITE. RESET(F, 'TESTFILE') will result in an error message unless the file TESTFILE already appears on the disk.

Once a file is opened and associated with a file variable, the filename need no longer be specified.

Here is a program that combines the two programs seen so far. First it inputs strings and stores them in TESTFILE. Then it resets TESTFILE and reads the data back out for display:

```
program WriteAndRead;
  var
    f : file of string[20];
    x : string[20];
begin
  rewrite(f, 'testfile');
  repeat
    write('Talk to me: ');
    readln(x);
    if length(x) > 0 then
      write(f, x);
  until length(x) = 0;
  reset(f);
  while not eof(f) do
    begin
      read(f, x);
      writeln(x)
    end
end.
```

The RESET procedure call does not specify a filename, but Pascal has no trouble identifying the file to read the data from since F was associated with TESTFILE in the REWRITE procedure call.

Occasionally, it is necessary to break the association of a file variable with a file. This is the opposite of opening the file, and is performed with the procedure CLOSE. To close the file F, include the statement:

```
close(f);
```

All files are automatically closed when a program terminates.

You have probably noticed that READ and WRITE work with files very much as they have when they were READing data from the keyboard and WRITEing data to the screen. Just about everything you have learned about READ and WRITE applies where files are concerned. There are a few exceptions, however. READLN and WRITELN will not work with conventional data files, and Pascal will produce an error message if you try to use them. Also, field width parameters will have no effect on the storage of data. Real numbers will be stored in exponential form, regardless of the field width parameters used.

This brings us to ask which data types may be stored into files. Actually, files can be declared to store just about any data type. This is not surprising with the simple data types such as Char, Integer, and Real. However, files can also store records, strings, and enumerated types. About the only type that cannot appear in a file type definition is File itself. We cannot have a FILE OF FILE OF INTEGER, for example.

MODIFYING THE CONTENTS OF FILES

One way to modify a Pascal data file is to copy portions of the file into a new file, along with the modifications. Let's build two programs. The first will generate a file containing the integers 0 through 50. The second program will examine the file and produce a new file containing only the even integers.

```
program countfile;
  var
    i : integer;
    intfile : file of integer;
begin
  { create the file }
  rewrite(intfile, 'oldfile');
  for i := 0 to 50 do
    write(intfile, i);
  { display file contents }
  reset(intfile);
  while not eof(intfile) do
    begin
      read(intfile, i);
      write(i)
    end
end.
```

This program creates a file named OLDFILE and writes fifty-one integers into the file using a FOR loop. The WHILE loop is there simply to show the contents of the file, confirming its contents.

- Enter and run the program.

Now, here is the updated program:

```
program WriteEven;
  var
    i : integer;
    fromfile, tofile : file of integer;
begin
  reset(fromfile, 'oldfile');
  rewrite(tofile, 'newfile');
  { copy even entries in fromfile into tofile }
  while not eof(fromfile) do
    begin
      read(fromfile, i);
      if not odd(i) then
        write(tofile, i)
    end;
  { display the new file }
  reset(tofile);
  while not eof(tofile) do
    begin
      read(tofile, i);
      write(i)
    end
end.
```

In this program, two data files must be open at the same time, which presents no problem. The only requirement is that each file be associated with an appropriate file variable. OLDFILE is the file created in the last program, containing both even and odd random numbers. OLDFILE is associated with the variable FROMFILE.

The disk file NEWFILE will be newly created by this program and will contain the even numbers that are selected from OLDFILE. NEWFILE will be accessed through the variable TOFILE.

The process of placing the even values into NEWFILE is quite simple. A value is read from OLDFILE into the variable I. If I is not odd, it is written to NEWFILE. This continues until EOF is TRUE for FROMFILE, at which time the first loop stops.

A final loop displays the contents of NEWFILE to confirm that it contains only the even numbers that were in OLDFILE.

■ Enter and run this program, paying attention to the length of time required to complete the task of copying the integers.

Suppose that we have no further need for the data in OLDFILE and that we would like to copy the data in NEWFILE back into OLDFILE. Modify the program so that, when execution is completed, OLDFILE will contain only even integers.

WORKING WITH RANDOM-ACCESS FILES

Because they must always be worked with one component at a time, starting at the beginning of the file, REWRITE and RESET open files that are called *sequential access* files.

There is another kind of file called a *random access* file. With these files, any component of the file may be directly accessed without going through the preceding components. Random access files also permit us to change any component in the file without erasing the rest of the components, as is done by REWRITE. You may have noticed that it took an appreciable amount of time to read the positive integers from one file and to write them into a new one. The larger the files become, the slower this process is. By using random access files, many of these copy operations can be avoided.

In this sense, random access files are similar to arrays. They are, however, much slower in operation since the disk drive is a mechanical device, whereas arrays are manipulated entirely in the computer's much faster electronic memory.

When a file is opened with the procedure OPEN, it has both random access and read/write capability. This means that we can read from the file to locate a component to be changed and then write an updated component in its place. Files are opened with OPEN in the familiar manner.

```
open(data, 'datafile');
```

opens a file named DATAFILE and associates it with the file variable DATA. The file window is set to the first component of the file. The difference between OPEN and RESET or REWRITE is that either READ or WRITE procedures may be used following OPEN.

When a file is opened with OPEN, several new procedures become available. SEEK may be used to move the file window directly to any component of the file.

■ If you have changed the contents of OLDFILE, use the program COUNTFILE to restore the Integers to the file.

■ Here is another sort of update program. It examines the file OLDFILE, making every third entry negative. Try it. Then we will examine the update method used.

```
program Negate;
  var
    i, component : integer;
    intfile : file of integer;
begin
  { negate every third component of file }
  open(intfile, 'oldfile');
  component := 2;
  seek(intfile, component);
  while not eof(intfile) do
    begin
      read(intfile, i);
      seek(intfile, component);
      write(intfile, -i);
      component := component + 3;
      seek(intfile, component);
    end;
  { display the file contents }
  reset(intfile);
  while not eof(intfile) do
    begin
      read(intfile, i);
      write(i)
    end
end.
```

File components are numbered starting with zero. To avoid an Off-By-One Error, the third component must be found by seeking for component number 2. Before the WHILE statement begins, SEEK (INTFILE,2) positions the file windows over the third component in the file. The WHILE loop begins by reading the value of this component into the variable I.

Remember that READ and WRITE both advance the file window to the next component. If we want to write back in the same position, therefore, a SEEK must be performed again prior to executing the WRITE statement. Following the WRITE statement, the COMPONENT variable is incremented by three, and a SEEK is performed to position the window to the next element.

If an attempt is made to SEEK past the end of the file, the window is placed over the end-of-file mark. In the present program, this is the signal to end the WHILE loop. This feature can also be used to position the window to the end of the file so that a value may be appended. For a file F, SEEK(F,MAXLONGINT) will always position the window to the end-of-file, since a file cannot contain more than MAXLONGINT components.

If we add the function FILEPOS, we have the capability of writing a simple data base which may be updated on request. FILEPOS returns the current position of the file window.

A SIMPLE INVENTORY PROGRAM

The simple data base we will create will be an inventory file. It will keep track of items by name

and will keep track of the quantity on hand for each item. We will have two capabilities: adding a new item to the inventory and changing the quantity on hand.

One of the nice things procedures allow us to do is to outline programs using calls for procedures that are not yet written. In fact, we can outline the entire program very simply.

```
program Simple_Inventory;
  type
    item_data = record
      name : string[10];
      quantity : integer;
    end;
    instring = string[10];


  var
    item : item_data;
    stock : file of item_data;
    com : char;

  procedure add;
    { add a new item to stock file }
  end;      {of add}

  procedure change;
    { enter a new quantity and store in stock file }
  end;    { of change }

begin   {main program}
  showtext;
  open(stock, 'Stockfile');
  repeat
    write('Add,  Change, or Quit? ');
    read(com);
    case com of
      'a', 'A' :
        add;
      'c', 'C' :
        change;
      otherwise
        ;
    end; { of case }
    {display a stock listing after every successful command}
    if pos(com, 'aAcC') > 0 then
      while not eof(stock) do
```

```
            begin
              read(stock, item);
              write('Item: ', item.name : 10);
              writeln('Quantity: ' : 15, item.quantity : 5)
            end;
    until com in ['q', 'Q'];
end.
```

This program skeleton maps out the essentials of the program. The main program functions quite simply. A character is read to determine the operation to be accomplished. A CASE statement uses this character to determine a procedure to be executed.

Only six characters produce results. Upper- and lowercase "A" and "C" trigger specific procedures. All other characters are handled by the OTHERWISE clause of the CASE statement, which does nothing. If the character was "Q" or "q", however, it will cause the REPEAT loop to terminate.

A printout of the stock file contents will be displayed after an ADD or a CHANGE has been performed. By now, the display of a file contents is a straightforward process, so the code for this has been included in the first program outline.

The data type ITEM_DATA is the most important feature in the declarations. This record definition has only two components, not enough for a realistic inventory program, but enough for demonstration purposes. ITEM_DATA is the component type for the data file STOCK.

Obviously, the program cannot do much in its present state; the procedures are empty shells. However, the main program can be executed and tested. Later, when the procedures ADD and CHANGE are developed, they will be placed in a program that is known to be working. This makes it easy to isolate problems. If a bug emerges just after a working version of ADD is merged with the program, we can be fairly certain that the problem is in ADD, not in the rest of the program, which has been tested.

The rest of the programming effort has been broken down into two tasks: development of the ADD and CHANGE procedures. Let's outline the things these procedures must accomplish:

```
procedure add;
begin
  {enter the name of the item to be added}
  if item is found then
    write('That item is already in inventory.')
  else
    begin
      {enter the quantity of the item}
      {add the item to the stock file}
    end { of else }
end;     {of add}
```

Large portions of this outline are English phrases. Each comment indicates the need for one or more Pascal statements. The IF..THEN statement is outlined in nearly complete Pascal form. However, the conditional part of the statement is obviously not a proper Pascal expression. These lines must be refined into Pascal statements. By developing a general outline of the procedure and by concentrating on each portion of the outline separately, the process of developing the procedure is broken down into small, manageable parts.

When developing such outlines, use your knowledge of Pascal to get as much into them as possible. I included an IF..THEN..ELSE statement outline because I was pretty sure that this was the way to proceed. Don't try to do too much in your first outlines of procedures, however. This is a perfectly acceptable first pass at the procedure.

Now we need a similar outline for CHANGE:

```
procedure change;
begin
  {enter the name of the item to be changed}
  if item is found then
    begin
      {enter the new quantity}
      {replace the current file entry with the new one}
    end
  else
    writeln('That item is not in inventory')
end;   { of change }
```

Both ADD and CHANGE require a method of determining if an item is already established in the data file. ADD must only add a new item if it has not been entered already. CHANGE, obviously, cannot alter an entry that is not in the file. Therefore, we will assume the existence of a function FOUND that will output TRUE if an item is found in the stock file.

We can now turn our attention to refining ADD. Entering the new information is no problem. Adding the new data is possible since the file will be opened by OPEN. SEEK can be used to position the file window to the end of the file; then a simple WRITE will add the item. With this established, the rest of the procedure is easy to write:

```
procedure add;
  var
    newitem : instring;
begin
  write('Name of item: ');
  {enter the name of the item to be added}
  readln(newitem);
  if found(newitem) then
    writeln('That item is already in inventory.')
  else
    begin
      {enter the quantity of the item}
      item.name := newitem;
      write('Quantity on hand: ');
      readln(item.quantity);
      {add the item to the stock file}
      seek(stock, maxlongint);          { prepare to append }
      write(stock, item);
```

```
        writeln('New item added to inventory.');
        writeln
      end { of else }
end;     {of add}
```

The only thing remaining is to develop the function FOUND. The function should accept a string, look for the string in the name fields of the stock file, and output a Boolean value determined by the success of the search. At first, it may be desirable to simulate the function so that ADD can be tested. A dummy version of FOUND that will always output FALSE can be easily produced. This will let us test the ability of ADD to insert a new item in the stock file.

```
function found (key : instring) : boolean;
begin
  found := false;
end;
```

■ Enter the skeleton program that was introduced earlier, inserting the ADD procedure as just developed. Also include FOUND, which must, of course, appear before ADD in the program.

■ Test the program. You will want to increase the size of the Text window to accommodate the wide lines of text that will be produced by the program. At the prompt, use the A option to add the following items:

Item	Quantity
screw	1000
bolt	850
nut	700

After each addition, a report lists all items in the inventory. Each new item is added as the last item in the stock file. At this point, the program will accept duplicate entries for items. To eliminate this, the FOUND function must be completed.

```
function found (key : instring) : boolean;
  {outputs true if key is found in stock file}
begin
  reset(stock);
  item.name := '';
  while ((item.name <> key)) and (not eof(stock)) do
    read(stock, item);
  found := item.name = key
end;
```

Once a file is opened with OPEN, it retains its read/write capability even though RESET or REWRITE are used. RESET is used here as a convenient way to position the window to the beginning of the file. ITEM.NAME must be initialized since it is a global variable which almost certainly contains a product name. If the present value of ITEM.NAME matches the value of KEY, then the WHILE statement will never execute.

The WHILE loop reads each component of the file as long as the key is not found or until the end of the file is reached. After the loop terminates, FOUND is assigned its output value based on whether ITEM.NAME matches the search key. If the loop terminated because a match was found, FOUND will be TRUE. If the loop reached the end of the file without finding a match, FOUND will be FALSE.

■ Include the final version of FOUND in the program, run it, and try adding a duplicate entry. Be sure that the program reacts appropriately to new and to duplicate entries.

The remaining procedure is CHANGE, which is slightly more complicated than ADD. CHANGE must substitute a new component for one already stored in the file. This requires a three step process:

1. Find the position of the old component.
2. Use SEEK to position the window over the old component.
3. Write the new component.

Here is the CHANGE procedure:

```
procedure change;
  var
    key : instring;
begin
  {enter the name of the item to be changed}
  write('Name of item: ');
  readln(key);
  {if found it's ok to change}
  if found(key) then
    begin
      writeln('Item: ', item.name);
      writeln('Quantity: ', item.quantity);
      {enter the new quantity}
      write('Enter new quantity: ');
      readln(item.quantity);
      {replace the current file entry with the new one}
      seek(stock, filepos(stock) - 1);
      write(stock, item)
    end  { of then }
  else
    writeln('That item is not in inventory')
end;   { of change }
```

Since the WHILE loop in FOUND terminated just after the old entry was found, the current file position is one past the old entry. The expression FILEPOS(STOCK) – 1 therefore provides SEEK with the appropriate location. This approach borders on being a tricky one that could get us in trouble. In order to understand how CHANGE works, we must understand the side effect of FOUND: that it leaves the file window just after the matching component. It is far better to make this explicit by having FOUND assign the file position to a variable. We are then relying not on a side effect of FOUND

but on an intentionally designed feature of the function. A simple modification to FOUND accomplishes this without altering the main mission of the function to determine if a key item is already present in the file.

- Add this variable declaration in the program's main VAR block:

```
position : longint;
```

- Add this line just before the assignment statement in FOUND:

```
position := filepos(stock) - 1;
```

- Change the SEEK statement in CHANGE to this:

```
seek(stock, position);
```

- Test the CHANGE procedure by running the program and making the following entries:

Item:	Quantity:
screw	900
bolt	700
washer	650

The program should accept the changes for "screw" and for "bolt," but it should reject the entry for "washer."

Here is a complete listing of the program SIMPLE INVENTORY:

```
program Simple_Inventory;
  type
    item_data = record
      name : string[10];
      quantity : integer;
    end;
    instring = string[10];
  var
    item : item_data;
    stock : file of item_data;
    com : char;
    position : longint;

function found (key : instring) : boolean;
  {outputs true if key is found in stock file}
begin
  reset(stock);
  item.name := '';
  while ((item.name <> key)) and (not eof(stock)) do
    read(stock, item);
  position := filepos(stock) - 1;
```

```
    found := item.name = key
  end;

  procedure add;
    var
      newitem : instring;
  begin
    {enter the name of the item to be added}
    write('Name of item: ');
    readln(newitem);
    if found(newitem) then
      writeln('That item is already in inventory.')
    else
      begin
        {enter the quantity of the item}
        item.name := newitem;
        write('Quantity on hand: ');
        readln(item.quantity);
        {add the item to the stock file}
        seek(stock, maxlongint);        { prepare to append }
        write(stock, item);
        writeln('New item added to inventory.');
        writeln
      end { of else }
  end;      {of add}

  procedure change;
    var
      key : instring;
  begin
    {enter the name of the item to be changed}
    write('Name of item: ');
    readln(key);
    {if found it's ok to change}
    if found(key) then
      begin
        writeln('Item: ', item.name);
        writeln('Quantity: ', item.quantity);
        {enter the new quantity}
        write('Enter new quantity: ');
        readln(item.quantity);
        {replace the current file entry with the new one}
        seek(stock, position);
        write(stock, item);
```

```
        writeln
      end { of then }
    else
      writeln('That item is not in inventory')
  end;   { of change }

  begin   {main program}
    showtext;
    open(stock, 'Stockfile');
    repeat
      write('Add, Change, or Quit? ');
      read(com);
      case com of
        'a', 'A' :
          add;
        'c', 'C' :
          change;
        otherwise
          ;
      end; { of case }
      {display a stock listing after every successful command}
      reset(stock);
      if com in ['a', 'A', 'c', 'C'] then
        while not eof(stock) do
      begin
            read(stock, item);
            write('Item: ', item.name : 10);
            writeln('Quantity: ' : 15, item.quantity : 5)
          end;
    until com in ['q', 'Q'];
  end.
```

TEXT FILES

In addition to standard data files, Pascal has another sort of file called a *text file.* In many ways, text files behave like files of characters. There are a few characteristics, however, that distinguish text files from files of type Char.

Text files may be visualized as lines in a book. The lines may be of varying length. Just as with standard data files, an end-of-file marker is placed at the end of the file. However, text files use another marker that data files do not have, an end-of-line marker.

Text files are ideally suited for the storage of text data. However, anything that can be written to the screen with WRITE or WRITELN can be written to a text file. This excludes structured data types such as records and arrays, but permits integers, real numbers, Boolean values, strings, characters, and enumerated types to be written to text files. Once stored in text files, READ and READLN may be used to retrieve the data.

Notice that READLN and WRITELN may be used with text files while their use is forbidden with nontext files. This is the case because text files are line oriented, and we must have a way of skipping from one line to the next.

Let's create a text file and look at methods involved in retrieving data from it.

■ Enter this program, which will create a simple text file:

```
program WriteTextfile;
  var
    txt : text;
begin
  rewrite(txt, 'textfile');
  write(txt, 12);
  writeln(txt, 'buckle my shoe');
  writeln(txt, '34 shut the door.')
end.
```

Even more than with data files, writing to text files resembles writing to the Text window. Ignore the fact that the program is writing to a file for a moment, and visualize how it would appear if the same WRITE statements were placing text on the screen.

Notice the variable declaration for TXT. TEXT is a type, similar to FILE OF CHAR. TXT is not declared to be a FILE OF TEXT, but simply to be of the type TEXT.

Now we will try to retrieve the text from the file. Here is the first program we'll use:

```
program ReadTextfile;
  var
  txt : text;
  s : string;
begin
  reset(txt, 'textfile');
  while not eof(txt) do
    begin
      read(txt, s);
      writeln(s);
    end
end.
```

■ When you execute this program, its behavior is rather puzzling. The program prints "12buckle my shoe" and then does nothing else. But Run at the top of the screen continues to be printed in reverse, indicating that the program is still running! Let's try to determine what is happening.

■ Terminate the program by choosing Halt in the Pause menu.

■ Choose Reset in the Run menu. This forces the program to begin with a closed file, canceling the message you received that the file reading would continue where it left off.

■ Start the program by choosing Step-Step in the Run menu. The program will begin to execute very slowly. The special feature of Step-Step is that the pointing hand indicates each statement as it is executed. With this program, the pointing hand clearly indicates that the WHILE statement is executing repeatedly. This behavior is the common bug of the *infinite loop*; the finishing condition

for the WHILE loop is never met, so the loop cannot terminate.

■ The WRITE statement is executing but nothing apparently is being printed. Let's check that out. Change the statement as shown:

```
writeln(s, 'x');
```

■ Execute the program with Step-Step. Notice that an "x" is printed each time the WRITELN statement executes. The WRITELN statement must be printing S also. The problem, therefore, is that S contains no value that can be printed.

As noted earlier, an end-of-line marker is found at the end of each line in a text file. These markers are placed in the file whenever a WRITELN is used to store information. WRITE statements do not place end-of-line markers.

Just as WRITELN statements start new lines when writing to text files, READLN statements cause the program to skip to the next line after reading to any variables that are specified in the parameter list.

The problem in the current version of the program is that READ cannot begin a new line. If a READ occurs when the file window is at an end-of-line marker, nothing is read. When we have tried to read past the end of a file, Pascal has always informed us with an error message. This will not happen when we try to read past the end of a line. Pascal is perfectly happy to perform the READ statement. In the present case, the infinite loop occurs because the READ statement can never get past the end-of-line marker. Therefore EOF(TXT) can never become TRUE and the program can never terminate.

Notice how READ or READLN operate when reading from a text file into a string variable. Everything from the present window position to the next end-of-line marker is assigned to the variable. A single word cannot be read unless it is read one character at a time. When reading strings, be sure that the string variable has sufficient capacity to receive the text. Although text files can contain "lines" of any length, no more than 255 characters can be read into a string variable.

■ Choose Halt to terminate the program.

■ Change the READ to READLN and run the program again. The text window should look like this when the program is completed:

```
        12buckle my shoe
 34 shut the door
```

Why are the lines printed differently? Actually, the text on the screen exactly reflects the way the text was stored in the text file. 12 was stored as an integer. Therefore, Pascal stored it in a standard width integer field, with leading spaces. Since we didn't include any spaces in the string 'buckle my shoe', none were used to separate 12 and "buckle."

The second line was written to the file as a single string, however; so it was stored without leading spaces.

Even though the data were stored as strings, they need not be retrieved that way.

■ Try this version of READTEXTFILE. It retrieves the numbers as integers:

```
program ReadTextfile;
  var
  txt : text;
  i : integer;
begin
```

```
  reset(txt, 'textfile');
  read(txt,i);
  writeln(i);
  read(txt,i);
  writeln(i)
end.
```

A little problem developed: the second READ could not be accomplished since the window was not pointing to a digit. When READ is accepting data from the keyboard, it terminates entry of a number when a character is read that cannot appear in data of the present type, either Integer or Real. READLN will ignore illegal characters, waiting for you to either type more digits or to press the RETURN key.

When getting its data from a text file, however, READ is not so forgiving. If a READ is performed and an acceptable character is not in the file window, READ aborts the program, printing an error message. We must be much more careful when reading from text files than when reading from the keyboard.

In this case, it is necessary to move the file window to the next line so that the second READ will encounter acceptable data. This can be done in two ways. The program will work fine if the first READ is changed to a READLN. Alternatively, a READLN can be inserted ahead of the second READ. If this READLN contains no variable for receiving data, the only effect will be to move the window to the beginning of the next line.

- Insert this line before the second READ and execute the program:

```
readln(txt);
```

The two numbers will be read and printed as we wished.

Imagine that V is a variable and the file TXT contains the string:

```
98.6 degrees
```

The following statement is executed:

```
read(txt, v);
```

What value will be read into V if V is of type Integer? Real? Char? String?

If V is of type	This value is read
Integer	98
Real	98.6
Char	9
String	98.6 degrees

READ always obtains data from a text file one character at a time. Reading stops when a character is encountered that is incompatible with the type of the variable that will receive the data. When V is of type Integer, READ stops at the decimal point. When V is Real, READ stops at the "d". Char variables will accept only one character, so READ terminates after that character is read. String variables, however can accept any data in a text file; READ does not stop storing data in a String variable until the end of a line is reached. READLN behaves just like READ, while data is being read. The only difference is that READLN moves the file window to the beginning of the next line after the read terminates.

So far, READ has caused us a great deal of trouble. Does this mean that we want to avoid using READ with text files? No, it means that READ and READLN must be used appropriately. Here is a program that is intended to read the text from TEXTFILE one character at a time. Let's see how well it works.

■ Enter and run the program:

```
program ReadChars;
  var
    txt : text;
    ch : char;
begin
  reset(txt, 'textfile');
  while not eof(txt) do
    begin
      readln(txt, ch);
      write(ch)
    end
end.
```

It didn't work did it? All you got were the first characters in each of the two lines of the file: a space (shown as a blank line) and the character "3". After the first character in the file was read, READLN moved the window to the next line. In that line, after the character was read, the READLN also moved to the next line, which was past the end of the file. This caused EOF(TXT) to be TRUE and the loop terminated.

So this is a case where READ must be used.

■ Change the READLN to READ and try the program again.

We are back to the infinite loop. Neither READ nor READLN works. Does this mean that we cannot read the file using character variables?

Actually we can do it, if we take steps to detect the end-of-line marker when it is reached. Pascal provides an end-of-line function. EOLN returns TRUE when the end of a line has been reached. To read the file, we need to read each line until EOLN is TRUE. Then a READLN is performed to move to the next line. To accomplish this, we will use a nested loop:

```
program ReadChars1;
  var
    txt : text;
    ch : char;
begin
  reset(txt, 'textfile');
  while not eof(txt) do
    begin
      while not eoln(txt) do
        begin
          read(txt, ch);
          write(ch)
```

```
        end;
      readln(txt);
      writeln
    end
end.
```

The outer WHILE loop is the familiar one that reads from a file until EOF is TRUE. The inner loop reads the characters from a single line until EOLN is TRUE. After the line is completed, READLN(TXT) moves the window to the first character of the next line. A WRITELN is also performed to start a new line for the text being printed on the screen.

WRITE and READ can contain multiple expressions in their parameter lists. Anything you can WRITE to the screen can be written to a text file in a similar manner. The only requirement is the addition of a file variable to the beginning of the parameter list.

Several of the points made or implied in this section should be emphasized:

- READ cannot obtain data from a file when EOLN is TRUE.
- If a READLN is performed when EOLN is TRUE, no data will be read, even if variables are included in the READLN parameter list. The only effect of READLN in this case is to start a new line in the file.
- All data is read from a text file one character at a time, even though a READ with an integer, real, or string variable may capture several characters.
- A READ terminates when a character is encountered that is incompatible with the type of the variable into which the date is being read.
- READLN always moves the file window to the beginning of the next line after the read takes place.
- After READLN, EOLN will always be FALSE.
- READLN presents difficulties when reading into Char variables since only one character per line will be read.
- READ presents difficulties with Strings since EOLN will always be TRUE after a string is read with READ. The program will hang until a READLN is performed.
- If READ or READLN encounters a data type mismatch, the program will be aborted.

THE USE OF THE INPUT AND OUTPUT FILES

You have probably noticed the similarities between writing to a text file and writing text to a screen, or between reading from the keyboard and reading from a text file. This is not coincidental, since the keyboard and the screen are defined as special Pascal text files.

Every version of Pascal predefines two files that correspond to the standard input and output devices of the computer. In the case of the Macintosh, input is assumed to come from the keyboard unless Pascal is informed differently. The keyboard corresponds to the special file INPUT.

The normal output device with MacPascal is the Mac's video monitor. Unless we indicate another destination, all writing operations are directed toward the screen, which corresponds to the special file OUTPUT.

Both INPUT and OUTPUT are text files. INPUT is a read only text file, while OUTPUT is write only.

To direct writing to a file, a file variable is placed at the beginning of the WRITE or WRITELN parameter list. If no file variable appears, Pascal assumes that text should be displayed to the screen.

Actually, it is perfectly all right to use OUTPUT as a file parameter in a WRITE statement. These two statements are functionally equivalent:

```
write(output, 'message');

write('message');
```

Also, these two statements have the same effect:

```
read(input, s);

read(s);
```

Because they are text files, many of the problems we have encountered with text files apply to INPUT and OUTPUT, and in particular to INPUT.

■ Here is a program that tries to use READ to input two strings from the keyboard. When you run it, you will find that it does not work smoothly:

```
program Strings;
  var
    s : string;
begin
  read(s);
  writeln;
  writeln(s);
  read(s);
  writeln;
  writeln(s)
end.
```

You will notice that you are only permitted to enter one string. Recall that EOLN is always TRUE after a string has been read by READ. The first READ causes EOLN to be TRUE, and the second READ does not read anything, since READ automatically ends when EOLN is TRUE. If you edit the program, changing each READ to READLN, it will work properly. Moral: when entering strings from the keyboard, always use READLN. There is no case in which READ offers advantages when strings are being input.

Technically, INPUT and OUTPUT are supposed to appear in a program parameter list if the keyboard input or screen output are to be performed. Most of the programs we have written, then, should have had headings like this:

```
program example (input, output);
```

When INPUT and OUTPUT are included in this way, the associated files are automatically opened when the program begins to execute. MacPascal is unusual in that these program file parameters are not required. These files are opened automatically for every program.

Because INPUT and OUTPUT are opened automatically, and because they must remain open, standard Pascal does not allow them to be opened with OPEN, RESET, or REWRITE. However, there is one case allowed by MacPascal, in which it is useful to apply REWRITE to OUTPUT. The state-

ment REWRITE(OUTPUT) will clear the Text window, moving the text cursor to the top left corner. In later programs, this feature will be used to clear old text from the window.

INPUT AND OUTPUT WITH MACINTOSH DEVICES

Screens and keyboards are not the only things Pascal views as files. If you have a printer attached to your Mac, you can use file techniques to direct output to it. The printer is treated as a write-only file with the name PRINTER:. Here is a program that will print text. It assumes that a Macintosh compatible printer has been properly attached to your Mac's printer connector.

```
program printdemo;
  var
    print : text;
begin
  rewrite(print, 'printer:');
  write(print, 'Aren''t you glad');
  writeln(print, ' you have a permanent copy of this sentence?')
end.
```

■ If you have a printer, enter the program and run it.

Most printers have special features that are turned on and off by sending control codes. Some of these codes include nonprintable characters, characters that you cannot type from the keyboard. For example, a character known as the ESCAPE character is commonly used. On the Apple Imagewriter printer, ESCAPE followed by the character "X" turns on underlined printing. ESCAPE followed by a "Y" ends underlined printing.

The "X" can be written directly, but the ESCAPE must be output by using the CHR procedure. Recall that CHR outputs a character associated with an integer parameter. CHR(27) outputs the ESCAPE character.

■ If your printer is an Imagewriter, we can underline some characters in the printer demonstration program. Modify the first WRITE statement of the demonstration program to this form:

```
write(print, 'Aren''t you ', chr(27), 'X', 'glad', chr(27), 'Y');
```

You will find many other printer control codes in your printer manual. Most manuals indicate not only the names of the special characters but also their ASCII codes. ASCII codes are standard codes used to identify characters in microcomputers. To enter these ASCII codes in WRITE statements, simply include them as the parameters of CHR statements.

Often we would like to choose when text will be written to the screen or to the printer. This is quite possible through control of the file parameter in the program WRITE statements. However, because OUTPUT is a special file device, certain complications are involved.

Normally, a file parameter is not included when a WRITE statement is expected to print to the Text window. However, OUTPUT may appear as a file parameter, and these statements will have the same effect:

```
write('hello');

write(output, 'hello');
```

To direct output to a printer, a file parameter must be included. We would like to have a single statement that can easily direct output either to PRINTER: or to OUTPUT. This should be possible through the use of a new file variable. Suppose that we have a variable OUTFILE, which is assigned an appropriate value. Would this enable the following statement to write either to the printer or to the screen, depending on the value of OUTFILE?

```
writeln(outfile, 'hello');
```

This can be determined by experimenting with two programs. Each uses an identical WRITELN statement, but the output goes to different places.

```
program ToPrinter;
var
  outfile : text;
begin
  rewrite(outfile, 'printer:');
  writeln(outfile, 'This goes to the printer.')
end.

program ToScreen;
var
  outfile : text;
begin
  outfile := output;
  writeln(outfile, 'This goes to the screen.')
end.
```

Individually, these programs work just fine. In TOPRINTER, a REWRITE statement is used to associated OUTFILE with the printer.

However, OUTPUT is a file variable, and it cannot appear as the device parameter of a REWRITE statement. OUTFILE cannot be associated with OUTPUT by saying REWRITE(OUTFILE, 'OUTPUT'). Fortunately, it works when an assignment statement is used to associate the two files. We now have methods that, though different, accomplish our goals.

However, a difficulty arises when we try to combine the statement portions of the two programs.

■ Enter the following program. Be sure to save it before you run it!

```
program ToBoth;
  var
 outfile : text;
begin
  rewrite(outfile, 'printer:');
  writeln(outfile, 'This goes to the printer.');
  outfile := output;
  writeln(outfile, 'This goes to the screen.')
end.
```

When you ran the program, the first string was successfully sent to the printer. However, before anything could be sent to the Text window, a major error occurred. The window you saw is displayed by the Mac when an error occurs that cannot be ignored or corrected. The only way out of this error is to click the RESTART box. This will cause a reboot of the Mac.

■ After the reboot is complete, start Pascal by opening the program you just saved. A simple modification will correct it so that it will run perfectly.

The problem is that OUTFILE begins by being associated with an open PRINTER: file. Then, without any warning, the program attempts to ignore that association and to connect OUTFILE to OUTPUT. This hopelessly confuses the Mac, which has different ways of dealing with the printer and the screen displays. To solve the problem, the association with the printer must be completely broken before you attempt to establish another output path. This is done by closing the printer file.

■ Add this line before the assignment statement in the program:

```
close(outfile);
```

The program will now function properly.

Let's take the inventory program from earlier in this chapter and add the ability to produce a printed report on demand.

■ Make the indicated modifications to the main program:

```
begin  {main program}
  showtext;
  open(stock, 'Stockfile');
  repeat
    write('Add, Change, Print, or Quit? ');    {add Print option}
    read(com);
    writeln;
    outfile := output;                         {new line}
    rewrite(outfile);
    printer := false;                          {new line}
    case com of
      'a', 'A' :
        add;
      'c', 'C' :
        change;
      'p', 'P' :                               {add case for P}
        begin
          close(outfile);
          rewrite(outfile, 'printer:');
          printer := true
        end;
      otherwise
      ;
  end; { of case }
  reset(stock);
```

```
    {display a stock listing after every successful command}
    if pos(com, 'aAcCpP') > 0 then                    {add p and P}
    while not eof(stock) do
      begin                                           {add OUTFILE to}
        read(stock, item);                            {writeln stments}
        write(outfile, 'Item: ', item.name : 10);
        writeln(outfile, 'Quantity: ' : 15, item.quantity : 5)
      end;

      if printer then                                 {new IF statement}
        begin
          for i := 1 to 40 do
            write(outfile, '=');
          writeln(outfile);
          close(outfile)
        end;
    until com in ['q', 'Q'];
  end.
```

■ Three variable declarations must be added to the VAR block of the program:

```
outfile : text;
i : integer;
printer : boolean;
```

Now to examine the program revisions. These new statements:

```
outfile := output;
rewrite(outfile);
printer := false;
```

associate the text variable OUTFILE with the file OUTPUT. The REWRITE is done simply to clear the Text screen. PRINTER is set to FALSE so that the new IF statement will normally not execute.

A new case has been added to process a Print request:

```
'p', 'P' :                                    {case for P}
  begin
     close(outfile);
     rewrite(outfile, 'printer:');
     printer := true
  end;
```

First, OUTFILE is closed. This does not close OUTPUT, which will still be used as the default file for text output. MacPascal does not permit OUTPUT to be closed. (Don't count on this with other versions of Pascal!)

Following the close, OUTFILE may be reopened with the device PRINTER:. Subsequent output

then is directed to the printer. The Boolean variable PRINTER is set TRUE to enable this IF statement at the end of the main program:

```
if printer then
  begin
    for i := 1 to 40 do
      write(outfile, '=');
    writeln(outfile);
    close(outfile)
  end;
```

The FOR loop draws a double line on the printer to separate entries. After this, a WRITELN starts a new line and OUTFILE may be closed. OUTFILE is now free to be reassigned to output at the top of the REPEAT loop.

■ Be sure to try the new addition to the program. You will immediately see that the ability to print reports adds greatly to the value of the inventory program.

MANIPULATING THE FILE BUFFER

Recall that two distinct actions occur when READ is applied to a file:

1. The data in the file window is assigned to the variable in the READ parameter list.
2. The window is moved to the next item in the file.

Similarly, when a WRITE takes place, these two things happen:

1. The value of the data variable in the WRITE parameters is stored in the file window.
2. The window is moved to the right.

On occasion, it is desirable to have separate control over these functions. This requires us to look more closely at the sequences of events that surround the reading and writing of data in files.

We have frequently referred to a file window. Technically, this window is known as the *file buffer.* The file buffer is a place in the computer's memory that contains the one value in the file that we can manipulate at any given moment.

Whenever a file is opened, Pascal automatically creates a *file buffer variable.* This variable has the same name as the file-type variable that is associated with the file, except that it is tagged with a circumflex: "^". For example, this statement:

```
rewrite(datfile, 'employee_data');
```

automatically creates the variable DATFILE^. It is through the file buffer variable that data is read from or written to the file.

Early in the chapter, a data file was diagrammed using a box to indicate which item of data was in the window. The window corresponds to the file buffer variable.

The value in the buffer variable is always available to the program. A simple assignment statement can be used to extra this value. For example:

```
i := f^;
```

The buffer variable may also appear in more complex expressions such as:

```
i := round(f^ / 3);
```

Thus it is no problem to extract data from a file. However, this alone does not move the file buffer to the next component in the file. To move the buffer, GET is used.

```
get(f);
```

moves the file buffer to the next component in the file F. The buffer variable then takes on the value of this new component.

It may be seen, therefore, that READ combines these two operations.

```
read(f, i);
```

is equivalent to these two statements taken together:

```
i := f^;
get(f);
```

If GET is applied when no next component exists, EOF(F) becomes TRUE and the value of F ^ is undefined.

To store a value in the file buffer it is merely necessary to assign the value to the buffer variable. For example:

```
f^ := i;
```

Following such an assignment, it is necessary to use PUT to store the file buffer in the file and to move the buffer forward in the file:

```
put(f);
```

Thus, WRITE can be expressed in terms of an assignment statement and a PUT.

```
write(f, i);
```

performs the same task as these two statements:

```
f^ := i;
put(f);
```

If EOF(F) is TRUE, PUT appends the value of F^ to the end of the file and EOF(F) remains true. The value of F^ is undefined, as it always is when EOF(F) is TRUE.

YOUR PASCAL VOCABULARY

You now know the following Pascal words. New words are printed in bold face type.

Reserved Words

PROGRAM	BEGIN	END	VAR
TO	PROCEDURE	FUNCTION	CONST
TYPE	DO		

Statement Types

Assignment (:=)	Compound		
FOR..TO	FOR..DOWNTO	WHILE	REPEAT..UNTIL
IF..THEN	IF..THEN..ELSE	CASE	WITH

Data Types

BOOLEAN	CHAR	INTEGER	REAL
STRING	DOUBLE	EXTENDED	LONGINT
ARRAY	RECORD	FILE	

Graphics Data Types

RECT	POINT	FONTINFO

Procedures

READ	READLN		
WRITE	WRITELN		
NOTE			
FRAMERECT	PAINTRECT	FRAMEOVAL	PAINTOVAL
INVERTRECT		INVERTOVAL	
MOVETO	LINETO	LINE	
PENPAT	PENSIZE	PENMODE	
GETMOUSE			
SHOWTEXT	SHOWDRAWING	SETDRAWINGRECT	SETTEXTRECT
INSERT	DELETE		
TEXTFONT	TEXTFACE	TEXTSIZE	WRITEDRAW
RESET	REWRITE	OPEN	CLOSE
SEEK	GET	PUT	

Operations

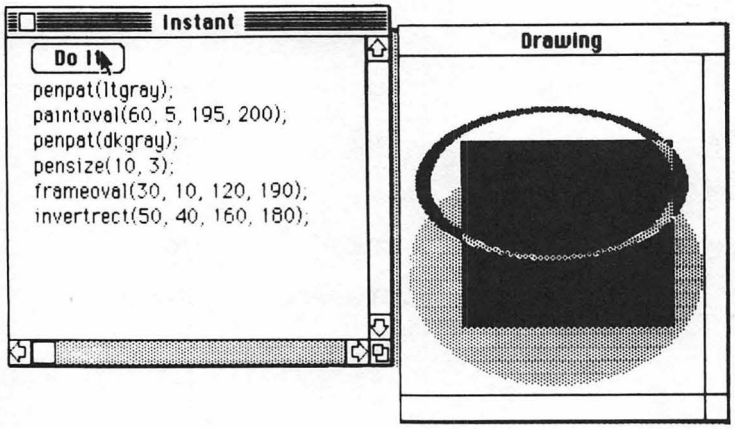
+	-	*	/
DIV	MOD		
>	>=	<	<=

<>	=		
NOT	AND	OR	IN

Functions

ROUND	TRUNC	SIN	COS
RANDOM	ORD	SUCC	PRED
BUTTON			
LENGTH	COPY	CONCAT	POS
OMIT	INCLUDE	STRINGWIDTH	STRINGOF
EOF	**EOLN**	**FILEPOS**	

Chapter 13



Searching, Inserting, and Shuffling

The need to order data in arrays or files is frequently encountered. Most often we simply want to display the data in alphabetical or numerical order. However, many processes are made easier or faster if data are stored in ordered form. A good example is the process of searching for an item in a large array. As this chapter will demonstrate, a search can be performed much faster if it can take advantage of the ordered nature of an array or file.

There are two ways to produce ordered arrays and files. If the data are not presently ordered, it can be sorted. However, sorting can be a time consuming process. If the data is already in the desired order, new items can be simply inserted into their proper locations, saving considerable time. This chapter will examine two techniques of inserting data into files and arrays. Sorting, a more complex topic, will receive a chapter of its own.

Occasionally, it is desirable to randomize the order in which data are stored. When writing a card game, for example, it is necessary to shuffle the deck. The last program in this chapter will demonstrate a method of shuffling the data in an array.

TOPICS COVERED IN THIS CHAPTER

- Sequential searching of arrays
- Binary search techniques for arrays and files
- Inserting new data into ordered arrays and files
- Shuffling of arrays

SEARCHING FOR DATA IN ORDERED ARRAYS: SEQUENTIAL SEARCH

The most obvious way to look for one item in an array is to examine each element in turn until

the desired element is found. This can be done with a simple loop, the strategy employed by this program:

```
program lookup;
  const
    size = 676;
  type
    shortstring = string[2];
    stringarray = array[1..size] of shortstrin
  var
    a : stringarray;
    s : shortstring;
    loc, i, j : integer;
    time : longint;

  function lookup (s : shortstring;
        a : stringarray) : integer;
  var
    i : integer;
  begin
    i := 1;
    while (i < size) and (a[i] <> s) do
      i := i + 1;
    if a[i] = s then
      lookup := i
    else
      lookup := -1
    end;

begin
  showtext;
  for i := 0 to 25 do
      for j := 0 to 25 do
        a[i * 26 + j + 1] := stringof(chr(i + 97), chr(j + 97));
  repeat
    write('type string to look for: ');
    readln(s);
    if length(s) > 0 then
      begin
        writeln;
        time := tickcount;
        loc := lookup(s, a);
        write('the lookup took ');
        writeln((tickcount - time), ' 60ths of a second');
        if loc >= 0 then
```

```
            writeln(s, ' is element ', loc)
          else
            writeln(s, ' was not found');
          end
      until length(s) = 0
    end.
```

The main program begins by using nested FOR loops to create an array with 676 elements. The values in the array range from 'aa' to 'zz'. This is a rather complex loop for its size and deserves some examination. Each pass through the inner loop, the expression I * 26 + J + 1 is used to index the array. If you substitute values, you will discover that the expression counts from 1 to 676 as the nested loop executes.

For example, if I is 0 and J is 12, the array index will be 13. Or, if I is 21 and J is 5, the array index will be 552. Perform some value substitutions of your own to be sure you understand how the array index is produced.

The CHR function outputs characters, determined by adding the values of I and J to 97. The lowercase letter "a" has an ordinal value of 97. Therefore, when I is 0, the expression CHR(I + 97) outputs the character 'a'. When I is 25, the expression outputs the character 'z', which has an ordinal value of 122.

Each loop counter variable is used to produce a character. These characters are assembled into a string by the STRINGOF function, and the strings are stored in the array. Since the J loop is nested within the I loop, J counts from 0 to 25 before a new value is determined for I. The first pass through the J loop, therefore, the strings 'aa' through 'az' are stored. I is then incremented, causing the next J loop to store the strings 'ba' through 'bz'. The final result is that the array A will contain strings from 'aa' to 'zz' in alphabetical order.

After the array is established, a REPEAT loop is used to control the remainder of the program. First the program requests a string to search for. If you enter a string of zero length (one entered by pressing the Return key without typing any characters), the IF..THEN statement will not execute the compound statement that calls the LOOKUP function. A string of zero length also terminates the REPEAT loop.

Since we will be comparing the time requirements for two search strategies, I have included statements to determine how long the lookup takes. TICKCOUNT outputs the time since the Macintosh system was last started. The output of TICKCOUNT is a long integer that represents elapsed time in 60ths of a second. Before a lookup is begun, the current value of TICKCOUNT is stored in the variable TIME.

The program then calls on the function LOOKUP, which determines if the string in S is found in the array A. Immediately after the lookup is completed, the elapsed time is calculated and printed.

LOOKUP is a very simple function. It accepts two parameters, a string variable and a string array. LOOKUP also depends on the value of the global variable SIZE, the value of which is the number of elements in the string array. A WHILE loop is used to examine each element of the array, continuing until the value of i equals the value of SIZE or until a match is found. After the loop terminates an IF statement checks to determine if the search string was found, in which case A[i] will equal S. If S was found, the THEN clause outputs an integer representing the location of the string in the array. If the string was not found, then −1 is output.

The value output by LOOKUP is used to determine which message will be printed by the IF statement in the main program.

■ Try the program, searching for the strings 'aa', 'mz', and 'zz'. Notice that considerably more

time is required to find 'zz' than 'aa'. Almost four seconds is needed to find 'zz' while 'aa' is found in about a tenth of a second. This time increase is to be expected since 676 comparisons must be made to find 'zz', but only one is required for 'aa'.

On the average, this version of LOOKUP must perform half as many comparisons as there are elements in the array. As the array size increases, the average search time also increases. Imagine relying on this approach with arrays having several thousand elements. If a great number of lookups were to be performed, we would probably find ourselves wishing for a faster strategy.

A BINARY SEARCH STRATEGY

This faster strategy may be discovered by playing a simple game. Have a friend determine a secret number between 1 and 1000. Your job is it guess the number. Your friend is allowed to make only the following responses:

- If your guess is too low, your friend can say, "That's too low."
- If your guess is too high, your friend can say, "That's too high."
- If your guess is correct, your friend can tell you so.

How many guesses, on the average, do you need to determine the number? If you are just guessing, you probably are not doing too well.

If you are using an optimum strategy, however, you will never need more than ten guesses. In fact, you can always guess a number between one and one million in 17 guesses or less!

The strategy is a simple one: always guess the middle number in the range of numbers that may contain the unknown number. In that way, you can always eliminate half of the remaining possibilities.

For example, if the number is between 1 and 1000, guess 500. Whether you are high or low, you will eliminate half the numbers. If you were low, the number is between 500 and 1000. Guess 750 for your second guess. High this time? Guess 625. Each time, half of the remaining numbers are eliminated from consideration. This search style is called a *binary search.*

This approach can be directly implemented in a procedure.

■ Substitute this version of LOOKUP for the version presently in the program:

```
function lookup (s : shortstring;
                 a : stringarray) : integer;
  var
    midpoint, bottom, top : integer;
begin
  bottom := 1;
  top := size;
  repeat
    midpoint := (bottom + top) div 2;
    if s < a[midpoint] then
      top := midpoint - 1;
    if s > a[midpoint] then
      bottom := midpoint + 1;
  until (s = a[midpoint]) or (bottom > top);
  if bottom <= top then
```

```
    lookup := midpoint
  else
    lookup := -1
end;
```

■ Run the program and search for the strings 'aa', 'mz', and 'zz' again. Compare the search times to the times you obtained when the first version of LOOKUP was being used. Here are the times I got when I was testing the different search strategies:

Search String	Sequential Search	Binary Search
aa	5	12
mz	157	6
zz	220	13

The times you obtain can vary, sometimes by a large amount. I occasionally timed binary searches of more than a second. These erratic times occur because a single microprocessor is controlling every function in the Macintosh. Pascal is sharing the microprocessor with the screen display, the clock/calendar, the mouse machinery, and so forth. If one of these functions demands attention, Pascal may be temporarily slowed. Most of the times you obtain, however, should be similar to the times in the chart.

When searching for 'aa', the binary search required more time than the sequential search. When searching for 'zz', however, the binary search was the winner by an overwhelming margin. The binary search for 'mz' was very fast since 'mz' was one of the first array elements examined by the procedure. It is readily apparent that the binary search is faster in the majority of cases.

The only catch with the binary search is that the array must be sorted in alphabetical order. If the array is the least bit disordered, we cannot guarantee that a target will be found with a binary search. In those cases, a sequential search is the only method guaranteed to find the target. In fact, the order of data in an array is irrelevant with a sequential search, and a search on a disordered array generally proceeds as quickly as a search on an ordered array.

Let's examine the way the binary search procedure works. BOTTOM and TOP are the variables used to indicate the range of array elements that has not yet been eliminated. We will call the part of the array that has not yet been eliminated the *search space*. As the procedure begins, the search space includes the entire array. BOTTOM receives the initial value of 1, while TOP is assigned the value in SIZE. As in the first version of the procedure, a value must be assigned to SIZE somewhere else in the program.

At the beginning of the loop, the middle element in the array is found by summing BOTTOM and TOP and then dividing by 2. During the first pass through the REPEAT loop, MIDPOINT has a value of 338. The array element A[338] has a value of 'mz'.

Now, one of three things will happen:

- If S < A[MIDPOINT] then we can eliminate the array from MIDPOINT up to TOP. TOP is therefore given the value of MIDPOINT - 1.
- If S > A[MIDPOINT] then we can eliminate the array from BOTTOM up to MIDPOINT. BOTTOM is therefore given the value of MIDPOINT + 1.
- The only remaining possibility is that S = A[MIDPOINT]. In this case, the search is completed and the REPEAT loop is terminated.

There is one more way to end the loop. During the search process, TOP will be decreasing in value and BOTTOM will be increasing. If the string in S is not found in the array, a point will be reached at which TOP will be found to be less than BOTTOM.

The last step in the function is to test TOP and BOTTOM to see if the search string was found. If TOP is still greater than BOTTOM, the string was found and the value of MIDPOINT becomes the value output by the function. Otherwise a – 1 value, which indicates to the calling statement that the search was a failure, is output.

BINARY SEARCHES IN FILES

The binary search technique is extremely useful with files also. Here is a demonstration program which does for files what we have just done for arrays:

```
program lookup_file;
  type
    shortstring = string[2];
    stringfile = file of shortstring;
  var
    s : shortstring;
    loc, i, j : integer;
    time : longint;
    a : stringfile;

  function lookup (s : shortstring;
            var lookfile : stringfile) : integer;
    var
      midpoint, bottom, top : longint;
  begin
    seek(lookfile, maxlongint);
    bottom := 0;
    top := filepos(lookfile);
    repeat
      midpoint := (bottom + top) div 2;
      seek(lookfile, midpoint);
      if s < lookfile^ then
        top := midpoint - 1;
      if s > lookfile^ then
        bottom := midpoint + 1;
    until (s = lookfile^) or (bottom > top);
    if bottom <= top then
      lookup := midpoint
    else
      lookup := -1
  end;
```

```
begin
  showtext;
  open(a, 'strings');
  for i := 0 to 25 do
    for j := 0 to 25 do
        write(a, stringof(chr(i + 97), chr(j + 97)));
  repeat
    write('type string to look for: ');
    readln(s);
    if length(s) > 0 then
      begin
        writeln;
        time := tickcount;
        loc := lookup(s, a);
        write('the lookup took ');
        writeln((tickcount - time), ' 60ths of a second');
        if loc >= 0 then
          writeln(s, ' is element ', loc)
        else
          writeln(s, ' was not found');
      end
  until length(s) = 0
end.
```

The file variable is passed to LOOKUP as a variable parameter. This is the only way file variables may be passed as parameters to procedures. Attempts to use a value parameter with a file variable will produce an error message.

Notice the use of the file buffer variable in the LOOKUP procedure. By using this variable, it became unnecessary to read the current file component into a variable.

Most of the program is quite similar to the programs you have been working with. By now, you should be able to analyze this version and determine how it works.

■ Run the program and look up several strings. Notice that it rarely takes more than a second to find an item in the file. Much of this time is spend just accelerating the disk in the disk drive. If you enter a new string as soon as one is printed, the search will start while the disk is still turning, resulting in a greatly shortened search time.

INSERTING DATA INTO ORDERED ARRAYS

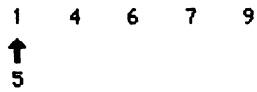
If an array is stored randomly, a binary search is of no value. In a disordered array, only looking at each element in turn can guarantee finding a given string. Since data are not always neatly arranged, we will often need to turn to techniques for rearranging the data into ordered form.

One way to arrange the data is to sort the array in which it is stored. Two techniques of sorting will be examined in the next chapter. At that time, we will see that sorting is a time consuming task. The faster of the two sorts we will consider requires ninety seconds to sort an array containing a thou-

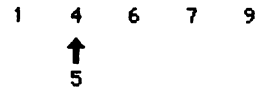
sand elements. Often, therefore, it is good practice to store the data in the array such that it is already in ordered form.

This is done by inserting new data into the array in its correct position. This is a simple enough process. Imagine that an array is storing these five integers:

To insert the integer 5 into this array, the first step is to find its proper location. This may be done by simply comparing 5 to each element of the array in turn. Here, 5 is compared to the first element:



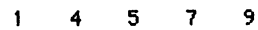
Since 5 is greater than 1, we move on to the next location, comparing 5 to 4:



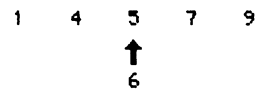
5 is also greater than 4, so we continue:

6 is the first integer in the array that is greater than 5. We now know where 5 should be inserted into the array. However, in order to place 5 in its proper position, all of the greater integers must move up one position.

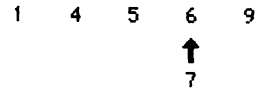
The first step in accomplishing this is to store 5 in the element that currently holds the value 6. The array now looks like this:



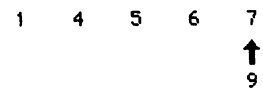
However, when this is done, the 6 is lost. In order to save it so that it can be restored to the array, a temporary variable must be created. Before 5 is stored in the array, the current value of the array variable is copied into the temporary one. We will represent this by placing the 6 below the arrow:



Moving one more position, the 7 is stored in the temporary variable and the 6 is stored in the array in place of the 7.



After another such exchange, 9 is stored in the temporary variable, and the array looks like this:



All that remains is to add 9 to the end of the array:

```
1   4   5   6   7   9
```

Here is a procedure that performs such insertions on an integer array. The type KARRAY is simply an ARRAY[1..100] OF INTEGER. The procedure relies on a global variable SIZE to indicate the number of data items that have been stored in the array.

```
procedure insert (int : integer;
      var intarray : karray);
  var
    temp : integer;
    pointer : 1..max;
begin
  pointer := 1;
  while (intarray[pointer] <= int) and (pointer <= size) do
    pointer := pointer + 1;
  while pointer <= size do
    begin
      temp := intarray[pointer];
      intarray[pointer] := int;
      int := temp;
      pointer := pointer + 1
    end;
  size := size + 1;
  intarray[size] := int;
end;
```

The first thing the procedure must do is to locate the place where the value in INT should be inserted. This is done with a WHILE statement:

```
pointer := 1;
while (intarray[pointer] <= int) and (pointer <= size) do
  pointer := pointer + 1;
```

The variable POINTER is used to index the array, starting with the first element. The WHILE loop simply increments the pointer until INT is greater than INTARRAY[POINTER]. POINTER now indicates which array element INT should be stored in. The WHILE loop will also terminate if POINTER becomes larger than SIZE; this indicates that INT must be stored as the new last element of the array.

Another loop is used to insert the value of INT and to move the remainder of the array up:

```
while pointer <= size do
  begin
    temp := intarray[pointer];
    intarray[pointer] := int;
    int := temp;
```

```
    pointer := pointer + 1
  end;
```

TEMP is used to store the current value of INTARRAY[POINTER]. After that value is saved, INT can be stored in the array variable.

Returning to the example array presented earlier, here is the array after the 5 has been inserted. Also shown are the values of the variables involved:

```
                1    4    5    7    9

INT = 5         TEMP = 6         POINTER = 3
```

Now the value of TEMP must be stored in element 4 of the array.

We cannot simply store TEMP in INTARRAY[4], since this would cause the value of INTARRAY[4] to be lost. We must store 7 in a temporary variable, just as was done with 6 before 5 was stored. Does this mean that we need yet another temporary variable?

The solution is to assign the value of TEMP to INT. Since the value of INT has already been stored in the array, nothing is lost. Once the value has been stored in INT, TEMP may be used to store the 7 from INTARRAY[POINTER].

The moral is that when swapping the values of two variables, you always need a third variable which can be used to temporarily store one of the values.

After INT contains the next value to be stored, POINTER can be incremented and the loop can repeat. Using this strategy, the sample variables would have these values at the bottom of the loop:

```
                1    4    5    7    9

INT = 6         TEMP = 6         POINTER = 4
```

The procedure is now ready to repeat the loop, thus storing 6 in position 4 and moving 7 into the TEMP variable.

After the 7 has been stored, TEMP will still have a value of 9, the last value of the original array. This is stored in the array with a final WRITE statement. The procedure also increments SIZE, since an element has been added.

```
size := size + 1;
intarray[size] := int;
```

■ To test the procedure, run this program, which will insert 100 randomly selected integers into an array. When you type it in, be sure to include the procedure INSERT, which we just examined:

```
program Insertion;
  const
    max = 100;
  type
    karray = array[1..max] of integer;
  var
    mainarray : karray;
    x, count : 1..max;
    size : 0..max;
    time : longint;
```

```
  {include procedure INSERT here}
begin { main program }
  showtext;
  size := 0;
  time := tickcount;
  for count := 1 to max do
      insert(abs(random), mainarray);
  time := (tickcount - time);
  for count := 1 to (max div 10) do
  begin
    for x := 1 to 10 do
      write(mainarray[(count - 1) * 10 + x] : 6);
    writeln
  end;
  writeln('time required: ', time / 60 : 10 : 2)
end.
```

The expression ABS(RANDOM) is capable of generating the full range of positive integers. Each in turn is inserted into INTARRAY with the INSERT procedure.

After all the integers are inserted, a final pair of loops prints out the contents of the array. The loops are nested and configured so that ten values will be printed on each row of the screen.

Following printout of the array, the elapsed time is printed. This time is calculated using the methods we examined when doing lookups earlier in the chapter.

■ Try the program. Examine the final values in the array to confirm proper operation of the program. Also, make a note of the time required. We are about to see if we can speed up the insertion process.

IMPROVING INSERTION WITH A BINARY SEARCH

When looking up values, we found that the process could be speeded up by using a binary search strategy. The first part of INSERT uses a sequential search. Can we speed up the procedure by improving the search strategy?

A binary search for INSERT is a bit more complicated than it was in LOOKUP since the search must do more than simply locating an item in the array. If the item is not found, the search must indicate where it should be inserted. Several more possibilities must therefore be considered. For example:

- What should happen if the new value is greater than the last value in the array?
- What should happen if the new value is smaller than the first value in the array?
- Where should the new value be inserted if it does not match a current value in the array?

These questions are all taken into account by the following function, which outputs the position at which the value of INT should be inserted into the array:

```
function location (int : integer;
          intarray : karray) : integer;
  var
    bottom, top, mid : integer;
```

```
begin { location }
  if size = 0 then
    location := 1
  else
    begin { #1 }
      bottom := 1;
      top := size + bottom - 1;
      if int > intarray[top] then
        location := top + 1
      else if int < intarray[bottom] then
        location := 1
      else
        begin { #2 }
          repeat
            mid := (top + bottom) div 2;
            if int > intarray[mid] then
              bottom := mid;
            if int <= intarray[mid] then
              top := mid;
          until (int > intarray[mid]) and (int <= intarray[mid + 1]);
          location := mid + 1
        end { for begin #2 }
    end    {for begin #1 }
end;    {location}
```

If SIZE = 0 then there are no values in the array that the new value may be compared to. In this case, LOCATION is simply given the value of 1. If SIZE is greater than 0, the ELSE part of the IF statement performs all of the possible comparisons.

A series of IF..THEN statements examines the various possible locations for INT. If INT is greater than INTARRAY[TOP], it should be inserted at the end of the array and LOCATION is assigned the value of TOP + 1. If INT is less than the first element of the array, it should be inserted at the beginning of the array. LOCATION is then assigned a value of 1. If neither of these conditions holds, then the procedure must look for the data's location within the array.

If an exact match for INT is not found, the insertion point for INT will fall between two array elements. Therefore two conditions must be met to determine if the proper location has been determined. As one example, if INT is greater than INTRARRAY[5] and INT is less than or equal to INTARRAY[6] then INT should be inserted in position 6.

To adapt it for use with insertion, several changes must be made to the original binary search strategy:

- If INT > INTARRAY[MID] then BOTTOM is assigned the value of MID instead of MID+1. This is done since INTARRAY[MID] may be the array element that is just less than INT.
- The second test changes the INT < INTARRAY[MID] test to INT < = INTARRAY[MID].
- If INT < = INTARRAY[MID] then TOP is assigned the value of MID instead of MID–1.

• The test to end the REPEAT loop contains two conditions that must be met, where one sufficed with the LOOKUP function.

Some modification of INSERT is required to take advantage of the LOCATION function.

■ Remove the following lines from the present version of INSERT:

```
pointer := 1;
while (interray[pointer] <= int) and (pointer <= size) do
  pointer := pointer + 1;
```

■ Replace these lines with this statement:

```
pointer := location(int, interray);
```

■ Add the new function LOCATION to the program INSERTION, just before the INSERT procedure.

■ Run the program and note how long it now takes to insert 100 values into the array. It is a bit faster than the version that used sequential search, but not a lot.

There are two parts to an insertion: searching for the new location, and inserting the value. While the search can be speeded up by applying a binary search strategy, there is not much we can do to accelerate the insertion process. We cannot get out of swapping each pair of elements from the insertion point to the end of the array. Unfortunately, it is the insertion, not the search that takes the most time. The new insertion method is faster than the old one, but not remarkably so.

Insertion is a useful technique but it has its limitations. For one, it is difficult to use insertion to change the order of an already sorted array. Suppose, for example, that we had an array of records. We could insert the data according to one field, perhaps a name field. Suppose that we wished to rearrange the data by another field, possibly by age. To do this with insertion, the array would have to be copied to another array. This is time consuming and wasteful of computer memory.

■ Change the value of MAX in the CONST block to 1000. How long does the program take to build an array of 1000 elements by using insertion? Do this only when you can bear to part with your computer for awhile, since it will take over an hour.

Insertion is most useful when a relatively small number of items must be added to an array that is already ordered, or when items are added one at a time with other actions coming between the additions. When many new items are being added to the array all at once, it may be faster to use one of the sorts introduced in the next chapter.

INSERTING DATA INTO FILES

Performing insertions on files is not greatly different from performing insertions on arrays. Much of the difference in the procedures results because the program can address only one file component at a time, while the array version could address array components with much greater freedom.

One of the oddest problems that we must solve with file searches results because FILEPOS cannot be used to tell if a file contains one or no components. Normally, to find the number of components stored in a file, we need simply use SEEK to find the end of the file, and then determine the file position with FILEPOS. The value output by FILEPOS is the number of components in the file, remembering that the first component is numbered as zero.

However, we will illustrate that this will not work if no data have been written to the file. Assume

that the following statements have been executed on a brand new file, where F is declared to be a FILE OF INTEGER:

```
open(f, 'testfile');
seek(f, maxlongint);
writeln(filepos(f));
```

The WRITELN statement will print the number 1. Remember that this is a new file that has never been written to. Also remember that the first component of a file is component number 0. Since FILEPOS(F) outputs a value of 1, we might be fooled into thinking that component 0 contained valid data.

In fact, the following statements will produce an identical result to the ones just discussed:

```
open(f, 'testfile');
write(f, 123);
seek(f, maxlongint);
writeln(filepos(f));
```

FILEPOS outputs the same value whether the file contains one data component or none. It would be more convenient if the first example output 0, since this would indicate that no data had been stored in the file.

In the present situation, we cannot use FILEPOS to determine if a file is new or not. The solution to the dilemma is to ignore component 0 of the file, since we can use FILEPOS to determine if data has been stored in component 1.

The majority of the changes in the INSERT and LOCATION procedures were needed to adapt the program from arrays to files. Here is the file insertion program:

```
program File_Insertion;
  type
    shortstring = string[20];
    stringfile = file of shortstring;
  var
    s, s1 : shortstring;
    sfile : stringfile;
    time : longint;

  function location (s1 : shortstring;
            var lookfile : stringfile) : integer;
    var
      bottom, top, mid : longint;
      s2, s3 : shortstring;
  begin {location}
    seek(lookfile, maxlongint);
    if filepos(lookfile) = 1 then     {see if it's the first entry}
      location := 1
    else
```

```
    begin  {#1}
      bottom := 1;
      top := filepos(lookfile) - 1;
      seek(lookfile, top);          {see if it belongs on top}
      read(lookfile, s2);
      if s1 > s2 then
        location := top + 1
      else
        begin  {#2}
          seek(lookfile, 1);        {see if it belongs on bottom}
          read(lookfile, s2);
          if s1 < s2 then
            location := 1
          else
            begin  {#3}
              repeat
                mid := (top + bottom) div 2;
                seek(lookfile, mid);
                read(lookfile, s2);
                if s1 > s2 then
                  bottom := mid;
                if s1 <= s2 then
                  top := mid;
                seek(lookfile, mid);
                read(lookfile, s2, s3);
              until (s1 > s2) and (s1 <= s3);
              location := mid + 1
            end   {for begin #3}
        end     {for begin #2}
    end       {for begin #1}
end;    { location }

procedure insert (s : shortstring;
          var insertfile : stringfile);
  var
    temp : shortstring;
begin
  seek(insertfile, location(s, insertfile));
  while not eof(insertfile) do
    begin
      read(insertfile, temp);
      seek(insertfile, filepos(insertfile) - 1);
      write(insertfile, s);
```

```
            s := temp;
          end;
        write(insertfile, s);
      end;

begin { main program }
  showtext;
  rewrite(sfile, 'stringdat');   { clear out the file }
  close(sfile);
  open(sfile, 'stringdat');
  write(sfile, '');              { something has to be in component 0 }
  repeat
    write('String to insert: ');
    readln(s);
    if length(s) > 0 then
      insert(s, sfile);
    seek(sfile, 1);
    writeln('The file contents are:');
    while not eof(sfile) do
      begin
        read(sfile, s1);
        write(s1, ' ')
      end;
    writeln;
  until length(s) = 0;
end.
```

■ Run the program and enter several strings. Add strings that belong in the beginning, the end, and in the middle of the list of strings that is built, to demonstrate that the demonstration program really does work.

The main program in this version begins by clearing the file. This means that data stored during one running of the program will be erased at the beginning of the next. Often, however, we want files to accumulate data, retaining it even if the program is stopped and rerun. To retain the previous entries, simply remove these lines:

```
rewrite(sfile, 'stringdat');
close(sfile);
```

After the file is opened, an empty string is stored in the first component. Pascal will not allow us to store data in the second component until the first (numbered zero, remember) has been filled. The empty string is just a place holder, and has no further effect on the program.

I think you will be able to puzzle out the file version of this program. I have kept the different versions of INSERT and LOCATION fairly similar, so you should be able to compare them easily.

The big difference in the new version was the need for more variables so that the values of two file components could be examined in the same statement.

SHUFFLING

Every once in a while we need to randomize an array. This necessity probably arises most commonly in games, but other situations suggest themselves. If you are writing a multiple choice test, you might wish to present the answer selections in randomized order so that students could not memorize the positions of the answers, for example.

Compared to insertion, shuffling is a piece of cake. The approach is simple:

1. For each element in the array, randomly select an element from the rest of the array.
2. Swap the two elements.

A simple FOR loop is all we need, as is shown by this procedure:

```
procedure shuffle (var a : stringarray);
  var
    i, j : integer;
    temp : shortstring;
begin
  for i := 1 to size - 2 do
  begin
    j := rand(i + 1, size);
    temp := a[i];
    a[i] := a[j];
    a[j] := temp
  end
end;
```

The procedure uses the RAND function, which we defined in Chapter 7, to select a random number between 1 and SIZE. This number is used along with i to index the array when the values are swapped.

It is only necessary to loop until i is equal to SIZE – 2. This will leave two array elements from which the random selection may be made: A[SIZE] and A[SIZE–1]. No point is served by incrementing i to SIZE – 1 since only one value would remain as potentially selectable.

Using parts of the LOOKUP program used earlier in the chapter, here is a program that uses SHUFFLE to randomize a 676 element array. The array is originally built of two character strings sorted alphabetically, using techniques we have previously examined. Since we already know these statements produce an ordered array, the array is not displayed. You may wish to print out the array to verify the fact that it starts out in an ordered condition.

■ Enter the program, including SHUFFLE where indicated:

After the array is initialized, it is shuffled and printed out.

```
program shuffle;
  const
    size = 676;
```

```
  type
    shortstring = string[2];
    stringarray = array[1..size] of shortstring;
  var
    a : stringarray;
    s : shortstring;
    i, j : integer;

  function rand (lowlimit, toplimit : integer) : integer;
  begin
    rand := random mod (1 + toplimit - lowlimit) + lowlimit
  end;

{ Insert the SHUFFLE procedure }

begin
  showtext;
  writeln('Initializing the array.');
  for i := 0 to 25 do
    for j := 0 to 25 do
      a[i * 26 + j + 1] := stringof(chr(i + 97), chr(j + 97));
  writeln('Shuffling the array.');
  shuffle(a);
  writeln('Here it is:');
  for i := 1 to size do
    write(a[i] : 3)
end.
```

I didn't bother to time the shuffling since it is not being compared to another technique. It takes a fair amount of time since the array is quite large. Since you will probably be working with smaller arrays, you should find the speed of SHUFFLE to be quite adequate.

YOUR PASCAL VOCABULARY

You now know these Pascal words. New ones are printed in bold face type.

Reserved Words

PROGRAM	BEGIN	END	VAR
DO	PROCEDURE	FUNCTION	CONST
TYPE	OF		

Statement Types

Assignment (:=) Compound

FOR..TO	FOR..DOWNTO	WHILE	REPEAT..UNTIL
IF..THEN	IF..THEN..ELSE	CASE	WITH

Data Types

BOOLEAN	CHAR	INTEGER	REAL
STRING	DOUBLE	EXTENDED	LONGINT
ARRAY	RECORD	FILE	

Graphics Data Types

RECT	POINT	FONTINFO

Procedures

READ	READLN		
WRITE	WRITELN		
NOTE			
FRAMERECT	PAINTRECT	FRAMEOVAL	PAINTOVAL
INVERTRECT		INVERTOVAL	
MOVETO	LINETO	LINE	
PENPAT	PENSIZE	PENMODE	
GETMOUSE			
SHOWTEXT	SHOWDRAWING	SETDRAWINGRECT	SETTEXTRECT
INSERT	DELETE		
TEXTFONT	TEXTFACE	TEXTSIZE	WRITEDRAW
RESET	REWRITE	OPEN	CLOSE
SEEK	GET	PUT	

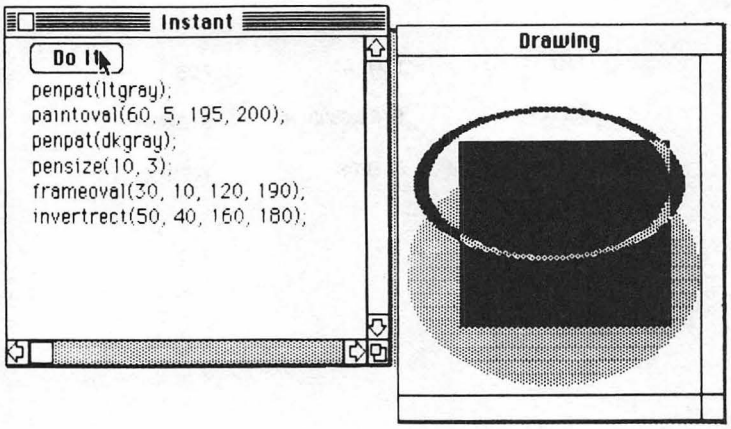
Operations

+	-	*	/
DIV	MOD		
>	>=	<	<=
<>	=		
NOT	AND	OR	IN

Functions

ROUND	TRUNC	SIN	COS
RANDOM	ORD	SUCC	PRED
BUTTON			
LENGTH	COPY	CONCAT	POS
OMIT	INCLUDE	STRINGWIDTH	STRINGOF
EOF	EOLN	FILEPOS	**TICKCOUNT**

Chapter 14



Sorting

Sorting is the process of reordering data that have already been stored. While this can be done with both files and arrays, efficient sorting of files is beyond the scope of this book. However, reasonably sized files may be sorted by reading the contents into an array, sorting the array, and restoring the data to the file, so you are not totally out of luck where files are concerned.

The first sorting technique we will discuss is called the *bubblesort.* This is not a very efficient sorting method, but it has the virtue of being easy to understand. *Quicksort,* the second sort we will look at, is much faster and much more complex.

TOPICS COVERED IN THIS CHAPTER

- Using bubblesort with arrays
- Improving bubblesort when arrays are only slightly disordered
- Using quicksort
- Performance comparison of bubblesort and quicksort

BUBBLESORT

The principle of bubblesort is that consecutive elements of the array are compared and swapped if they are out of order. When this is done enough times and in a thorough way, all of the elements will wind up in their proper positions.

To illustrate, picture an array containing five integers, arranged as follows:

```
9  4  5  8  2
```

The sort proceeds from left to right, comparing each pair of elements. If a pair is out of order, the elements are swapped. First the program compares the first two elements, 9 and 4, which are underlined for clarity:

9 4 5 8 2

Since 9 is greater than 4, these numbers must be exchanged. After the swap, the array looks like this:

4 9 5 8 2

Next, the second and third elements are compared:

4 9 5 8 2

Again a swap is performed. Bubblesort proceeds to test the rest of the pairs. Here is the sequence of events that follows the comparison above. In each case, the two numbers being compared are underlined. If a swap is called for, it will be reflected in the next line.

4	5	9	8	2	*swap*
4	5	8	9	2	*swap*
4	5	8	2	9	*end of sorting pass*

This completes the first round of comparisons. Notice that one element, the 9, has been placed in its proper position. On each pass, Bubblesort will always place at least one number in its final position.

We cannot count on the other numbers, however. 8 started out in its correct location but was moved to a new location. The other elements tended to move left, toward their final locations. Bubblesort gets its name from this tendency for elements to move to their proper positions, much as bubbles drift up through a liquid.

With one pass completed, we start again at the left:

4	5	8	2	9	*no swap*
4	5	8	2	9	*no swap*
4	5	8	2	9	*swap*
4	5	2	8	9	*end of sorting pass*

The 8 and 9 were not compared this time, since the 9 was guaranteed to be in its proper position after the first sorting pass. Similarly, the 8 will be correctly positioned after the second pass. This means that only the first three elements need be considered in the next attempt:

4	5	2	8	9	*no swap*
4	5	2	8	9	*swap*
4	2	5	8	9	*end of sorting pass*

In the final pass, only one comparison must be made. After that comparison, the array will be sorted.

```
4   2   5   8   9   swap

2   4   5   8   9   end of pass, end of sort
```

Bubblesort has a very clear plan of attack, which is easy to program. Here is a bubblesort procedure. The array to be sorted is passed as a variable parameter to the array INT, which is simply an array of type Integer. The procedure expects the global variable SIZE to indicate the number of elements that are stored in the array:

```
procedure bubblesort (var int : intarray);
  var
    count, temp, top : integer;
begin
  for top := size downto 2 do
    for count := 1 to top - 1 do
      if int[count] > int[count + 1] then
        begin
          temp := int[count];
          int[count] := int[count + 1];
          int[count + 1] := temp
        end;
end;
```

The procedure consists of two FOR loops. The inner loop performs a comparison pass through the array. The outer loop repeats the inner loop until the sort is complete.

Let's examine the inner loop first. With COUNT equal to 1, this loop begins by determining if INT[1] is greater than INT[2]. If this is the case, the compound statement swaps the values of the two variables. A swap always needs a third, temporary variable and requires three steps. This is how a swap would be made between INT[1] and INT[2]:

1. Store INT[1] in the variable TEMP.
2. Copy the value of INT[2] to INT[1].
3. Store TEMP, the original value of INT[1], in INT[2].

After INT[1] and INT[2] have been compared, the loop compares INT[2] to INT[3]. Eventually, every pair in the array will be examined.

The inner loop assigns to COUNT the values of 1 through TOP-1. Therefore, the comparisons repeat until INT[TOP-1] has been compared to INT[TOP].

The value of TOP is taken from the outer loop. This loop counts down, starting with the value of SIZE. During the first pass through the array, TOP will point to the last element in the array. Recall that each pass is guaranteed to place one value in its final location. Since the first pass assigned INT[TOP] its final value, there is no need to examine that array element again. For this reason, the outer FOR loop decrements TOP by 1 for each pass through the array.

The outer loop counts down only to 2. When TOP has a value of 2, then the inner loop will count from 1 to TOP-1, that is, from 1 to 1. This means that the inner loop will execute only one time, com-

paring INT[1] to INT[2]. This is the last comparison that needs to be made. After this one, the loop has been sorted.

■ Here is a program that demonstrates bubblesort. Enter the program, inserting BUBBLESORT where indicated:

```
program SortDemo;
  const
    maxarray = 100;
  type
    intarray = array[1..maxarray] of integer;
  var
    integers : intarray;
    i : integer;
    size : 1..maxarray;
    time : longint;

  procedure showarray (integers : intarray);
    var
      i, j, k : integer;
  begin
    for i := 1 to (size div 10) do
      begin
        for j := 1 to 10 do
          begin
            k := (i - 1) * 10 + j;
            write(integers[k] : 6);
          end;
        writeln
      end;
      k := k + 1;
      while k <= maxarray do
        begin
          write(integers[k] : 6);
          k := k + 1
        end;
      writeln
  end;

{ Insert BUBBLESORT procedure here}

begin
  showtext;
  for i := 1 to maxarray do
    integers[i] := random;
```

```
  size := maxarray;
  showarray(integers);
  time := tickcount;
  bubblesort(integers);
  writeln('time required: ', (tickcount - time) / 60 : 6 : 2);
  showarray(integers);
end.
```

The main program starts by building an array containing 100 randomly selected integers. This array is displayed by SHOWARRAY before BUBBLESORT is called. When the sort is completed, the array is displayed again, along with the time required to sort it. SHOWARRAY is a nice procedure for displaying the data in arrays. In this configuration, it prints the data in rows of ten items each. The first nested loop prints the array items from 1 to MAXSIZE DIV 10. A final WHILE loop is required to print any remaining elements; if, for example, MAXSIZE is 105, 5 array elements will remain unprinted after execution of the nested IF loops.

■ Enlarge your Text window to the full width of the screen. Then run the program, paying attention to the length of time required for the sort. About 40 seconds will be needed.

The array that was sorted in this program started out in a highly disordered condition. Occasionally, we must work with arrays that are very nearly in sorted order. This might happen when a few array items were updated while most remained the same. How does bubblesort perform when an array is only slightly disordered?

■ Add these lines to the end of the main program:

```
integers[75] := 30000;
time := tickcount;
bubblesort(integers);
writeln('time required: ', (tickcount - time) / 60 : 6 : 2);
showarray(integers);
```

The first new line makes a single change in the sorted array. It is very unlikely that this changed item will be inserted in its expected position. The result is an array where the first 74 items remain in their correct positions. This array is sorted and then redisplayed.

■ Run the program. How long does the second sort take? You will probably find that it requires fully half of the time needed for the first sort. This is a lot of work simply to get one item into position.

Even though most of the array was still in order, BUBBLESORT insisted on making all of the comparisons that is made the first time the array was sorted. This is especially wasteful since 30000 was probably in its proper position after the first pass. The only time savings resulted because fewer swaps had to be performed. It would be nice to have a way to stop sorting if a pass is made without finding any required swaps. That would mean that the array was in order and that BUBBLESORT could cease.

A few changes will do the trick. This modification of BUBBLESORT uses the Boolean variable DONE to indicate whether or not any swaps were made in a pass. The outer FOR loop from the present version is replaced by a REPEAT loop:

```
procedure bubblesort (var int : intarray);
  var
    count, temp, top : integer;
```

```
    done : boolean;
begin
  top := size;
  repeat
    done := true;
    for count := 1 to top - 1 do
      if int[count] > int[count + 1] then
        begin
          temp := int[count];
          int[count] := int[count + 1];
          int[count + 1] := temp;
          done := false
        end;
    top := top - 1;
 until done or (top < 2);
end;
```

Before each comparison pass, DONE is assigned the value TRUE. Any exchange within the inner loop will change the value of DONE to FALSE. If no exchanges are made, DONE will remain TRUE and the REPEAT..UNTIL loop will terminate.

■ Install this new version of BUBBLESORT and try the program again. This time, the second sort of the array should require only one or two seconds, quite an improvement.

If you are very patient, try the following steps to sort a very large array with BUBBLESORT.

■ Remove the last five lines from the end of the main program. There is no need to try resorting the array in this experiment.

■ Change the value of MAXARRAY to 1000 and run the program.

■ Go to lunch. The sort will take over an hour to complete. You can see why we need a faster sort on occasion, even though it is harder to understand.

QUICKSORT

Quicksort uses a different approach to sorting. In this sort, the data in the array are repeatedly divided into groups that fall above and below some middle value. These groups are themselves divided, and the resulting groups are divided again. This process continues until each group contains just one member, at which point the array is sorted.

Here is a sample set of integers to be sorted with Quicksort:

```
289365471
```

To begin the sort, a middle value must be selected around which to divide the data. Ideally, this value should be the median of all of the values, but it turns out that it is too time consuming to calculate the median each time it is needed. Instead, the general practice is to use whatever value falls in the middle position of the array.

As we will see, this approach can result in the selection of some very inefficient dividing points. By and large, however, the acceptable choices outweigh the bad.

There is a rationale for choosing the middle member of the array as the dividing point. If the array is in some degree of order, the middle number is likely to be near to the median value for the

array. For arrays that are only slightly disordered, this will result in a more efficient search.

The index for the middle value can be calculated by summing the index values for the first and last items in the group to be sorted. This sum is divided by 2, using DIV, to yield the midpoint. In the present case, the bottom index is 1 and the top index is 9. Therefore, the index to the dividing point is calculated as (1 + 9) DIV 2. The resulting index is 5, so the value of the dividing point in this case is 6.

The next step is to divide or *partition* the data into two groups: data less than the value of the dividing point and data greater than or equal to it. Such a division on the example array will result in the following arrangement, where the gap between the numbers indicates a partition between the two groups:

```
2 1 4 3 5   6 9 7 8
```

This process must be repeated with the subgroups. In the first group, the dividing point would be 4 and this configuration results:

```
2 1 3   4 5   6 9 7 8
```

When dividing the "6 9 7 8" group, the dividing value is 9. Earlier we noted that simply taking the middlemost element of the group would result in our accepting some rather poor choices. In this situation, 9 is such a choice, since it only allows us to partition one value off from the group. The data are now arranged like this:

```
2 1 3   4 5   6 8 7   9
```

Here are the rest of the partitions that are performed to completely sort the list:

```
2 1 3   4 5   6 8 7   9

1   2 3   4 5   6 8 7   9

1   2   3   4   5   6 7   8   9

1   2   3   4   5   6   7   8   9
```

Now each group has been reduced to having only one member, and the sort is complete.

Here are the tasks that must be accomplished by Quicksort:

- Selecting the index for the dividing value
- Partitioning the data into two groups, one less than the dividing point and the other greater than or equal to it
- Repeating the Quicksort process on each of the resulting subgroups.

We have already seen how a dividing value may be calculated. We will require variables to keep track of the array index values for first and last elements of the group that is currently being examined. These variables will be called FIRST and LAST. If the array is named INTARRAY, the dividing value is determined by the expression: INTARRAY[FIRST + LAST) DIV 2]. This value will be assigned to the variable DIVIDER.

Here is the sample array. The values pointed to by the index variables FIRST, LAST, and DIVIDER are indicated as F, L, and D:

```
2 8 9 3 6 5 4 7 1
F       D       L
```

Next the array must be partitioned. This requires that we create two new index variables to point to the array data; P1 and P2. Initially, P1 will index the first element in the array and P2 will index the last. The array can be pictured like this:

```
 2 8 9 3 6 5 4 7 1
P1                 P2
 F         D        L
```

The first step in the partitioning process is to move P1 right until it encounters a value that belongs to the right of the dividing value:

```
2 8 9 3 6 5 4 7 1
  P1               P2
F         D        L
```

Next, we move P2 left until it points to a value that is less than the divider. Since P2 is already pointing to 1, it need not be moved.

The values pointed to by P1 and P2 are now exchanged:

```
2 1 9 3 6 5 4 7 8
  P1               P2
F         D        L
```

Again, P1 is moved right and P2 is moved left until each points to a value to be exchanged:

```
2 1 9 3 6 5 4 7 8
    P1       P2
F         D        L
```

Following the exchange the array looks like this:

```
2 1 4 3 6 5 9 7 8
    P1       P2

F         D        L
```

Again the pointers are moved

```
2 1 4 3 6 5 9 7 8
        P1 P2
F         D        L
```

and the values they find are swapped:

```
2 1 4 3 5 6 9 7 8
        P1 P2
F         D        L
```

When the pointers are moved again, we find that they have crossed. Since P1 is greater than P2, we know that this pass is completed. The subgroups have now been partitioned.

```
2 1 4 3 5   6 9 7 8
        P2  P1
F             D     L
```

Here are the statements that execute a single pass through the data:

```
repeat
  while intarray[p1] < divider do
    p1 := p1 + 1;
  while intarray[p2] > divider do
  p2 := p2 - 1;
  if p1 <= p2 then
    begin
      swap(intarray[p1], intarray[p2])
      p1 := p1 + 1;
      p2 := p2 - 1;
    end
until p1 > p2;
```

The values of P1, P2, and DIVIDER will be established before these statements are executed. This portion of the procedure performs three functions:

1. The first WHILE loop increments P1 until it indexes an element that belongs in the second part of the array.
2. The second WHILE loop decrements P2 until it indexes an element that belongs in the first part of the array.
3. Provided that the value of P1 is not greater than P2, the values in the selected array elements are swapped, using a procedure SWAP, which will be defined later.

After a swap, P1 is incremented and P2 is decremented, preparing the procedure to search for more data to be swapped. The REPEAT loop is terminated when P1 and P2 cross, indicating that no more swaps need to be made with this particular dividing value.

Next, a strategy must be found for sorting the two subarrays that have been created. The first subarray consists of the elements from INTARRAY[FIRST] through INTARRAY[P2]. The second subarray consists of the elements from INTARRAY[P1] through INTARRAY[LAST].

It turns out that this second subarray is not difficult to process. All we need do is assign the value of P1 to FIRST and repeat the sort.

However, it is not so easy to repeat the sort with the first subarray. The thing that is needed is a way to remember the portions of the array that still require work. The tool we will use is called a *stack.*

A stack behaves exactly like a paper spindle in an office. When using a paper spindle, we remember something by writing a note and sticking it on the top of the spindle. We can add as many notes as we like. Later when we need to retrieve something that was stored, we can remove the top piece of paper from the spindle. It is important to observe that the last note placed on the spindle is always the first one taken off.

A stack can be implemented using two things: an array in which to store data and a pointer variable, which points to the last item added to the stack. We start with an empty array and a pointer variable which has a value of 0. In fact, a pointer value of 0 is used to indicate that the stack is empty.

Storing an item in the stack is known as "pushing the data onto the stack." A push requires two actions: incrementing the pointer variable and storing the data into the array element that is pointed to.

Here is a procedure that pushes an integer onto the array STACK. STACKPOINTER is a global variable that indexes the stack array.

```
procedure push (int : integer);
begin
  stackpointer := stackpointer + 1;
  stack[stackpointer] := int
end;
```

The action of retrieving data from the stack is called "pulling". A pulling procedure is also needed:

```
procedure pull (var int : integer);
begin
  int := stack[stackpointer];
  stackpointer := stackpointer - 1
end;
```

Now we have a tool for storing the indexes for subarrays that have not yet been sorted. When such a subarray is identified, its beginning and ending indexes are pushed onto the stack. Later, they may be retrieved in the proper order for continued sorting. This process will soon be examined in detail.

We can pull the pieces together now. Here is the complete QUICKSORT procedure:

```
procedure quicksort (var interray : interray);
  var
    p1, p2, divider, first, last, stackpointer : integer;
    stack : array[1..50] of integer;
procedure push (int : integer);
begin
  stackpointer := stackpointer + 1;
  stack[stackpointer] := int
end;
procedure pull (var int : integer);
begin
  int := stack[stackpointer];
  stackpointer := stackpointer - 1
end;
procedure swap (var i, j : integer);
  var
    temp : integer;
begin
  temp := j;
  j := i;
  i := temp
end;

begin
  stackpointer := 0;
  push(1);
  push(size);
```

```
    repeat                                    { outer repeat loop }
      pull(last);
      pull(first);
      p1 := first;
      repeat                                  { middle repeat loop }
        p2 := last;
        divider := intarray[(first + last) div 2];
        repeat                                { inner repeat loop }
          while intarray[p1] < divider do
            p1 := p1 + 1;
          while intarray[p2] > divider do
            p2 := p2 - 1;
          if p1 <= p2 then
          begin
            swap(intarray[p1], intarray[p2]);
            p1 := p1 + 1;
            p2 := p2 - 1;
          end
        until p1 > p2;                        { end of inner loop }
        if first < p2 then
        begin
          push(first);
          push(p2);
        end;
        first := p1
      until first >= last                     { end of middle loop }
    until stackpointer = 0                    { end of outer loop }
  end;
```

QUICKSORT contains three local procedures: PUSH, PULL, and SWAP. Since these procedures must always be present for QUICKSORT to work, they were included within its structure. In this way, the entire procedure can be easily copied into other programs. Earlier I said that procedures could contain just about any feature that a program could contain. Here we see that procedures can contain their own procedure, and by implication, their own functions. These procedures and functions, just like the variables created within the procedure, are limited in scope and are not available globally.

The first action in the main part of the procedure is to initialize the stack. First 1 and then SIZE are pushed onto the stack. When these values are pulled, the complete array will be sorted.

The procedure contains three nested REPEAT loops:

1. The outermost one ensures that every entry pushed onto the stack will be pulled.
2. The next loop repeats the passes through the loop, pushing values of unsorted subarrays onto the stack.
3. The innermost loop makes a single partition pass through one section of the array.

This gets a bit complicated. Let's examine the procedure as it sorts the sample data we used earlier.

Here are the key variables and their values, along with the contents of the stack:

```
STACK:                  INTARRAY:       2 8 9 3 6 5 4 7 1

      >9<               P1 = undefined  DIVIDER = undefined
       1                P2 = undefined  STACKPOINTER = 2
                        FIRST = 1       LAST = 9
```

The STACK array is depicted vertically, in keeping with the stack metaphor. STACKPOINTER currently has a value of 2, pointing to the second element of the stack, which is 9. The item that is currently being pointed to by STACKPOINTER is enclosed like this: >9<. The first element of the stack is 1.

INTARRAY is the array to be sorted. The data in INTARRAY are depicted horizontally, as has been our practice in this chapter.

To begin, the starting values are pulled to start the sort. Then the values of P1, P2, and DIVIDER are calculated. Now the variables have the following values. The array values that are currently pointed to by P1 and P2 are indicated.

```
STACK:                  INTARRAY:       2 8 9 3 6 5 4 7 1
                                        P1               P2
        9               P1 = 1          DIVIDER = 6
        1               P2 = 9          STACKPOINTER = 0
       > <              FIRST = 1       LAST = 9
```

P2 obtained its value from the top value in the stack and P1 was assigned the second value. DIVIDER has been calculated to have a value of 6.

Numbers are not erased from the stack when they are pulled. They are simply ignored when the stack pointer is below them. Right now, STACKPOINTER has a value of 0 and the stack is considered to be empty. Since the stack pointer does not point to an element of the array, the pointer is shown as empty like this, > < .

Control now passes to the innermost REPEAT loop. The first WHILE statement at the beginning of this loop acts to point P1 to an array value that belongs in the second half of the array. Then the second WHILE statement points P2 to a value that belongs in the first half of the array. The variables now have these values:

```
STACK:                  INTARRAY:       2 8 9 3 6 5 4 7 1
                                          P1             P2
        9               P1 = 2          DIVIDER = 6
        1               P2 = 9          STACKPOINTER = 0
       > <              FIRST = 1       LAST = 9
```

After this, the array values are swapped. Then P1 is incremented, and P2 is decremented. Here are the values of the variables at the bottom of the inner loop:

```
STACK:                  INTARRAY:       2 1 9 3 6 5 4 7 8
                                          P1           P2
        9               P1 = 3          DIVIDER = 6
        1               P2 = 8          STACKPOINTER = 0
       > <              FIRST = 1       LAST = 9
```

The inner REPEAT loop executes again, selecting another pair of values for exchange. After the exchange is complete, this is the state of the variables:

```
STACK:          INTARRAY:       2 1 4 3 6 5 9 7 8
                                    P1         P2
    9           P1 = 3          DIVIDER = 6
    1           P2 = 7          STACKPOINTER = 0
  > <           FIRST = 1       LAST = 9
```

One more exchange can be made, after which the variables look like this:

```
STACK:          INTARRAY:       2 1 4 3 5 6 9 7 8
                                        P2 P1
    9           P1 = 6          DIVIDER = 6
    1           P2 = 5          STACKPOINTER = 0
  > <           FIRST = 1       LAST = 9
```

At this point, two important things have taken place. The partition is complete: all of the array values to the left of the 6 are less than 6, and all of the values to the right of 6 are greater than 6.

The other important event is that P1 is now greater than P2. This is the signal for the inner REPEAT loop to terminate. Since FIRST is less than P2, both FIRST and P2 are pushed onto the stack. In this way, the procedure remembers that the array from INTARRAY[FIRST] to INTARRAY[P2] remains unsorted.

After the values are pushed, FIRST is assigned the value of P1. Right now, the variables have these values:

```
STACK:          INTARRAY:       2 1 4 3 5 6 9 7 8
                                        P2 P1
   >5<          P1 = 6          DIVIDER = 6
    1           P2 = 5          STACKPOINTER = 2
                FIRST = 6       LAST = 9
```

The procedure has now reached the bottom of the second REPEAT loop. Since FIRST is still less than LAST, this loop cannot terminate. We return to the top of this middle loop, this time to partition INTARRAY[6] through INTARRAY[9]. After the variables are set up for the new pass, things look like this:

```
STACK:          INTARRAY:       2 1 4 3 5 6 9 7 8
                                      P1       P2
   >5<          P1 = 6          DIVIDER = 9
    1           P2 = 9          STACKPOINTER = 2
                FIRST = 6       LAST = 9
```

Another partition pass is made. This time, however, only the last four array elements are examined. The dividing value now is 9. After the first exchange, the values of the variables are:

```
STACK:          INTARRAY:       2 1 4 3 5 6 8 7 9
                                            P1
                                            P2
   >5<          P1 = 8          DIVIDER = 9
    1           P2 = 8          STACKPOINTER = 2
                FIRST = 6       LAST = 9
```

P1 and P2 now point to the same array element. Not surprisingly, therefore, the next pass through the loop cannot find any values to exchange. Pay careful attention to the two WHILE loops. In the first, P1 is incremented until it points to the last element in the array. However, since INTARRAY[P2] is already less than DIVIDER, P2 is not decremented and retains the value of 8. Since P1 was in-

cremented, P1 and P2 cross, ending the inner loop, and resulting in the following condition:

```
STACK:                INTARRAY:      2 1 4 3 5 6 8 7 9
                                                 P2 P1
        >5<           P1 = 9         DIVIDER = 9
         1            P2 = 8         STACKPOINTER = 2
                      FIRST = 6      LAST = 9
```

The inner loop terminates and the middle loop takes over. Since FIRST is less than P2, FIRST and P2 are pushed onto the stack. FIRST is then assigned the value of P1. Now the variable have these values:

```
STACK: >8<            INTARRAY:      2 1 4 3 5 6 8 7 9
        6                                        P2 P1
        5             P1 = 9         DIVIDER = 9
        1             P2 = 8         STACKPOINTER = 4
                      FIRST = 9      LAST = 9
```

At this point, notice that FIRST and LAST share the same value, indicating that only one value remains in the sublist, and that no more sorting of the sublist is needed. Since FIRST = LAST, the middle REPEAT loop terminates.

Control now passes to the outer REPEAT loop. Since STACKPOINTER in not 0, this loop executes again.

The first thing done at the top of this loop is to pull two values from the stack. LAST receives the value 8, and FIRST receives the value 6. These values correspond to a portion of the array that is not yet completely sorted. Here are the variable values after P1 is assigned the value of FIRST:

```
STACK:  8             INTARRAY:      2 1 4 3 5 6 8 7 9
        6                                    P1   P2
        >5<           P1 = 6         DIVIDER = 9
         1            P2 = 8         STACKPOINTER = 2
                      FIRST = 6      LAST = 8
```

The procedure is now preparing to sort the array from INTARRAY[6] through INTARRAY[8]. After P2 is assigned the value of LAST and DIVIDER is calculated, the procedure is ready to make another partition pass:

```
STACK:  8             INTARRAY:      2 1 4 3 5 6 8 7 9
        6                                    P1   P2
        >5<           P1 = 6         DIVIDER = 8
         1            P2 = 8         STACKPOINTER = 2
                      FIRST = 6      LAST = 8
```

Here is the condition after this pass is made:

```
STACK:  8             INTARRAY:      2 1 4 3 5 6 7 8 9
        6                                      P2 P1
        >5<           P1 = 8         DIVIDER = 8
         1            P2 = 7         STACKPOINTER = 2
                      FIRST = 6      LAST = 8
```

Since P1 now exceeds P2, the inner loop ends, returning control to the middle loop. This loop pushes FIRST and P2 to the stack and sets FIRST equal to P1 with this result:

```
STACK:  >7<          INTARRAY:      2 1 4 3 5 6 7 8 9
          6                                   P2 P1
          5          P1 = 8         DIVIDER = 8
          1          P2 = 7         STACKPOINTER = 4
                     FIRST = 8      LAST = 8
```

Since FIRST is equal to LAST, the middle loop terminates.

The STACKPOINTER has a value greater than 0, and the outer loop repeats. The top two values are pulled, and the process continues. Here is the condition before the middle loop executes again:

```
STACK:   7           INTARRAY:      2 1 4 3 5 6 7 8 9
           6                                 P1 P2
         >5<         P1 = 6         DIVIDER = 6
           1         P2 = 7         STACKPOINTER = 2
                     FIRST = 6      LAST = 7
```

Only two values are contained in this portion of the array, and they are already in order. P1 will not be incremented since INTARRAY[6] is not less than DIVIDER. However, P2 is decremented, with the result that P1 becomes equal to P2.

This causes an interesting swap to be made. P1 and P2 both point to the same variable. Therefore no essential change is made. The important thing is that P1 is incremented and P2 is decremented, causing P1 to become greater than P2. Now the inner loop terminates. Here are the variable values at the end of the inner loop.

```
STACK:   7           INTARRAY:      2 1 4 3 5 6 7 8 9
           6                               P2   P1
         >5<         P1 = 7         DIVIDER = 6
           1         P2 = 5         STACKPOINTER = 2
                     FIRST = 6      LAST = 7
```

Control now lies with the middle loop. Since FIRST is greater than P2, no values are pushed onto the stack. The array is now sorted from INTARRAY[6] through INTARRAY[9].

Next, FIRST is assigned the value of P1, which is 7. FIRST and LAST now have the same value, and the middle loop terminates. Control reverts to the outer loop.

Since the value of STACKPOINTER is greater than 0, the outer loop executes again. It starts by pulling the two remaining values from the stack, producing this condition:

```
STACK:   7           INTARRAY:      2 1 4 3 5 6 7 8 9
           6                               P2   P1
           5         P1 = 7         DIVIDER = 6
           1         P2 = 5         STACKPOINTER = 0
         > <         FIRST = 1      LAST = 5
```

The complete sorting process is ready to begin again, this time sorting array elements 1 through 5. I will not explain the rest of the sort, since the process is the same for this part of the array as it was when the last four array elements were sorted.

There is one difference: when this half of the array is completed, STACKPOINTER will have a value of 0 at the end of the outer loop. This is the signal that the sort is completed and that the procedure can terminate.

The process we have just been through is called *hand simulation.* There is nothing the computer can do that we cannot do with a pencil and a piece of paper, and I have just simulated the operations of the QUICKSORT procedure for you. Hand simulation is useful when a complex program must be

"reverse engineered," so that we can figure out how it works. Hand simulation is also a powerful, if tedious, debugging tool. If you have tried without success to get a program to work, try hand simulation.

The QUICKSORT procedure may be substituted for BUBBLESORT in the demonstration program. Try the following experiments:

- Replace BUBBLESORT with QUICKSORT. Be sure to change the procedure call in the main program so that it calls QUICKSORT instead of BUBBLESORT.
- With MAXSIZE set to a value of 100, how long does quicksort require to sort the array? Is this a significant improvement?
- Change MAXSIZE to 1000 and run the program again. Is quicksort faster than bubblesort?
- Change MAXSIZE back to 100. Install these lines at the end of the main program:

```
integers[75] := 30000;
time := tickcount;
quicksort(integers);
writeln('time required: ', (tickcount - time) / 60 : 6 : 2);
showarray(integers);
```

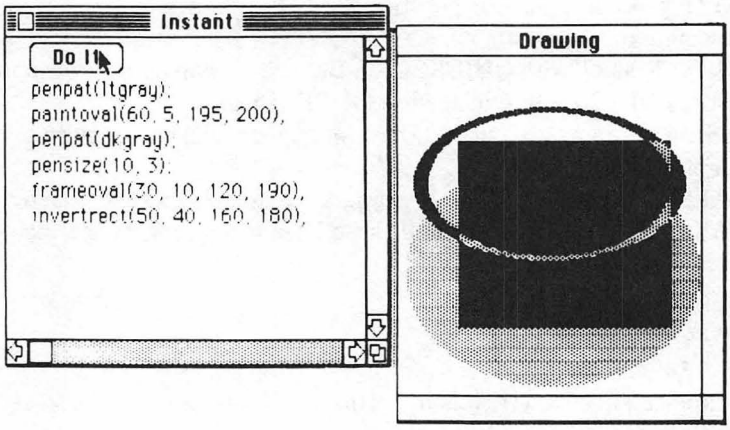
- Run the demonstration again. Is quicksort a good sort to use when only a few items are out of order? Interestingly, quicksort does not do as well as the second version of bubblesort. Quicksort has no way of shutting itself off when the array is in order.

If you have understood the explanations of quicksort, you are now a master of one of the most powerful sorts available. If you found the going a bit heavy, don't worry. You can use the sort without fully understanding it. Just remember that variable types will have to be adjusted to accommodate different array types. You will need to change the type and perhaps the name of the INTARRAY parameter as well as the type of TEMP in the SWAP procedure.

YOUR PASCAL VOCABULARY

This chapter was tough enough without adding any new vocabulary items. Take a break.

Chapter 15



An Improved Inventory Program

To pull the previous lessons together, this concluding chapter will present a more complete inventory program. There will be very little in the program that is new to you, but that in itself contains an important bit of knowledge. You now know enough to build some interesting and fairly complex programs.

Since not much in this chapter is new, discussion will not be very extensive. If you have mastered the first fourteen chapters, you will not require a great deal of explanation. However, you will encounter one or two new things, such as a strategy for deleting items from files.

THE INVENTORY PROGRAM

The program is quite long. In fact, it could not be much longer and still be able to fit on a 128K Macintosh. Because of its size, I will print it only once. You will probably wish to enter the entire program before reading the discussion that follows.

```
program inventory (input, output);
  type
    item_data = record
          name : string[20];
          quantity : integer;
          price : real;
        end;
    comstring = string[40];
    stockfile = file of item_data;
```

```
var
  item : item_data;
  stock : stockfile;
  com : char;
  size : longint;

procedure fulltext;
  var
    r : rect;
begin
  with r do
    begin
      top := 50;
      left := 0;
      bottom := 325;
      right := 510;
    end;
  settextrect(r);
  showtext
end;

function realinput : real;
  var
    s : string[10];
    r : real;
    i : integer;
    ok : boolean;
begin
  repeat
    readln(s);
    s := concat('0', s);
    ok := true;
    for i := 1 to length(s) do
      if pos(copy(s, i, 1), '.0123456789') = 0 then
        ok := false;
    if not ok then
      writeln('Only digits and periods allowed. Please reenter.');
  until ok;
  readstring(s, r);
  realinput := r
end;  { of realinput }

procedure displayitem (item : item_data);
  var
```

```
    len : integer;
begin
  with item do
    begin
      len := length(name) + 2;
      writeln('Item name:' : 12, name : len);
      writeln('On hand:' : 12, quantity : len);
      writeln('Item price:' : 12, price : len : 2);
  end; { with }
  writeln
end;

function lookup (s : item_data;
                 var lookfile : stockfile) : longint;
  var
    midpoint, bottom, top : longint;
begin
  bottom := 1;
  reset(lookfile);
  top := size;
  repeat
    midpoint := (bottom + top) div 2;
    seek(lookfile, midpoint);
    if s.name < lookfile^.name then
      top := midpoint - 1;
    if s.name > lookfile^.name then
      bottom := midpoint + 1;
  until (s.name = lookfile^.name) or (bottom > top);
  if bottom <= top then
    lookup := midpoint
  else
    lookup := -1
end;   { of lookup }

procedure insert (item : item_data;
                  var infile : stockfile);
  var
    temp : item_data;
    i, location : longint;
begin
  temp.name := '';
  while (filepos(infile) <= size) and (temp.name < item.name) do
    read(infile, temp);
```

```
    if temp.name >= item.name then
      begin
        location := filepos(infile) - 1;
        for i := size downto location do
          begin
            seek(infile, i);
            read(infile, temp);
            write(infile, temp)
          end;
        seek(infile, location);
      end;
    write(infile, item);
    size := size + 1;
    temp.quantity := size;
    seek(infile, 0);
    write(infile, temp)
  end;    { of insert }

  procedure add;
    var
      item : item_data;
  begin
    write('(Press RETURN to cancel add)  Name of item: ');
    readln(item.name);
    if length(item.name) > 0 then
      if (lookup(item, stock) >= 0) then
        writeln('That item is already in inventory.')
      else
        begin
          write('Price of item: ');
          item.price := realinput;
          write('Quantity on hand: ');
          readln(item.quantity);
          insert(item, stock);
          writeln('New item added to inventory.');
          writeln
        end  { of else }
  end;       { of add }

  procedure buy;
    var
      item : item_data;
      qty : integer;
```

```
    recnum : longint;
begin
  write('(Press RETURN to cancel buy) Item to buy: ');
  readln(item.name);
  if length(item.name) > 0 then
    begin
      recnum := lookup(item, stock);
      if recnum < 0 then
        writeln('That item is not in inventory.')
      else
        begin
          write('Quantity to buy: ');
          readln(qty);
          seek(stock, recnum);
          read(stock, item);
          item.quantity := item.quantity + qty;
          seek(stock, recnum);
          write(stock, item);
        end  { of else }
    end    { of if }
end;      { of buy }

procedure delete;
  var
    ch : char;
    item : item_data;
    i, recnum : longint;
  begin
    write('(Press RETURN to cancel deletion)  ');
    write('Name of item to delete: ');
    readln(item.name);
    if length(item.name) > 0 then
      begin   { of if #1 }
        recnum := lookup(item, stock);
        if recnum < 0 then
          writeln('No entry was found for that item.')
        else
          begin
            seek(stock, recnum);
            read(stock, item);
            displayitem(item);
            writeln('Do you wish to delete this item? y or n');
            read(ch);
```

```
                if ch in ['y', 'Y'] then
                begin   { of if #2 }
                  size := size - 1;
                  seek(stock, 0);
                  item.quantity := size;
                  write(stock, item);
                  for i := recnum to size do
                  begin
                    seek(stock, i + 1);
                    read(stock, item);
                    seek(stock, i);
                    write(stock, item);
                  end  { of for}
                end  { of if #2  }
              end { of else }
    end { of if #1}
end;  { of delete  }

procedure find;
  var
    item : item_data;
    recnum : longint;
begin
  write('(Press RETURN to cancel find)  Name of item to find: ');
  readln(item.name);
  if length(item.name) > 0 then
    begin
      recnum := lookup(item, stock);
      if recnum < 0 then
        writeln('No entry was found for that item.')
      else
        begin
          seek(stock, recnum);
          read(stock, item);
          displayitem(item)
        end
    end
end;  { of find  }

procedure reprice;
  var
    item : item_data;
    recnum : longint;
```

```
  begin
    write('(Press RETURN to cancel reprice) Item to reprice: ');
    readln(item.name);
    if length(item.name) > 0 then
      begin
        recnum := lookup(item, stock);
        if recnum < 0 then
          writeln('That item is not in inventory.')
        else
          begin
            seek(stock, recnum);
            read(stock, item);
            writeln('Current price: ', item.price : 3 : 2);
            write('New price: ');
            item.price := realinput;
            seek(stock, recnum);
            write(stock, item)
          end { of else }
      end   { of if }
end;    { of reprice }

procedure sell;
  var
    item : item_data;
    qty : integer;
    recnum : longint;
begin
  write('(Press RETURN to cancel sell) Item to sell: ');
  readln(item.name);
  if length(item.name) > 0 then
    begin
      recnum := lookup(item, stock);
    if recnum < 0 then
      writeln('That item is not in inventory.')
    else
      begin
        seek(stock, recnum);
        read(stock, item);
        writeln('You have ', item.quantity : 1, ' on hand.');
        write('Quantity to sell: ');
        readln(qty);
        if qty > item.quantity then
          writeln('You don''t have that many . ')
```

```
        else
          begin
            item.quantity := item.quantity - qty;
            seek(stock, recnum);
            write(stock, item)
          end; { of else}
      end    { of else }
    end      { of if }
end;         { of sell  }

procedure totals;
  var
    item : item_data;
    total : real;
    i : longint;
begin
  total := 0;
  write('Item:' : 20, 'Price:' : 10);
  writeln('Quantity:' : 15, '$ in inventory:' : 20);
  write('-----' : 20, '------' : 10);
  writeln('---------' : 15, '---------------' : 20);
  seek(stock, 1);
  for i := 1 to size do
    begin
      read(stock, item);
      with item do
        begin
          write(name : 20, price : 10 : 2);
          writeln(quantity : 15, price * quantity : 20 : 2);
          total := total + price * quantity
        end; { of with }
    end; { of for }
  writeln(' ' : 45, '---------------' : 20);
  writeln('Total value of inventory: ' : 45, total : 20 : 2)
end; { of totals }

procedure pack;
  var
    item : item_data;
    backupfile : file of item_data;
    i : longint;
begin
  writeln('Packing the file');
```

```
    rewrite(backupfile, 'Backup Stockfile');
    reset(stock);
    for i := 0 to size do
      begin
        read(stock, item);
        write(backupfile, item)
      end;
    reset(backupfile);
    rewrite(stock);
    while not eof(backupfile) do
      begin
        read(backupfile, item);
        write(stock, item)
      end; { of while }
  end; { of pack }

begin   {main program}
  hideall;
  fulltext;
{REMOVE COMMENT BRACKETS FROM NEXT LINE AFTER DEBUGGING}
{hidecursor;}
{REMOVE COMMENT BRACKETS FROM NEXT TWO STATEMENTS}
{IF PROGRAM SHOULD CLEAR THE STOCK FILE ON STARTUP  }
{rewrite(stock, 'Stockfile');}
{close(stock);}
  open(stock, 'Stockfile');
  seek(stock, maxlongint);
  if filepos(stock) <= 1 then
    begin
      item.quantity := 0;
      rewrite(stock);
      write(stock, item)
    end
  else
    begin
      seek(stock, 0);
      read(stock, item);
      size := item.quantity
    end;
  repeat
    writeln;
    writeln('(Add, Buy, Delete, Find, Reprice, Sell, Totals, Quit)');
    write('Enter Command: ');
```

```
        read(com);
        rewrite(output);
        case com of
          'a', 'A' :
            add;
          'b', 'B' :
            buy;
          'd', 'D' :
            delete;
          'f', 'F' :
            find;
          'r', 'R' :
            reprice;
          's', 'S' :
            sell;
          't', 'T' :
            totals;
          'q', 'Q' :
            ;
          otherwise
            ;
        end { of case }
      until com in ['q', 'Q'];
      pack;
      rewrite(output);
      showcursor
    end.
```

If you have entered the program, you will find it easy to refer to the parts of the program in your MacPascal program window as you continue with the chapter. You may find it helpful to run the program, trying a few of the options that are available. The screen prompts should be fairly self-explanatory.

THE DECLARATION PART

The inventory program is oriented around the record type ITEM_DATA, which contains three fields. The file type STOCKFILE is declared to be a FILE OF ITEM_DATA. The other variables and types will be explained as they are used.

An important variable is SIZE. It will be used throughout the program to keep track of the number of records that are stored in the stock file. Although it is doubtful that the file would ever grow that large, SIZE and all file indexing variables will be of type LONGINT.

THE MAIN PROGRAM

Several setup actions are performed at the beginning of the main program. The first is to use HIDEALL to conceal all of the Pascal windows. This program is so large that these windows must

be concealed to prevent the program from running out of memory on a 128K Mac. When the program ends, you will have to restore the Program window manually.

Next the program calls FULLTEXT, a procedure that enlarges the Text window to fill the screen.

HIDECURSOR conceals the mouse cursor. This, incidentally, can be a dangerous thing to do until a program is thoroughly debugged. If the program enters an infinite loop, the only way to escape is to select Halt from the Pause menu. This is rather difficult to do if you cannot see the mouse cursor. For this reason, the procedure call is set off in comment brackets. When you are sure that the program is operating properly, remove the brackets to activate the procedure call.

The next two optional statements use REWRITE to clear the contents of the stock file, which is then closed. When these statements are active, the stockfile will be cleared every time the program is run. Normally, however, we want the file to retain the inventory contents from one session to the next. If the comment brackets are left around these statements, the program will start out with the same inventory information that it had at the end of the last session.

The stock file is opened next. The first entry in the file will be used to store the file size. The structure of ITEM____DATA includes an integer field named QUANTITY. This field will be used to store the size. After the file is opened, the program checks to see if FILEPOS(STOCK) <= 1. If it is, then the file may be new, and it is necessary to store a file size entry in position 0 of the file.

SIZE is assigned the value of ITEM.QUANTITY which is the current size of the file.

The bulk of the main program consists of a fairly large CASE statement. Again we see the ease with which a large program may be planned through use of a CASE statement. Before writing a single procedure, I could plan all of the options that would be available along with the names of the procedures that would execute the options.

If the "Quit" option is selected, the program exits the REPEAT loop. The concluding statements include a call to a PACK procedure. This procedure is part of the deletion method that is implemented in the program. More on this later.

THE FUNCTIONS AND PROCEDURES

Now for an item-by-item tour of the program. Let's just start back at the top and work down.

The Function REALINPUT

This function addresses a problem that appears often in interactive programs: user input error. Since users often make typing errors or misunderstand instructions, it is a good idea to include special procedures as needed to help with data input. This procedure will accept only positive, real numbers.

When an illegal character is typed when reading to a real variable, the READ procedure terminates. Unfortunately, a user who intended to type "12.34" might accidentally type "12.3r", in which case, Pascal accepts only "12.3". Unless we do some checking, the program will go on, but it will be processing the wrong data.

REALINPUT accepts input into a string variable. READ will accept any characters in a string variable, stopping only when the Return key is pressed. If a user types "12.3r", he or she may simply hit the Backspace key and correct the error before pressing Return.

Once the string is entered, however, it must be checked to ensure that it contains only digits and periods. This is done with a loop which uses the POS function, examining each character to see if it is contained in the string. '.0123456789'. If an illegal character is found, a message is printed, and the user is asked to enter another number.

We should note that the procedure does not prevent the user from entering more than one period in the number.

Once it has been determined that the string contains only periods and digits, READSTRING is used to read a real number R from the string S. Recall that READSTRING works about like READ except that it reads from a string instead of from a text file. It is the value in R that is output by the function.

The Procedure DISPLAYITEM

The program will be displaying the contents of item records quite often. The DISPLAYITEM procedure makes this easy to do.

The Function LOOKUP

This is a variation on the LOOKUP function introduced in Chapter 13. It has been adapted to work with data of type ITEM_DATA. TOP determines the last component in the file that will be examined by this procedure. The value of TOP is determined by the value of the global variable SIZE. The deletion method we will use prevents us from using the EOF function to find the last file component. Otherwise the procedure is quite similar to the version developed in Chapter 13.

The Procedure INSERT

After all of my preaching about binary search and insertion techniques, you may wonder why this procedure uses a sequential search to find the point in the file at which the new data should be inserted. The explanation has to do with memory limitations. I started working on the program using a binary approach, but soon ran out of memory. Since an inefficient program is better than one that won't fit on the computer, a sequential search is not such a bad deal. We saw in Chapter 13 that insertions were not dramatically improved through use of a binary search strategy.

An additional difference in this version of the insertion procedure should be noted. In Chapter 13, the procedure started at the insertion point and worked toward the end of the file as it moved items up. In this version, the top elements are moved up, starting from the top end of the file and working downward to the location where the new item should be inserted. This is done with a simple FOR..DOWNTO loop. The benefit of this method is that only one SEEK must be performed for each item that is moved up. On a large file, this should result in some improvement in performance.

Since INSERT adds an item to the stock file, the file size must be incremented. This is done by the last five statements in the procedure. SIZE is incremented, and its value is stored in the first component of the stock file. By storing the new size every time it is changed, the size is preserved in case an error terminates the program abnormally. If the size were not saved each time, a program malfunction would cause the value of the SIZE variable to be lost. In this case, the size stored in the file might not agree with the actual number of items in the file.

The Procedure ADD

This procedure asks for the name of an item to be added to inventory. First, a lookup is performed. If the item name already appears in the inventory file, the new entry is rejected.

If this is a new item name, the QUANTITY and PRICE fields are input, and the record is inserted into the stock file.

Like most procedures in this program, ADD gives the user a way out if the request to add an

item was made by mistake. If the user enters an empty string by pressing Return without typing anything, the rest of the procedure will be ignored. Otherwise, the user would have no choice but to enter an unwanted item and then to delete it.

The Procedure BUY

This procedure locates an item in the stock file. This item is read from the file, modified, and stored back in the file. Much of the procedure is similar to ADD, requiring little further explanation.

The Procedure DELETE

After locating the data that the user has asked to delete, the procedure displays the record and requests a confirmation of the deletion. Unless the user types "y" or "Y", the deletion is aborted. When an action on the part of the user might cause an accidental loss of valuable information, it is generally good policy to confirm the user's choice.

The deletion is performed by moving each of the later file components down one position. Suppose a file contains the following items:

```
apple banana orange pear eof
```

If we would like to remove "banana" we simply move "orange" and "pear" down like this:

```
apple orange pear pear eof
```

Even though "pear" was copied down to the next component, the last component retains its old value. Unfortunately, there is no way to remove this component and to move the end-of-file marker down.

Now we can see the need for the SIZE variable. We cannot remove the last component, but we can tell the program to ignore it by decrementing SIZE. Procedures can then use SIZE instead of EOF to determine where the last active component in the file is.

When program execution is terminated with the Quit option, a procedure named PACK will be used to remove the file components that are no longer being used.

The Procedure FIND

FIND simply calls LOOKUP to determine if an item is in the stock file. If the item is found, DISPLAYITEM is used to show it.

The Procedures REPRICE and SELL

These procedures are almost identical to BUY.

The Procedure TOTALS

The procedure simply prints out a list of the items in inventory, along with some totals. Since the Text window will only display a few lines, you may wish to modify this procedure so that the output will be sent to a printer.

The Procedure PACK

This procedure is not really necessary, but it serves two nice purposes. First, it removes the dead

elements that deletions may have created at the end of a file. It also creates a backup file that contains the same contents as the stock file. This file can be used if the stock file is damaged. It is very important with business programs to create extra copies of data files so that vital data will not be lost if a storage disk is damaged.

First the procedure copies the active components in STOCKFILE into the file BACKUP STOCKFILE. Then, since the backup file does not contain the deleted components, the procedure clears STOCKFILE with REWRITE and copies BACKUP STOCKFILE back to STOCKFILE. When this is complete, the files will be identical, but the inactive components will have been removed.

If the original copy of STOCKFILE is damaged, simply change the name of BACKUP STOCKFILE to STOCKFILE, following the instructions in your Macintosh manual.

You may want to copy the backup copy to another storage disk. If you do that, the original disk could be destroyed, and you would still have a backup copy of the file on another disk. To do this, you must know the name that was assigned to the second storage disk when it was formatted. Read about formatting in the Macintosh manual.

Suppose that your second disk was named "ARCHIVES." To place BACKUP STOCKFILE on that disk, simply add ARCHIVE: to the file name. With this example, you would change the REWRITE statement to:

```
rewrite(backupfile, 'Archive:Backup Stockfile');
```

CONCLUSION

That's it. There was nothing too difficult in the program, was there? I hope you will study it carefully. Even better, dream up some new features that you can add. If you own a 512K Mac, you can really expand it. If you have a 128K Mac, you may have to remove some of the current procedures to make room. FIND is probably not a very important procedure since TOTALS is available.

When you are working with large programs that approach the memory capacity of your Mac, it is a good idea to save the program before you run it. Occasionally, something may happen that can cause Pascal to experience a fatal error, losing the program. If you have saved it to disk, you can always recover the most recent copy.

If you have gotten through the entire book, you can be proud of yourself. Almost everything was probably new, and I know some of it was difficult. Therefore, my congratulations. If you caught the programming bug, and if you can't wait to do something new, that's even better. At its best, programming can be as fun and as challenging as an activity can get.

Appendix

The Macintosh Pascal Character Set

Here is a chart containing the characters that may be printed in MacPascal. Each character is associated with a number, representing its ordinal position in the type Char. Therefore, to print the character Ω, write the expression CHR(189).

For the characters from 32 to 127, these numbers are equivalent to the ASCII codes for the characters. ASCII (the American Standard Code for Information Interchange) is a widely used standard that assigns numeric codes to characters. You will often see references to ASCII codes in reference books. Above 127, the standard does not apply, and the Macintosh designers were free to add characters of their own selection.

You will notice that the list begins at 32. The characters below 32 are not printable characters. Instead they are used in various ways to control computer operations. For example, these characters are often used to control printer functions. Since the required characters vary depending on the equipment, and since the names of these characters are somewhat esoteric, I have chosen to leave them out of the table. Consult your equipment manuals to see which ones apply.

The font used in the Text window is Monaco. The actual characters you get will depend on the font being used in some cases. You have no control over this in the Text window, but you can control the font when writing text in the Drawing window.

THE PROGRAM

Incidentally, here is the program that was used to produce the table:

```
program characters;
  const
    last = 218;
    first = 32;
    columns = 5;
```

```
  var
    i, j, x : integer;
begin
  showtext;
  x := ((last - first) div columns + 1);
  for i := first to first - 1 + x do
    begin
      for j := 0 to columns - 1 do
        if (i + x * j) <= last then
        write(i + x * j : 5, chr(i + x * j) : 3);
      writeln
    end
end.
```

PRINTABLE CHARACTERS IN MACINTOSH PASCAL

32		59	;	86	V	113	q	140	å	167	ß	194	¬		
33	!	60	<	87	W	114	r	141	ç	168	®	195	√		
34	"	61	=	88	X	115	s	142	é	169	©	196	ƒ		
35	#	62	>	89	Y	116	t	143	è	170	™	197	≈		
36	$	63	?	90	Z	117	u	144	ê	171	´	198	Δ		
37	%	64	@	91	[	118	v	145	ë	172	¨	199	«		
38	&	65	A	92	\	119	w	146	í	173	≠	200	»		
39	'	66	B	93	]	120	x	147	ì	174	Æ	201	...		
40	(	67	C	94	^	121	y	148	î	175	Ø	202			
41	)	68	D	95	_	122	z	149	ï	176	∞	203	À		
42	*	69	E	96	`	123	{	150	ñ	177	±	204	Ã		
43	+	70	F	97	a	124	|	151	ó	178	≤	205	Õ		
44	,	71	G	98	b	125	}	152	ò	179	≥	206	Œ		
45	-	72	H	99	c	126	~	153	ô	180	¥	207	œ		
46	.	73	I	100	d	127		154	ö	181	µ	208	-		
47	/	74	J	101	e	128	Ä	155	õ	182	∂	209	–		
48	0	75	K	102	f	129	Å	156	ú	183	Σ	210	"		
49	1	76	L	103	g	130	Ç	157	ù	184	Π	211	"		
50	2	77	M	104	h	131	É	158	û	185	π	212	'		
51	3	78	N	105	i	132	Ñ	159	ü	186	∫	213	'		
52	4	79	O	106	j	133	Ö	160	†	187	ª	214	÷		
53	5	80	P	107	k	134	Ü	161	°	188	º	215	◊		
54	6	81	Q	108	l	135	á	162	¢	189	Ω	216	ÿ		
55	7	82	R	109	m	136	à	163	£	190	æ				
56	8	83	S	110	n	137	â	164	§	191	ø				
57	9	84	T	111	o	138	ä	165	•	192	¿				
58	:	85	U	112	p	139	ã	166	¶	193	¡				

Index

Edited by Marilyn L. Johnson

MacPascal Programming

If you are intrigued with the possibilities of the programs included in *MacPascal Programming* (TAB Book No. 1891), you should definitely consider having the ready-to-run disk containing the software applications. This software is guaranteed free of manufacturer's defects. (If you have any problems, return the disk within 30 days, and we'll send you a new one.) Not only will you save the time and effort of typing the programs, the disk eliminates the possibility of errors that can prevent the programs from functioning. Interested?

Available on disk for the Macintosh with at least 128K and Macintosh Pascal at $19.95 for each disk plus $1.00 each shipping and handling. (Note that Macintosh Pascal must be purchased from your computer dealer.)

I'm interested. Send me:

_____ disk for Macintosh with at least 128K and Macintosh Pascal (6239S)

_____ TAB BOOKS catalog

_____ Check/Money Order enclosed for $19.95 plus $1.00 shipping and handling for each disk ordered.

_____ VISA _____ MasterCard

Account No. _____ Expires _____

Name _____

Address _____

City _____ State _____ Zip _____

Signature _____

Mail To: **TAB BOOKS INC.**
P.O. Box 40
Blue Ridge Summit, PA 17214

(Pa. add 6% sales tax. Orders outside U. S. must be prepaid with international money orders in U. S. dollars.)

TAB 1891

OTHER POPULAR TAB BOOKS OF INTEREST

The Computer Era—1985 Calendar Robotics and Artificial Intelligence (No. 8031—$6.95)

Using and Programming the IBM PCjr®, including 77 Ready-to-Run Programs (No. 1830—$11.50 paper; $16.95 hard)

Word Processing with Your ADAM™ (No. 1766—$9.25 paper; $15.95 hard)

The First Book of the IBM PCjr® (No. 1760—$9.95 paper; $14.95 hard)

Going On-Line with Your Micro (No. 1746—$12.50 paper; $17.95 hard)

Mastering Multiplan™ (No. 1743—$11.50 paper; $16.95 hard)

The Master Handbook of High-Level Microcomputer Languages (No. 1733—$15.50 paper; $21.95 hard)

Apple Logo for Kids (No. 1728—$11.50 paper; $16.95 hard)

Fundamentals of TI-99/4A Assembly Language (No. 1722—$11.50 paper; $16.95 hard)

The First Book of ADAM™ the Computer (No. 1720—$9.25 paper; $14.95 hard)

BASIC Basic Programs for the ADAM™ (No. 1716—$8.25 paper; $12.95 hard)

101 Programming Surprises & Tricks for Your Apple II®/II®e Computer (No. 1711—$11.50 paper)

Personal Money Management with Your Micro (No. 1709—$13.50 paper; $18.95 hard)

Computer Programs for the Kitchen (No. 1707—$13.50 paper; $18.95 hard)

Using and Programming the VIC-20®, including Ready-to-Run Programs (No. 1702—$10.25 paper; $15.95 hard)

25 Games for Your TRS-80™ Model 100 (No. 1698—$10.25 paper; $15.95 hard)

Apple® Lisa™: A User-Friendly Handbook (No. 1691—$16.95 paper; $24.95 hard)

TRS-80 Model 100—A User's Guide (No. 1651—$15.50 paper; $21.95 hard)

How To Create Your Own Computer Bulletin Board (No. 1633—$12.95 paper; $19.95 hard)

Using and Programming the Macintosh™, with 32 Ready-to-Run Programs (No. 1840—$12.50 paper; $16.95 hard)

Programming with dBASE II® (No. 1776—$16.50 paper; $26.95 hard)

Making CP/M-80® Work for You (No. 1764—$9.25 paper; $16.95 hard)

Lotus 1-2-3™ Simplified (No. 1748—$10.25 paper; $15.95 hard)

The Last Word on the TI-99/4A (No. 1745—$11.50 paper; $16.95 hard)

101 Programming Surprises & Tricks for Your TRS-80™ Computer (No. 1741—$11.50 paper)

101 Programming Surprises & Tricks for Your ATARI® Computer (No. 1731—$11.50 paper)

How to Document Your Software (No. 1724—$13.50 paper; $19.95 hard)

101 Programming Surprises & Tricks for Your Apple II®/II®e Computer (No. 1721—$11.50 paper)

Scuttle the Computer Pirates: Software Protection Schemes (No. 1718—$15.50 paper; $21.95 hard)

Using & Programming the Commodore 64, including Ready-to-Run Programs (No. 1712—$9.25 paper; $13.95 hard)

Fundamentals of IBM PC® Assembly Language (No. 1710—$15.50 paper; $19.95 hard)

A Kid's First Book to the Timex/Sinclair 2068 (No. 1708—$9.95 paper; $15.95 hard)

Using and Programming the ADAM™, including Ready-to-Run Programs (No. 1706—$7.95 paper; $14.95 hard)

MicroProgrammer's Market 1984 (No. 1700—$13.50 paper; $18.95 hard)

Beginner's Guide to Microprocessors—2nd Edition (No. 1695—$9.95 paper; $14.95 hard)

The Complete Guide to Satellite TV (No. 1685—$11.50 paper; $17.95 hard)

Commodore 64 Graphics and Sound Programming (No. 1640—$15.50 paper; $21.95 hard)

TAB TAB BOOKS Inc.

Blue Ridge Summit, Pa. 17214

Send for FREE TAB Catalog describing over 750 current titles in print.

MacPascal Programming

Drew Berentes

Discover how easily you can master Pascal language for the Macintosh with this user-friendly guide!

Pascal is an increasingly important computer language that's not only easy to learn, it's especially effective when combined with the unique features of the Macintosh!

Now, with the easy-to-follow learning techniques highlighted in Berentes new MacPascal guide, you'll be amazed at how quickly you can begin using this versatile language to understand and use the full programming abilities and user-friendly characteristics your Mac was originally designed to display.

MacPascal Programming uses a building block technique to teach you the essentials of writing real working programs right from the start. Leading off with the basic features of the language, you'll cover progressively more detailed functions in logical sequence . . . using actual program examples.

Data types, text output, graphics and music, control statements, strings, and more are explained in detail. With this background, more complex concepts such as repetition, files, new data types, and binary searching fall easily into place. You'll see how a Pascal program is constructed, what each function "does" when entered into the computer, how to get the most from Mac's exceptional graphics, and how to integrate sound into your programs. Plus, you'll find such extras as:

- A fully worked out Inventory program.
- A complete listing of Pascal reserved words, procedures, and functions.
- An exceptionally well-documented approach to taking advantage of the Mac's user interface.
- Hands-on guidance in applying Pascal to real-life problem-solving.

Drew Berentes is a training designer for Texas Instruments and is currently working towards his Ph.D. in the use of computers in education. He is also the author of TAB's well-received *Apple® Logo: A Complete Illustrated Handbook.*

TAB TAB BOOKS Inc.

Blue Ridge Summit, Pa. 17214

FPT > $16.95

ISBN 0-8306-1891-0

PRICES HIGHER IN CANADA

1645-0685