Addison-Wesley Publishing Company

SCOTT KRONICK

Macintosh Pascal Illustrated

The Fear and Loathing Guide

MACINTOSH PASCAL

Illustrated

MACINTOSH PASCAL

Illustrated

The Fear and Loathing Guide

by **SCOTT KRONICK**

ADDISON-WESLEY PUBLISHING COMPANY, INC.

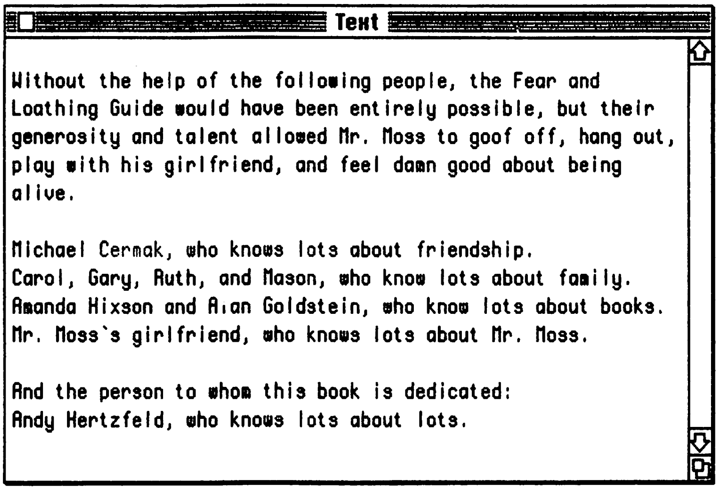
Reading, Massachusetts   Menlo Park, California   Don Mills, Ontario
Wokingham, England   Amsterdam   Sydney   Singapore   Tokyo
Mexico City   Bogotá   Santiago   San Juan

Library of Congress Cataloging in Publication Data

Kronick, Scott.
Macintosh Pascal illustrated.

Includes index.
1. Macintosh (Computer)—Programming. 2. PASCAL
(Computer program language) I. Title.
QA76.8.M3K76 1985 001.64'2 85-3907
ISBN 0-201-11675-8

Second Printing, September 1985
BCDEFGHIJ-HA-898765
Set in 11 pt. Garamond by General Graphic Services, Inc., York, PA
Text design by Patty Lowy, NY, NY
Cover design by Marshall Henrichs



Without the help of the following people, the Fear and Loathing Guide would have been entirely possible, but their generosity and talent allowed Mr. Moss to goof off, hang out, play with his girlfriend, and feel damn good about being alive.

Michael Cermak, who knows lots about friendship.
Carol, Gary, Ruth, and Mason, who know lots about family.
Amanda Hixson and Alan Goldstein, who know lots about books.
Mr. Moss`s girlfriend, who knows lots about Mr. Moss.

And the person to whom this book is dedicated:
Andy Hertzfeld, who knows lots about lots.

Contents

MACINTOSH
PASCAL
Illustrated

Introduction

If coloring books had instructions, kids would just chew the pages. Give them some crayons. Leave them alone. They don't need paper. They can Michelangelo the wall.

Macintosh is a mean set of crayons. Besides pictures, words, and numbers, it draws tools that work and makes voices that sing. But with the holy goose as my witness, no fat-pouted, mealy-mouthed smudgepot calling *hisself* a teacher is going to put his pigeon-smear ideas about computers near my new Macintosh.

That goes for books, too.

Computer books have got a boring evil that will put the devil to an iceman's sleep.

Macintosh shows me what I need to know, and when I muck up bad, it tells me that also.

All things considered, my Macintosh is probably stupider than my houndog, Rollo. Though for a machine, it has plenty of spunk. What's more, Rollo and I have taken a liking to Mac, and we're the kind of folk who look after those we care about.

Rollo understands plain English. Macintosh talks something called Macintosh Pascal. Without resorting to teaching—call me a teacher and you'll get a lip swollen the size of Missouri—there are some pointers worth knowing.

Part 1 puts your hands on the Mac. A step-by-step guide shows you how to use all the menu options of the MacPascal system.

Part 2 starts you writing programs with the Macintosh spirit. You will explore Pascal programming by using Quickdraw, the mouse, the printer, the disks, sound, the calendar/clock, and more from the heart and soul of the Macintosh Toolbox. Every chapter shows a working, practical program that you can build upon. Every program adds to your understanding of Pascal.

Part 3 turns you loose. A learn-by-example encyclopedia of Pascal, Quickdraw, and the Toolbox gives you immediate access to knock-out Macintosh programming. Almost every entry is illustrated with a stand-alone program, showing MacPascal's vocabulary in action. Beneath every program is the program's output, giving you the freedom to explore programs even when your computer is turned off.

No flow charts. No grammar lectures. No slobbering end-of-chapter exercises you would never do anyway. This book shows MacPascal sharp, active, and ripe for improvisation:

with 100 complete, unchopped programs.

The programs you create will have the *Mactintosh feel* that sets Macintosh apart from all other computers.

Any of you hotshot programmers may want to read this guide from the end backward. No matter how fast you get there, you will end up playing hot and heavy with the Toolbox—the tightest, cleanest box of crayons any kid has had in history.

Mr. Moss

PART 1

Running MacPascal: The User's Guide

The six chapters of Part 1 show you how to use Macintosh Pascal. All of the MacPascal menu options are introduced. You will see how to enter, save, run, print, alter, and investigate a Pascal program.

Investigate a program? MacPascal offers features that show you how a computer program operates—many of which have never before been offered by other computers or other versions of Pascal. At your command is a pointing hand that leads you through a program, a practice area for trying out Pascal instructions without actually writing a program, and an observation deck that allows you to look *inside* a program while it is running.

All along, even as you are typing at the keyboard, MacPascal checks for any errors you might make, and tells you where and why you have gone astray. And from the start, the *Fear and Loathing Guide* helps you prevent many problems from ever occurring.

Here is a summary of what you will find in Part 1:

- You will type in and run a tiny program called *FourPlay,* then enter and save an animated graphics program called *ComputerSewer,* which will be used to illustrate MacPascal in all six chapters.
- You will see the ways that MacPascal catches errors in

programs and find out how to protect your program from disk and power foul-ups.

- You will explore the Run menu's options, including MacPascal's unique *steps* and *stops,* which help you execute, and understand, a program line by line.

- You will experiment with the File, Edit, Search, and Windows menus to print a program, close a program, begin work on a new program, and edit a program quickly with the mouse and the automatic search option.

- You will use the Instant window to begin hands-on exploration of Macintosh's amazing graphic capabilities known as Quickdraw.

- You will open the Observe window and observe the changing values of a program in progress, a powerful tool to help you understand the mechanics of a computer language.

1 Entering the Sewer with a Boy or Girl Mac

1.1 When the going gets tough

Hook up your cords. Turn it on. Disk or no disk, your Macintosh comes alive, drinking 60 watts. Head for the fridge. Pet the dog. Unless you are gone for a couple days, better to leave Mac on. A powered-up Macintosh begs for attention.

Do you know how to pull down menus? Click and double click the mouse? If not, go directly to MacPaint. Create a masterpiece. But do not cut off your ear. You will need ears for the chapters on making sound.

Do you know how to use the mouse to edit text? If so, you already know most everything necessary to enter and manipulate a Pascal program. The manuals for both MacPascal and MacWrite explain how to edit text. The mouse, file, and edit features of MacPascal are nearly the same as those of MacWrite.

Macintosh programmers are not wimps. They can draw and write. And they can wield a mouse like pros.

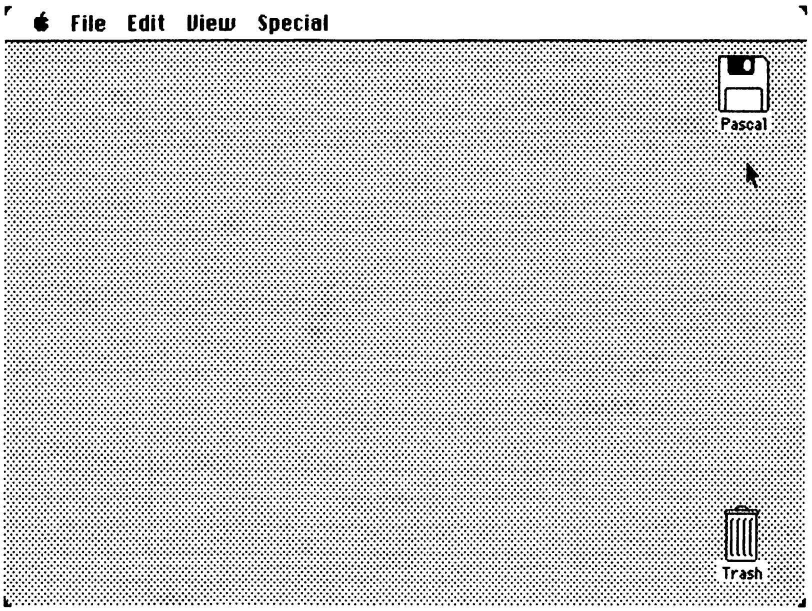
At Apple, people say, "When the going gets tough, the tough turn pro." Mr. Moss says:

When the going gets tough,
spend your nights with someone who isn't.

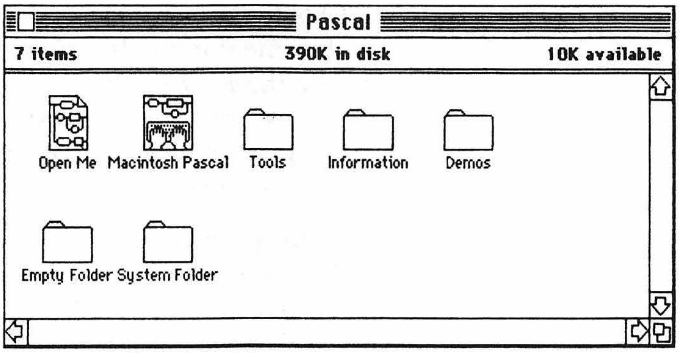
1.2 Those who can do; those who cannot teach; and those who cannot teach teach programming

If you are new to the Pascal programming languages, or all programming languages, you can learn by practicing with the programs in this book. The grammar and structure of Pascal are easier to learn after, not before, you have practiced running small programs.

Insert the Pascal disk. Here is what you see.



Double click on the Pascal disk icon. The disk icon opens into the MacPascal Desktop window.



If someone else has used the Macintosh Pascal disk before you, or if you are using an updated version, your Desktop window might be different from the one described in this section.

The MacPascal Desktop window initially displays several icons. The icon titled *Macintosh Pascal* opens to the Pascal language. You will be using this icon in the next section.

The System Folder contains operating information that Macintosh needs. You will not need to access anything directly in the System Folder to use MacPascal, yet it must stay on the disk.

The MacPascal icon and the System Folder are the only essential items on a MacPascal disk. At a later time, all other programs and folders should be dragged to the Trash to make more room on your Pascal disk. But before you trash anything, transfer copies of all programs and folders you might want to use later to a blank disk.

The Demo Folder holds a windowful of nifty sample programs. They are fine for seeing how Pascal works, but once you have written a couple programs on your own, you will say of these demos, "Gag me with a PCjr." Heck, later in this chapter your ComputerSewer will knock the socks off MacPascal's samples.

The Information Folder contains five text documents. The document titled *Read Me* introduces the MacPascal disk. The four other documents contain information that was left out of the *Reference Manual.*

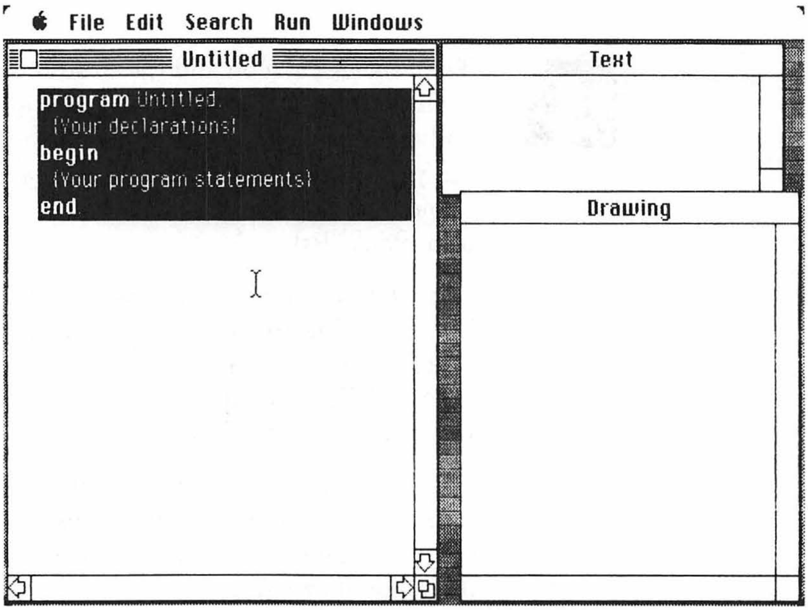
The Tools Folder contains three programs. "Browser" lets you select any document in the Information Folder, or any other file, for viewing on the screen. "PrintTextFile" offers the same option as "Browser," except instead of displaying the file on the screen, it prints a copy on your printer. A third program, "TextEditor," allows you to create your own text documents.

The "Open Me" program brings to the screen the text document called *Read Me.* The Read Me document explains how to use the Tools Folder to read what is in the Information Folder. The Read Me document happens to mention a book by another publisher to which Mr. Moss and the *Fear and Loathing Guide* say: Eat sewer, pal.

1.3 A tiny tease of a program

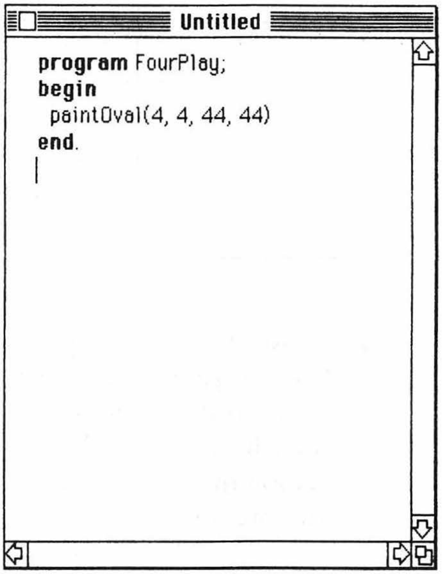
In this section you will type then run a Pascal program that draws a solid black circle in the Drawing window.

Double click on the Macintosh Pascal icon. Here is what you see.



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

Press the *backspace* key once. This will erase the blackened program outline that appears in the Program window. Then type the program as you see it below.

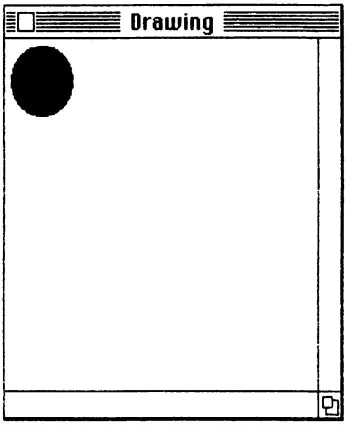


```
program FourPlay;
begin
 paintOval(4, 4, 44, 44)
end.
```

Do not bother to indent lines or use bold lettering. Macintosh Pascal does this for you automatically. The upper-case letters serve as a convenience for readability.

If you make an error or see your last typed line become Outlined, backspace or mouse-edit the unwanted characters and try again. The outlined code will return to normal print when the error is corrected and you click the mouse within the Program window.

Select Go from the Run menu. Here is what you should see in the Drawing window.



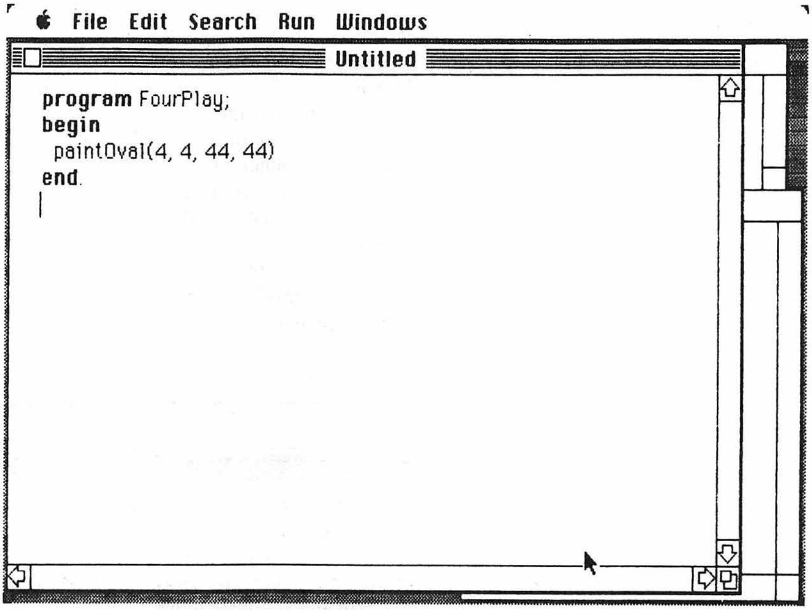
If the Macintosh makes a beeping sound and a message box appears with a picture of a bug, click the mouse with the cursor arrow anywhere inside the message box. The bug message will go away. Now edit or retype the five words, four numbers, and seven punctuation marks *exactly* as you see them in the Program window on the previous page.

When a program runs, the output is displayed in the Drawing window or the Text window. Graphics appear only in the Drawing window. Text can be written in either the Text window or the Drawing window. Text intended for the Text window uses different Pascal instructions than text intended for the Drawing window. Later chapters will show you how to take advantage of both windows.

1.4 The real stuff

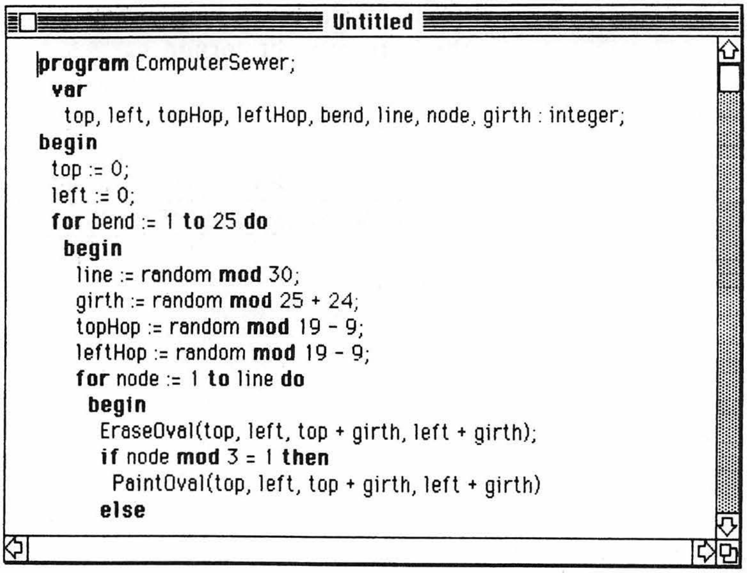
In this section you will type in an animated graphics program that will be used to demonstrate MacPascal throughout Part 1.

Resize the Program window so that it fills most of the screen. This will prevent the longer lines of the program from being hidden behind the right margin of the window. In case you are still new to the idea of Macintosh windows, they are resized by holding down the mouse button and dragging the cursor as it points in the small box (showing two overlapping rectangles) at the bottom-right corner.

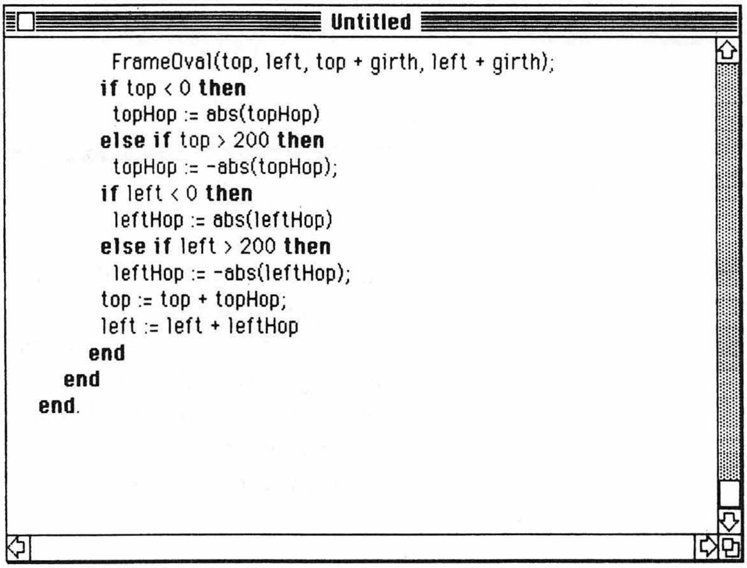


```
program FourPlay;
begin
 paintOval(4, 4, 44, 44)
end.
```

Now, you need to get rid of FourPlay. Move the cursor back into the Program window and select the text of FourPlay by dragging the mouse down the left margin of the Program window while holding down the button. Selected text appears blackened—white letters on a black background. Press the *backspace* key once to erase the selected text. Then type the program as you see it on the next page.



```
program ComputerSewer;
 var
  top, left, topHop, leftHop, bend, line, node, girth : integer;
begin
 top := 0;
 left := 0;
 for bend := 1 to 25 do
  begin
   line := random mod 30;
   girth := random mod 25 + 24;
   topHop := random mod 19 - 9;
   leftHop := random mod 19 - 9;
   for node := 1 to line do
    begin
     EraseOval(top, left, top + girth, left + girth);
     if node mod 3 = 1 then
      PaintOval(top, left, top + girth, left + girth)
     else
```



```
      FrameOval(top, left, top + girth, left + girth);
     if top < 0 then
      topHop := abs(topHop)
     else if top > 200 then
      topHop := -abs(topHop);
     if left < 0 then
      leftHop := abs(leftHop)
     else if left > 200 then
      leftHop := -abs(leftHop);
     top := top + topHop;
     left := left + leftHop
    end
  end
end.
```

The Program window will scroll by itself as you fill the window with code. Use the scroll bar when you want to see a different section of the program.

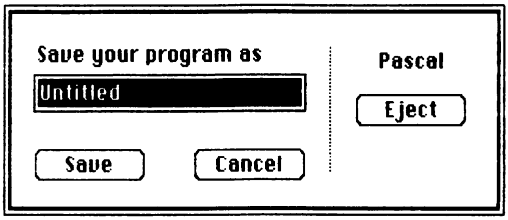
The same rules found in FourPlay apply to ComputerSewer: Indentations and **bold** lettering are done automatically by

MacPascal. The Outlined lettering means a mistake needs to be corrected.

1.5 A quick save and a green light for the brave

Before going any further, even before you run ComputerSewer for the first time, you should save the program you have typed onto a disk. Saving a program onto disk is done through the File menu. The contents of the Program window will be written onto a disk under a name you assign. Once saved, a program will be represented by its name and icon in the MacPascal Desktop window. A program saved on disk can be brought back onto the Macintosh screen by double clicking on the program's icon.

Choose Save As from the File menu. The following dialog box will appear.



Save your program as

Untitled

Type in the name ComputerSewer. Now click the mouse as it points on the Save button. The disk should whir and a copy of ComputerSewer should be safely stored onto the Pascal disk.

Now that you have saved ComputerSewer, you can leave MacPascal (by choosing Quit from the File menu), or even turn off the Macintosh, without having to retype the program the next time you want to use it. You can return the text of ComputerSewer to the Program window at any time by double clicking on its icon from the MacPascal Desktop window.

Chapters 2 and 4 have more information on saving programs on disk and using the File menu.

Interested in seeing what ComputerSewer actually does? If you are brave, bring forward the Drawing window by clicking the button with the cursor arrow anywhere in the Drawing window, then pull down the Run menu and select Go. If bravery is not your forte, wait until after the next chapter to select Go—then you will be better prepared to risk the twisted and ugly face of the *bug message.*

1.6 How to tell if your Mac is a boy or a girl

Macintosh computers are machines. There is no boy or girl variety. The fact that Mr. Moss calls his Macintosh *Twila* is not an attempt to ascribe human traits to a machine. Nor should the feminine name lead anyone to believe a computer has surrogate potential. To the contrary, what a Macintosh can do best is allure a special friend to your side. From then on, the evening is your responsibility: fine wine, candlelight dinner, Macintosh demonstration, two hands reaching for the mouse, touching . . .

Programming artists perform with imagination. The others crank COBOL on corporation mainframes.

2 Low-Down, Cheating Computers

What's next

Did you have any problems typing in ComputerSewer? If so, you are in the right place. This chapter covers trouble and how to get out of it. Everything that possibly could go wrong in a Pascal program is documented. And if you believe that, Mr. Moss requests you write him regarding real estate opportunities overlooking the Everglades.

2.1 White shirts, blue suits

Bugs, errors, glitches, zaps, bombs, crashes—programmers have more names for failure than Eskimos have for snow. No surprise that so many people's attitude toward computers stinks.

Mr. Moss points to one industry giant for fostering the "hands off, moron, they're *our* computers even after you buy them" mentality, but adds, "I Better not Mention who."

Excuse Mr. Moss for any hint of bitterness, as his formative exposure to computers was gotten by handing punched cards through a window to a snotty grad student wedged in front of the room-sized university mainframe, and then waiting three days of down time and low priority to have a different trout-faced flunkie hand him back the cards and mostly blank, trash paper, saying a control card comma should have been in column 12 instead of column 11.

2.2. Mr. Moss eats bugs for breakfast

The last chapter gave you a taste of computer programming. If you are like most learners, while typing in ComputerSewer you probably felt as hesitant as you would biting into lizard kidneys. Perhaps you made an error or came to a point where you were unsure of how to proceed.

A computer sucking electricity into its little plastic body can spew gibberish faster than a late-night carpet dealer preaching on UHF. Errors will come at you in all directions. Great; scream at them, threaten them, belittle their silica origins. MacPascal's best feature is its uncanniness in flushing bugs from their slimy haunts.

The next three sections of this chapter show how Macintosh Pascal reacts to incorrect programming. Sometimes MacPascal will outline the words following an error. Sometimes a picture of a *thumbs down hand* will appear in the left margin of the Program window. Sometimes the screen will splot with queer letters and designs. Sometimes a picture of a bomb appears, saying "serious system error."

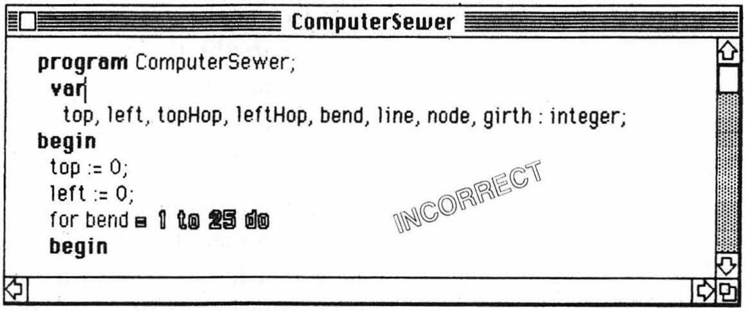
The bomb lies. No error is serious. Your program might be shot to hell, but programming errors are no more serious than is coming upon a new word while reading a book. Fixing those errors and recovering from undetermined computer weirdness demands, more than anything, your fascination. Your technical skill will grow as a result.

> Mr. Moss defines a serious error as failing to ask for a telephone number in a chance meeting with a person you would like to date.

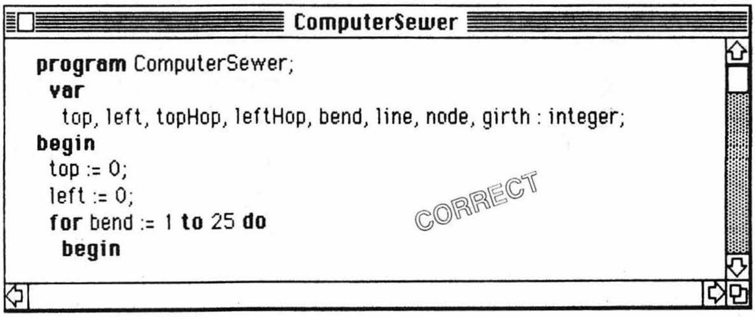
2.3 The outlined code

MacPascal checks for errors and formats each line of code with indentations and bold lettering after you type it in. Hitting the *return* key or inserting a semicolon tells MacPascal you have finished typing a program line.

The checking process will cause any unacceptable Pascal usage to be outlined from the point of the error to the end of the line. Here is a side-by-side example of an incorrect and correct Pascal command.



```
program ComputerSewer;
 var
  top, left, topHop, leftHop, bend, line, node, girth : integer;
begin
 top := 0;
 left := 0;
 for bend = 1 to 25 do
 begin
```



```
program ComputerSewer;
 var
  top, left, topHop, leftHop, bend, line, node, girth : integer;
begin
 top := 0;
 left := 0;
 for bend := 1 to 25 do
  begin
```

If you are curious about why the **for..to..do** statement requires a colon/equals sign (: =), flip forward to **for..to..do** in Appendix F, Part 3.

Whenever you get curious about the rules for using Pascal instructions, flip to the alphabetical list of explanations and examples in Appendix F, Part 3. Better to pace yourself than to choke on the "Pascal Stuffed Down Your Throat Approach" employed by the scholarly fishwrap.

Click into the Program window and introduce the error described above. Find the *for bend := 1 to 25 do* statement and erase the colon. Click the mouse and watch the line become outlined. Now you can fix the error simply by inserting the colon in the proper place and clicking again. The outlining returns to normal print without your having to backspace or erase the entire outlined code.

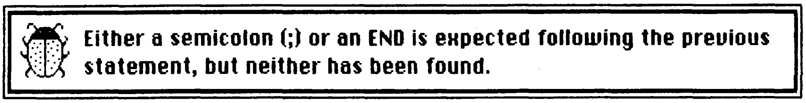
In addition to checking each program line for proper usage, MacPascal formats the line with indentations, and **bold** lettering. Not only does this make the program easier to read, it brings to your attention errors that might be grammatically correct but operationally not what you intended.

For instance, if you were to use the words **for**, **to**, or **do** in any other context than the **for..to..do** statement, their bold lettering in the Program window would bring to your attention their *reserved word* status in MacPascal. Likewise, you might notice an errant semicolon because of the indention of the next code line.

MacPascal reserves the use of thirty-eight special purpose words and highlights those words in **bold** lettering when a line is formatted. See Appendix E for a list of *reserved words*; also, each of MacPascal's *reserved words* is defined in Appendix F, "The Whiz Kid's Dictionary."

2.4 The run time bug message

Bug messages appear when you run an improperly constructed program. Here is a sample bug message:

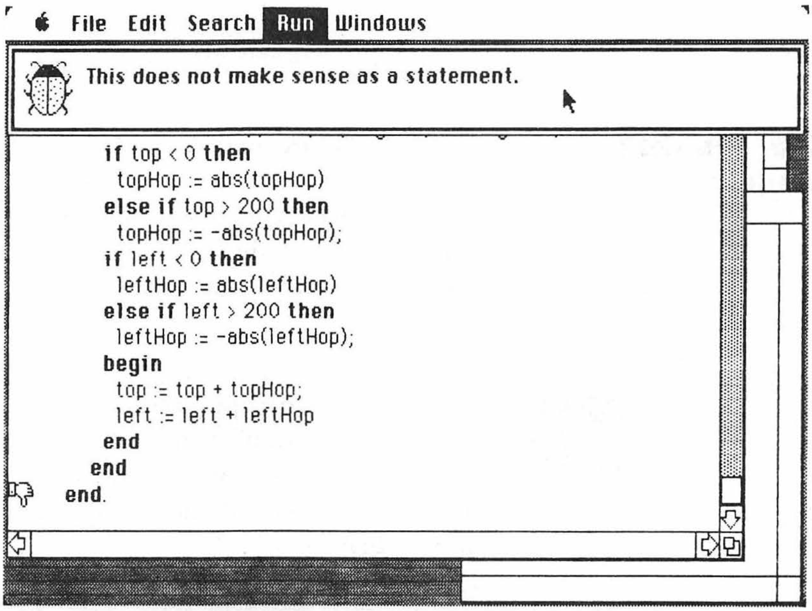


Either a semicolon (;) or an END is expected following the previous statement, but neither has been found.

There are innumerable ways of improperly constructing a program; however, there are a limited number of different bug messages. This means the bug message may not exactly identify your particular error. Clicking within the bug message will make the bug message disappear and enable you to make adjustments to the Program window.

In addition to the bug message, MacPascal will display a *thumbs down hand* in the left margin of the Program window at the line where the error is detected. Program execution will be halted at this point. Until you fix the infraction, there is no way for the program to proceed beyond the thumbs down hand.

The computer operates with Mr. Spock logic, yet many steps may pass before an error offends this logic. The moral: Don't expect the bug message and its thumbs down hand to explain and pinpoint all errors. In the example below, ComputerSewer was altered by inserting an extra **begin.** The pointing hand does not detect the error until the program finds that there is no corresponding **end.** Turn to the dictionary in Part 3 if you want to know more about the hows and whys of **begin** and **end.**



This does not make sense as a statement.

```
        if top < 0 then
         topHop := abs(topHop)
        else if top > 200 then
         topHop := -abs(topHop);
        if left < 0 then
         leftHop := abs(leftHop)
        else if left > 200 then
         leftHop := -abs(leftHop);
        begin
         top := top + topHop;
         left := left + leftHop
        end
       end
     end.
```

2.5 The glitch and the glop

Evil manifests itself most forwardly with the glitch and the glop. These are errors in the performance of a program, not its machine readability. No outlined code. No bug messages. Just garbage in the works. Though glitch and glop can be caused by computer equipment problems, more likely than not your programming will be the source.

A glitch is an unpleasant, often cosmetic, flaw. A program that outputs "two plus two equals 4our" has a glitch. Glop is a zit on your nose on prom night. "Two plus two equals five" is glop.

MacPascal offers tools to fight the glitch and the glop. Options from the Run menu permit line-by-line program execution. The

Instant and Observe windows of the Windows menu help you explore the dynamic guts of a running program. Use of these tools is illustrated in the next few chapters.

Glitch and glop are ridiculous terms used to say that a program needs more work. More pertinent to a project's completion is the creator's pride in his or her work. Good programs result from clever engineering, not from removing bugs. A programming artist's worst fear is that after all the glitches and glop have been removed, the result will be perfectly running, error-free gunk.

2.6 The disk munch

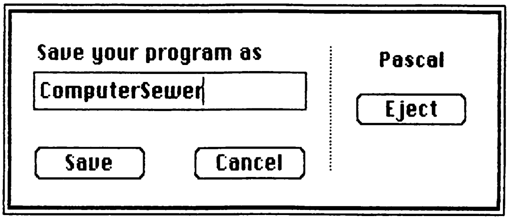
A power failure or system error will clear the memory of the computer and cause you to lose the contents of the Program window. However, if you have saved the program onto a disk before such a failure, you can reload the program by double clicking on the program's icon from the MacPascal Desktop window.

When you save a program, MacPascal creates a program icon with your program's name and installs it in the MacPascal Desktop window. A second copy of the disk will further protect you should your basset hound do something unmentionable with your floppy disk.

Mr. Moss's dog, Rollo, has never dumped in the house. The one time Mr. Moss found a deposit on the floor of his study, Rollo pointed to the computer and said it was the disk's fault. The disk was punished—Mr. Moss loaned it to a Hewlett-Packard salesman for a week—and a lesson was learned.

Disks, like reptiles, act on instinct, and should not be left alone with loved ones.

Saving a program onto disk is done through the File menu. In Chapter 1 you were instructed on how to save ComputerSewer. You chose Save As from the menu, typed in a name, then either pressed *return* or clicked the Save button. The File menu is covered in more detail in a later chapter, but if you have not done so already, save ComputerSewer onto a disk now.



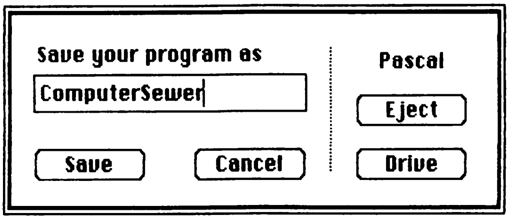
Save your program as

ComputerSewer

The code in the Program window should be saved on a disk frequently enough so that if someone accidently pulled the plug on your Macintosh, you would not commit a felonious assault. For a steel-trap memory like Mr. Moss's, this translates to every fifteen minutes.

Every two or three saves to disk, save a copy of your program on a second disk. This is done by again choosing Save As from the File menu, clicking the Eject button, then, since the program name is already typed in from the first save, simply click the Save button, and follow the Macintosh swapping instructions.

Those chumps lucky enough to have two disk drives need only to click the Drive button to select the disk drive onto which the program will be saved. The name of the currently selected drive appears above the Eject and Drive buttons.



Save your program as

ComputerSewer

No gruesome horror stories of lost work are forthcoming. An accidental yank of the Macintosh power cord or a sudden power surge when your kid brother shuts off his model cyclotron could wipe out your work in progress. Your only recourse is to protect everything you do by periodically—that means often—saving the contents of the Program window onto a disk. Then make a second copy of the program on disk by saving it onto another disk.

Disks are cheap. Heart failure is not.

Most disks die with bug messages explaining their demise. Others you can tell have croaked only because they are mired in glop. Be sure to check the tiny *write-protect notch* in the back-left corner of the disk before you decide a disk cannot be written on.

On Apple disks, opening and closing the sliding notch will respectively prevent or permit saving new information onto a disk. If you have a finished disk you do not want altered, protect it by opening the notch. Paper clips work better than fingernails.

Macintosh disks are remarkably reliable compared to the flexible disks used by many other machines. Do not let the cautions in this chapter turn your enthusiasm to paranoia. When and if the disk munch strikes, try to figure out what has happened. Like all programming trouble, you can learn things from disk problems you would not have learned elsewhere.

2.7 The burnout

Untalented programmers who enjoy their work are better off than *Wunderkinds* who always sleep alone at night. Balance your efforts. Compulsive behavior is gross no matter what the endeavor. Your health, friends, generosity, and passions need to balance with the hard work it takes to express yourself on a computer.

Choose your own subject material. Don't let a jerky teacher who forces you to write jerky programs ruin programming any more than you would let a deranged English teacher who believes no good literature has been written since the eighteenth century ruin books. Writing code is a pain in the ass if you don't appreciate the content of your program.

Pascal may be difficult, but challenges bring rewards; easy pleasures make you fat. Playing basketball like Dr. J or painting pictures like Georgia O'Keeffe is hard, too.

As computers evolve, they will become primarily vehicles of expression: thinking pens and paintbrushes more than spreadsheets or processors. The programmers who will make this happen will not find their resources in textbooks, classrooms, or years spent sequestered with electronics, but from the expansiveness and sensuality of their lives.

Call Mr. Moss a harping, maudlin piddlesnort who ought to stick to the business of Pascal, but don't blame him when your dreams at night fill with bloated hex demons, deviant variants, and sucking black hole recursions.

3 Don't Call It the Runs Menu

What's next

After a few words on windows, this section shows ways to run and stop a MacPascal program. The illustrative pointing hand of MacPascal will track the flow of program execution.

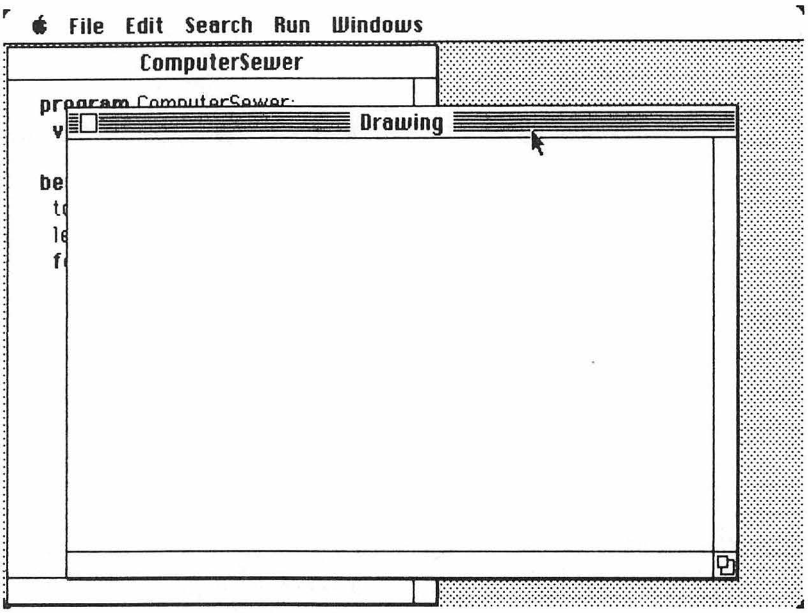
3.1 Resizing the drawing window

In Chapter 1 you typed ComputerSewer into the Program window. The Program window still hogs most of the Macintosh screen. Since the screen output of Pascal programs uses the Drawing and Text windows, the first step in running a program is making space for its output.

Clicking anywhere within a window will activate that window. The lines in the top of the window frame indicate an active window. You will find that many of the menu bar choices are dimmed, and thus unavailable, when the Program window is not active.

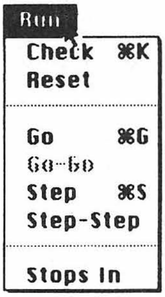
Click anywhere inside the Text window. Since you will not be using the Text window until Part 2, close the Text window by clicking inside the small Close box in the window's upper-left corner.

Now, reposition and resize the Drawing window to fill nearly all the screen. The Drawing window handles both pictures and words, and will show the output of ComputerSewer.



You have closed the Text window and buried most or all the Program window, but who cares. ComputerSewer now has some elbow room.

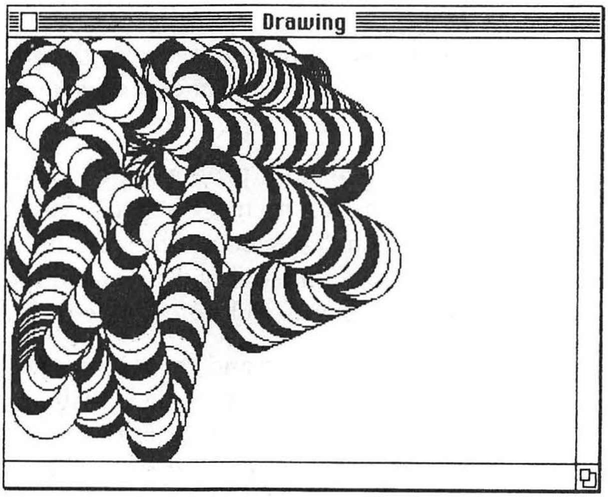
3.2 Am I really the first?



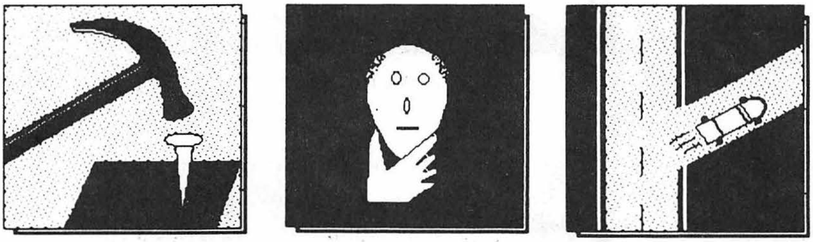
Choose Go from the Run menu. If an error message appears, find and correct the mistyped code in the Program window. Otherwise, sit back and relax. With any luck, your Macintosh will be as gentle with you for your first time as Twila was with Mr. Moss.

The disk purrs. The menu bar flickers, dims. Run is accentuated. Pause appears. For twenty seconds an unearthly tunnel weaves a cathodic tapestry. Then it is over. You are no longer a programming virgin. Maybe you expected rockets and fireworks. Instead you got the ComputerSewer. Mr. Moss puts his arm around your shoulder and tells you, not for the last time, his hurtful, troubling wisdom:

Programming is less fun than being in love.



3.3 Icons and Mr. Moss's girlfriend mentioned



In case you're wondering how many icon drawings this book uses, here they are together. When you see the *hammer ready to hit the nail,* you are being asked to do the following activity. The *thinking bald man* means a summary or noteworthy explanation is being presented. The following paragraph, as shown by the *vehicle hanging a right turn,* is a sidetrack.

Mr. Moss hates to be told to do things. When a book says *do this,* he purposely does it differently. When a program goes berserk because of a change Mr. Moss has made, he may laugh or curse, snicker or stomp. Yet he will never let a computer daunt him. Mr. Moss allows himself to be daunted only while he is alone with his girlfriend and the door to the room is closed.

3.4 Going, halting, stepping, and veal

Associating *run* and *menu* gives Mr. Moss the heebie-jeebies. Thank goodness the menu does not have an option called *veal.*

You can make a program Go in more ways than choosing Go from the Run menu. Choosing Step will run exactly one line of a program. To execute the next line you must select Step again. A picture of a hand will appear in the Program window pointing to the line that will be executed next.

Using the mouse and the Run menu can be tedious for stepping through many program lines. Instead, press together the *command* key and *s* key. Hereafter, the *command* key will be called the *cloverleaf* because the key symbol looks like a highway cloverleaf. This shortcut makes the stepping process more convenient.

All of the shortcut keypresses make use of the *cloverleaf* key and a single letter. The pull down menus show which keypress combinations can be used as shortcuts instead of pointing the mouse at a menu.

Click the mouse with the cursor arrow in the visible sliver of the Program window. This will bring forward the Program window. You could also activate the Program window by choosing ComputerSewer from the Windows menu.

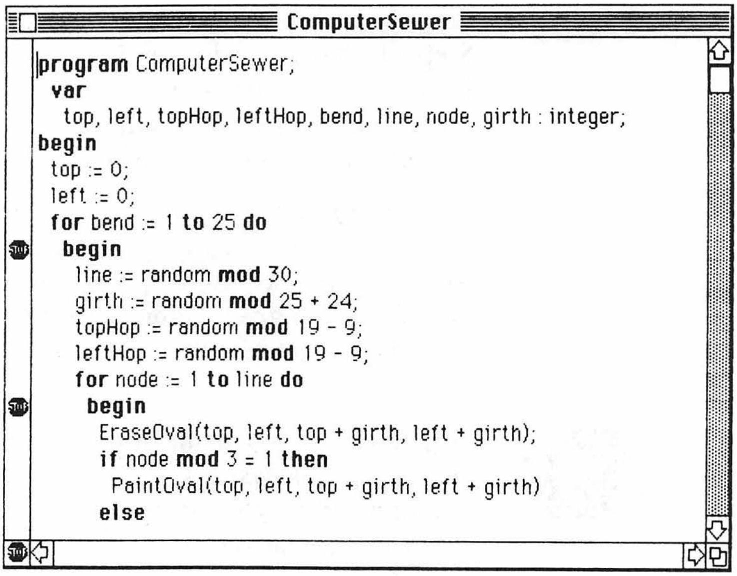
Now Step through ComputerSewer. The pointing hand of MacPascal better illustrates how a computer language talks to a computer than a thousand words dribbled from Mr. Moss's mouth.

Step-Step also brings out the pointing hand, however, unlike Step, the program does not stop after executing each program line. The hand step-steps right on through the text of the program, beginning to end, filling the Drawing window with sewer the same as the Go command.

The sewer grows much slower with Step-Step than with Go. After all, running with Step-Step points and scrolls along the Program window, whereas Go simply draws a sewer. One reason to use Step-Step is to give you more time to go up the Pause menu and Halt the program at the place you want it halted.

Stopping a program midstream is one of your most powerful debugging devices. As you watch something go haywire in an output window, you can Halt the program and the pointing hand ought to be near the suspected bug.

When you select Stops In from the Run menu, the Program window adds a new feature. One or more stop signs can be inserted in the left margin of the Program window by positioning the arrow cursor in the margin. The cursor then becomes a marginal stop sign, and clicking will set it. To remove a single stop sign, click on top of it. To remove all stop signs and return the Program window to its original format, select Stops Out from the Run menu.



```
program ComputerSewer;
 var
  top, left, topHop, leftHop, bend, line, node, girth : integer;
begin
 top := 0;
 left := 0;
 for bend := 1 to 25 do
  begin
   line := random mod 30;
   girth := random mod 25 + 24;
   topHop := random mod 19 - 9;
   leftHop := random mod 19 - 9;
   for node := 1 to line do
    begin
     EraseOval(top, left, top + girth, left + girth);
     if node mod 3 = 1 then
      PaintOval(top, left, top + girth, left + girth)
     else
```

The Go or Step-Step commands will run a program from the beginning until the first occurrence of a stop sign. The program will halt with the pointing hand on top of the stop sign, telling you this line of code will be executed next. Subsequent run commands will resume execution from this pointing hand, stopping again at any other stop signs you may have placed.

The Go-Go command will run a program identical to Go except that all stop signs are ignored. This allows you to run the entire program without first having to remove your stop signs.

The steps and stops of MacPascal will become more valuable as the complexity of your programs increases. The path of programming logic, which academic bozos call *algorithms,* has a nasty tendency to loop and branch. This turns out to be quite useful because, as Einstein taught, if all roads were straight and you went to buy a quart of milk, you would have to go to the end of the universe to return home. Still, loops and branches of even the best laid plans can lead you to the ragweed-chewing farmer in coveralls and flannel shirt who when asked for directions will smartly say:

You can't get there from here.

3.5 Check and reset

The Run menu offers two more commands, neither of which actually run the program. Choosing Check will read the Program window and produce an error message if the program breaks any of Pascal's rules. The Go and Step commands will do this anyway, but not as fast.

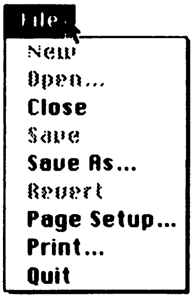
Choosing Reset is a way of making your program start from the beginning. After you have been using Step or Halt or Stops In, you can get rid of the pointing hand and begin the program afresh.

4 Macintosh Saves, Gretzky Scores on the Rebound

What's next

This section describes File, Edit, Search, and Window options. A work plan for programs-in-progress is presented.

4.1 Save revisited



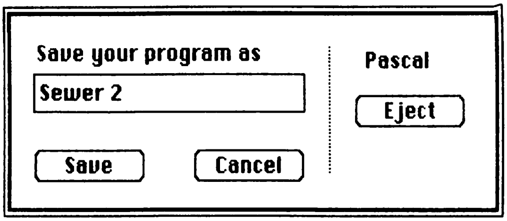
The File menu ought to be familiar to you from Chapter 2. After you typed ComputerSewer into the Program window, you saved the typed code onto disk by choosing Save As from the File menu. If for any reason you did not save ComputerSewer on disk, do so now by selecting Save As. Change the name if you wish.

A program saved on disk is called a *file,* and each file is given an identifying icon in the Desktop window, which opens when you insert the MacPascal disk. Thus, if you were to choose Quit, you would return to the Desktop window, also called the Finder, and see a picture of a piece of paper with the name "ComputerSewer," or whatever name you chose. (Keep your names under 20 characters. Spaces, numbers, and punctuation marks are okay to use.)

Don't use the Save option. The Save option, different from the Save As option, will not ask you to specify a file name. Save uses the contents of the Program window to *replace* the original file, giving the new file the same name. The old version will be lost—an unfortunate occurrence should you discover later that the old version worked better than your new one.

Better to use Save As, always giving a new version of your program a slightly different name. Save As allows you to save the old file along with the new file, provided that each is given a unique name.

Save a second copy of ComputerSewer using a similar name, perhaps Sewer 2.



Save your program as

Sewer 2

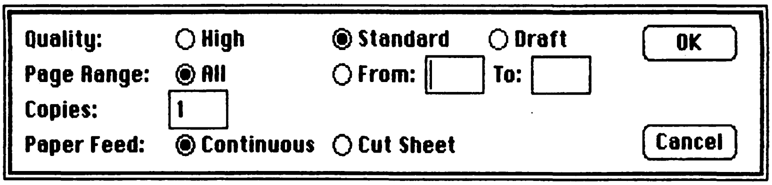
In the course of working on a program, your disk will fill with many versions. At this point, return to the Finder and drag your obsolete, unwanted versions into the Trash. Leave a couple of recent or tested versions as backup. And don't neglect to save recent or tested versions on a second disk. It is a simple procedure to click Eject from the Save As dialog box, insert a backup disk, and click the Save button.

4.2 How to print

In this section you will be creating a printed paper copy of the program code. You will also see how to print a copy of any single active window, as well as a method for printing an exact copy of all that you see on the Macintosh screen.

Instructions for attaching the printer to Macintosh come with the printer. If any printer trouble occurs while using Pascal, test the printer from MacPaint or MacWrite to make sure your hardware is functioning.

Selecting Print from the File menu produces a dialog box. The dialog box allows you to specify the kind of paper, print quality, page selection, and number of copies to print.



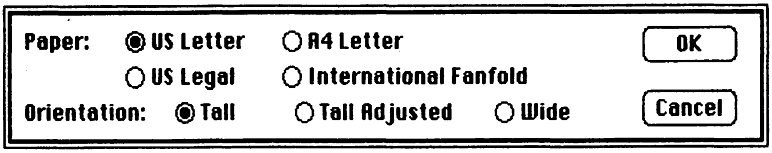
Quality: ○ High ● Standard ○ Draft

Page Range: ● All ○ From: [ ] To: [ ]

Copies: [1]

Paper Feed: ● Continuous ○ Cut Sheet

To set up a page in a nonstandard way, select Page Setup from the File menu prior to selecting Print, and toggle the dialog box options.



Paper: ● US Letter ○ A4 Letter

○ US Legal ○ International Fanfold

Orientation: ● Tall ○ Tall Adjusted ○ Wide

Each program's dialog boxes will remain adjusted to your specifications even after you have exited the program and turned the computer off. Your changes are recorded on disk along with the program.

Before attempting to print, check to see that the printer light and the Select light are on, then click the Okay button. After a few seconds delay, followed by a screen message telling you how to halt the printing process, the contents of the Program window should be printed.

The Print option of the File menu will be in bold lettering and available for use only when the Program window is the active window. If the Drawing or Text windows are active, Print will be dimmed in the menu. Consequently, the output windows—Drawing and Text—cannot be printed by using the mouse and the menu bar.

Any active window—Program, Drawing, Text, Instant, Observe, or Clipboard—can be printed by simultaneously holding down a three-key combination: *shift-cloverleaf-4*. Remember, the *cloverleaf* key, also called the *command* key, is marked with a highway cloverleaf design.

An exact copy of everything on the Macintosh screen will be printed by pressing in the *caps lock* key and then holding down the same *shift-cloverleaf-4* combination. This means everything: the active window, the visible portions of inactive windows, the menu bar, and the background pattern.

Whenever you print using the above key combinations, no dialog or message boxes will appear. The printing will be done in standard quality.

Draft-quality printing will not print full-size or bold lettering. However, the advantage of draft is that the printer operates twice as fast as with standard quality and four times as fast as high quality. Speed is nice for works in progress.

The best thing about paper copies of your programs is that you can carry your work to someplace really nice. Mr. Moss likes the city rose garden. Forget about Macintosh's cathodic omniscience. Think sunny outdoors or cozy fireplaces. Think of being beside your boyfriend or girlfriend, or sitting in a café where you might meet that special someone. Choose a study spot where you can concentrate on your work, look up, see faces and colors, unwind, refresh, concentrate again. There will be other opportunities to grind.

4.3 Open, close, revert, and new

Usually you will open programs by clicking on an icon in the Desktop window. Double click on the file icon and the program opens. Use Close and Open from the File menu if you are already working on one Pascal program and want to switch to work on another. Using Close and Open in this way is faster than using Quit and double clicking the next file's icon. The reason is each time a file is opened from the Desktop window, the Pascal language is loaded into memory. Open and Close act on the files alone; the Pascal language already in memory is unaffected.

Use Revert from the File menu to replace the current contents of the Program window with the most recently saved version. The Revert command cannot be undone, and as a precaution after selecting Revert, you will be warned by a dialog box that the current contents of the Program window will be lost. If you do not have a version of the program saved on disk under the same name as the Program window's title, the Revert option will be dimmed and inaccessible.

Use New from the File menu to obtain a clean, unused Program window in which to begin writing another program. Usually you will start a new program file by clicking on the MacPascal icon from the Finder. However, if MacPascal is already loaded, using New is faster than returning to the Desktop window with Quit and double clicking the MacPascal icon.

4.4 When to quit

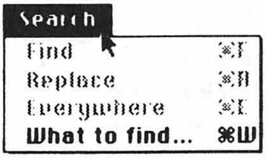
Use Quit when you have got a hot date. If you are not back by Sunday noon, instruct a friend to sublet the apartment and UPS the Macintosh to you.

Quit closes program files, leaves Pascal, and returns to the updated MacPascal Desktop window. At this point, you may want to free up disk space by dragging unwanted program versions to the Trash.

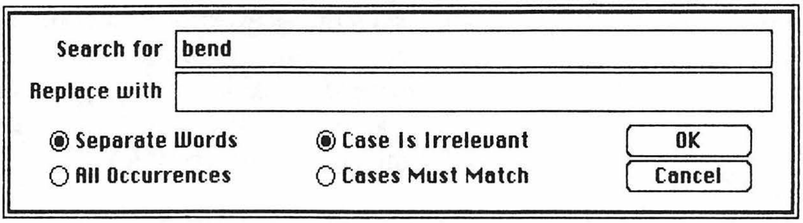
Turning the power off after working on a Pascal program does not substitute for selecting Quit. Quit performs protective housekeeping to ensure that your files can be opened in the same condition you left them.

4.5 Search and replace

In a computer program, there is sometimes the need to examine and possibly change a certain text. You may need to correct an error or add clarification by substituting new text into a program long after the original code was written. The Search option will help you find and, if desired, replace text in the program code.



Select What to Find from the Search menu.



Search for: bend

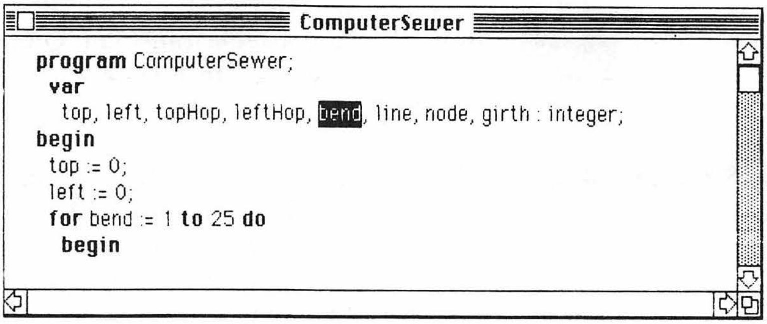
Replace with:

● Separate Words ○ All Occurrences

● Case Is Irrelevant ○ Cases Must Match

In the above example, to locate occurrences of the word *bend,* enter *bend* in the Search For box. Two check boxes (circles) ask whether you want to find *bend* as a separate word or all occurrences of the sequence of letters *b-e-n-d,* even if they happen to be part of a larger word such as *fender-bender.* The other two check boxes ask whether upper- and lower-case letters must match exactly or if case is irrelevant. For now, the boxes Separate Words and Case Is Irrelevant should be checked.

Click the Okay button and then choose Find from the Search menu. The first occurrence of *bend* found beyond the insertion bar is selected.



```
program ComputerSewer;
 var
  top, left, topHop, leftHop, bend, line, node, girth : integer;
begin
 top := 0;
 left := 0;
 for bend := 1 to 25 do
  begin
```

To find the next occurrence of *bend* select Find again, or try the shortcut *cloverleaf-f* key combination. Selected text can be replaced by cutting out *bend* and retyping *curve* or by using the Replace option of the Search menu.

Select What to Find again. The dialog box returns with *bend* still in the Search For box. Click in the Replace With box to put the insertion bar inside it. Type in *curve.* Click the Okay button to find the first occurrence of *bend.*

Now, choose Replace from the Search menu to replace *bend* with *curve.* The Replace option will change only the first occurrence of a word found beyond the insertion bar. Experiment using the above example, then try changing *curve* back to *bend.*

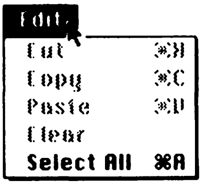
The Everywhere option of the Search menu works the same as the Replace option except that *all* occurrences of a word or text will be replaced. Everywhere performs an automatic Find and Replace from the insertion bar to the last occurrence of the indicated text.

Do not confuse the Everywhere option with the check box All Occurrences. Everywhere refers to everywhere the indicated word or text appears. The check box refers to the search for a sequence of letters such as *bend* occurring in *fender-bender.*

A warning message appears whenever you attempt to replace all occurrences of a word. Replacing all occurrences, especially a short sequence of letters, often encompasses more than you had expected.

Try using Everywhere. Override the warning by clicking Okay. Watch carefully in the Program window as each occurrence of *bend* is replaced by *curve.* Then reverse your changes by searching for *curve* and replacing them with *bend.*

4.6 The edit menu and the clipboard



MacPascal's Edit menu looks and works nearly the same as MacPaint's and MacWrite's. Mr. Moss could easily say "read about the Edit menu from the MacPaint or MacWrite manuals, then come back here when you are done." But failing to inform the reader on these important and somewhat difficult Macintosh concepts would be a cheap, inexcusable neglect of responsibility.

Read about the Edit menu from the MacPaint or MacWrite manuals, then come back here when you are done.

The Edit menu is most useful for copying sections of program code from one place to another. Without a lot of retyping or the expense of scissors, glue, and a Canon copier, you can rearrange a computer program.

The Clipboard is a temporary storage window for edited text. Using Cut or Copy from the Edit menu, depending on whether you want the original text to be removed or to remain, you can place selected portions of text onto the Clipboard. The Clipboard's contents can then be pasted, using Paste, elsewhere in the program. The Clipboard's contents can also be pasted into a different Pascal program on the same or another disk.

Select All is a quick way of selecting the entire text of a program at once. When text is selected, the characters are shown in inverse—white characters on a black background.

Practice by selecting a random chunk of ComputerSewer. Just drag the mouse vertically a few inches. Then choose Copy from the Edit menu. Now choose Clipboard from the Window menu to make the Clipboard visible. You can edit the contents of the Clipboard directly. Try it. Paste the revised text back into the Program window. Don't worry about messing up ComputerSewer because you have the original copy saved on disk.

Each Cut or Copy will replace the contents of the Clipboard with the newly selected material. The Clipboard does not work well for long-term storage purposes.

Clear works similar to Cut except that the selected text is not placed on the Clipboard, and therefore is not recoverable. Highlighted text can also be cleared by pressing the *backspace* key.

Like Clear, this will remove all selected text without the benefit of the Clipboard's temporary storage.

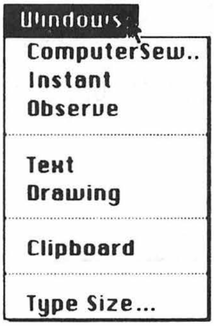
The Edit menu will become increasingly useful in your programming as you discover that some discrete sections of program code, called *subroutines,* serve a general purpose that many of your programs can use with little or no modification. *Searching* and *sorting* are two common examples of subroutine tasks that might be kept in a library file and inserted in your programs as desired.

Many of the programs to be found in Part 2 and Part 3 will be useful for your own programming projects. Cut, Copy, and Paste will make moving program code fast and convenient.

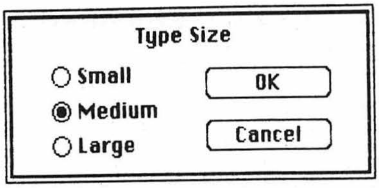
Starting a large computer program from scratch is a royal pain. Reusing subroutines makes the early stages of programming fun and visible. Likewise, don't be afraid to throw out everything you have done and start over. Learn from others. Programming artists enjoy their work.

Programming martyrs die of boredom.

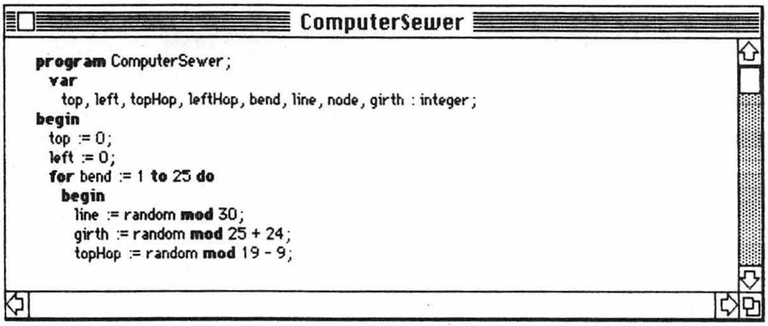
4.7 Whaddyamean I'm not your type



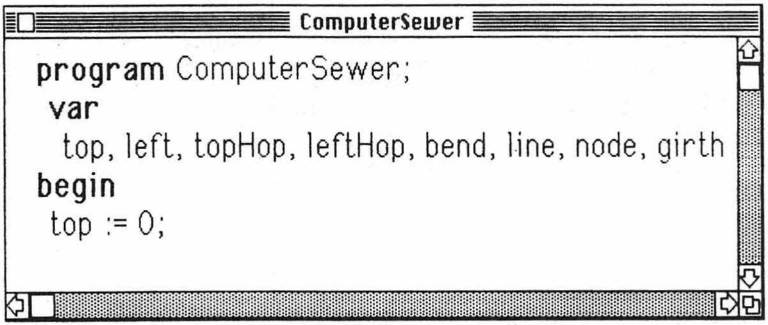
Choosing Type Size from the Window menu produces a dialog box offering three sizes of type. The selected size will appear in all windows. Macintosh Pascal defaults to the middle size. The small size is nice for viewing lots of code with a minimum of scrolling. The large size might be useful for people who have tampered with their brains the night before.


Type Size

○ Small
● Medium
○ Large



```
program ComputerSewer;
 var
  top, left, topHop, leftHop, bend, line, node, girth : integer;
begin
 top := 0;
 left := 0;
 for bend := 1 to 25 do
  begin
   line := random mod 30;
   girth := random mod 25 + 24;
   topHop := random mod 19 - 9;
```



```
program ComputerSewer;
 var
  top, left, topHop, leftHop, bend, line, node, girth
begin
 top := 0;
```

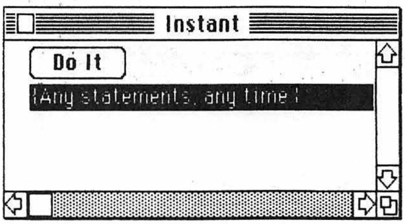
5 An Instant Cure for Premature Compilation

What's next

This section introduces the Instant window, an ideal place to experiment with new code before it goes into your program. New commands for drawing are shown in Instant examples and characteristics of the Drawing window are explored.

5.1 The instant window

Choose Instant from the Windows menu.



```
{Any statements, any time}
```

Pascal statements entered in the Instant window are performed immediately by clicking the Do It button. This contrasts with the Program window, which will only run a properly constructed Pascal program. As with the Program window, graphic and text statements of the Instant window output to the Drawing and Text windows respectively.

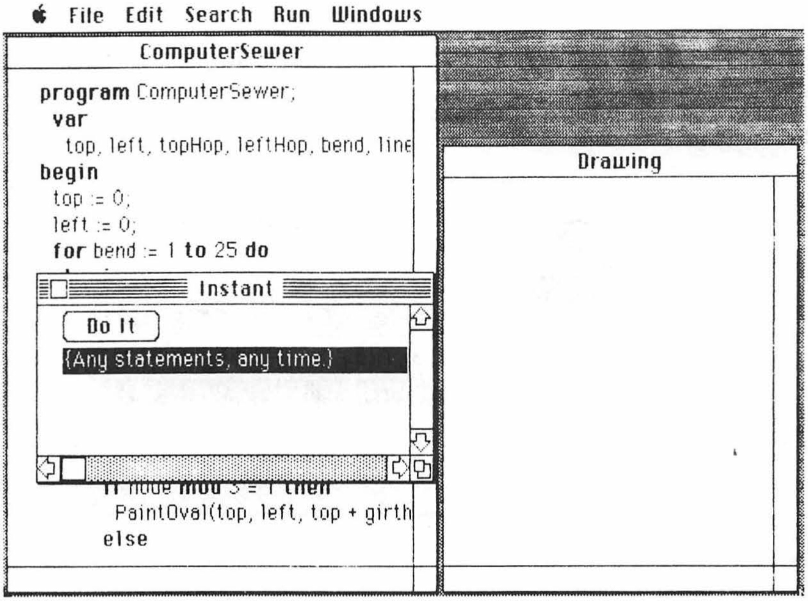
Proud as Macintosh programmers tend to be, Mr. Moss sincerely hopes he never sees the bumper sticker:

MacPascal Programmers Do It in Windows

5.2 Instant alterations on ComputerSewer

Beneath the Instant window should be the trusty, not yet rusty, ComputerSewer. If between the last chapter and this chapter you went on a muskrat safari in the Arkansas bush, bring ComputerSewer back on to the screen by clicking on its icon.

Arrange the windows on the screen to look like the illustration below.



```
program ComputerSewer;
 var
  top, left, topHop, leftHop, bend, line
begin
 top := 0;
 left := 0;
 for bend := 1 to 25 do
    if node mod 3 = 1 then
     PaintOval(top, left, top + girth
    else
```

```
{Any statements, any time.}
```

Run ComputerSewer. Notice that while the program is running, the only actions available to you are Pause and Halt. Once halted, you can resume activity with any Run command. The program, and its sewer, continues at the place it was halted.

If during a Halt the content of the Program window is changed in any manner, subsequent running of ComputerSewer, whether by Go, Go-Go, Step, or Step-Step, will clear the Drawing window and reset the program to begin anew.

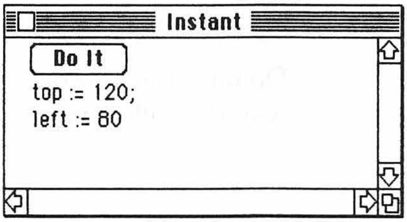
Unlike the Program window, the Instant window allows you to make changes inside a program during a Halt *without* resetting the program. The Drawing window will remain intact and any Run selection will resume the program at the place it was halted.

Now you are going to use the Instant window to change ComputerSewer in the middle of its construction.

Activate the Program window by placing the cursor anywhere in the window and clicking.

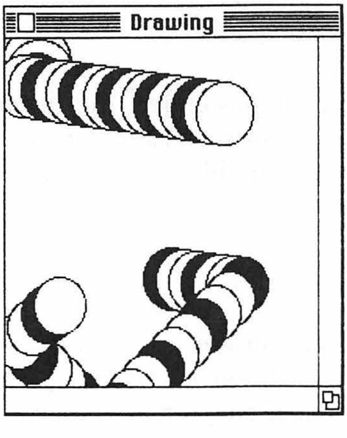
Choose Go from the Run menu, then Halt the program while the sewer is being built.

Select the Instant window and then type the following statements into it. Do not forget to include the semicolon.



```
top := 120;
left := 80
```

Click the Do It button, then choose Go again.

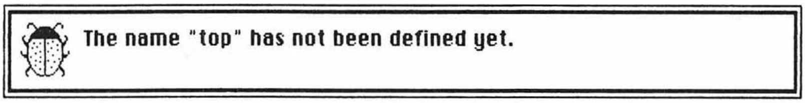


The ComputerSewer begins again, yet a different place from where it had stopped. The numbers you assigned to the words *top* and *left* in the Instant window became the new screen location assignments to the sewer. The program has adopted new screen location assignments from the Instant window.

top and *left* are names for program *variables.* (See Part 3 for an explanation of variables.) The values assigned to *top* and *left* are used in the Pascal drawing statements to determine the top and left screen locations for each sewer node. When *top* and *left*—or any of the sewer variables—are assigned new values in the Instant window, the program adopts these new values as the program resumes.

The coordinate system by which the values of *top* and *left* plot locations on the Macintosh screen will be discussed later in this chapter. For now, you can see that the Instant window lets you experiment with programming options in midcourse.

Try to click the Do It button now that ComputerSewer has ended. Can you guess why the following message appears?



```
The name "top" has not been defined yet.
```

When a program has ended, the Instant statements cannot identify variable names such as *top* and *left.* The Instant window can only Do It to variables of a program while the program is in midcourse.

5.3 Pinpointing rectangles

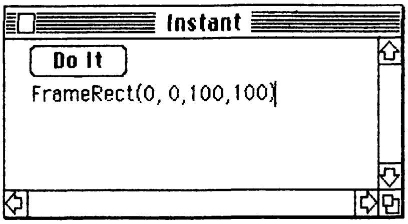
At this point, you are going to do some Instant work without the help of ComputerSewer. ComputerSewer will come in handy again in the next chapter, but it's goodbye for now.

Click in the Close box of the Program window. (Remember, the Close box is the small square in the top-left corner of each window.) Any window can be returned to the screen through the Windows menu.

Choose Reset from the Run menu to clear the Drawing window.

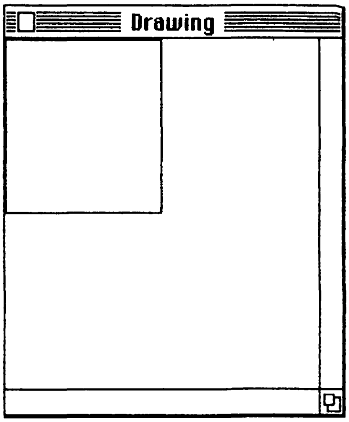
Clear the Instant window by selecting all the text, then pressing the *backspace* key. You could also clear selected text by using Cut or Clear from the Edit menu, but the *backspace* key is quicker.

Type the following statement into the Instant window.



```
FrameRect(0, 0,100,100)
```

Click the Instant window's Do It button. Your Instant command should draw a rectangle in the drawing window as shown below.



The four numbers in parentheses map the rectangle frame. Use the mouse to change (0, 0, 100, 100) to (0, 50, 75, 150).

Now Do It again. See the difference?

The Drawing window consists of a rectangle full of invisible dots, each represented in the computer's memory by two coordinate numbers (x, y). The top, leftmost dot is numbered (0, 0). As you address dots farther to the right—that is, horizontally—the first coordinate increases. As you address dots downward—or vertically—the second coordinate increases. For example, one dot to the right would be (1, 0). The first coordinate below the top-left coordinate would be (0, 1).

Sometimes while using the (x, y) notation, you will hear the horizontal coordinates referred to as positions along the x-axis, and vertical coordinates as positions along the y-axis. Point (x, y) would be located x dots to the right, and y dots down from the point (0, 0).

The visible dimensions of the Drawing window stretched to its maximum size are 497 dots across by 311 dots down, with the point (0, 0) at the top-left corner. At the bottom-right corner is point (497, 311).

A rectangle can be defined by its two opposing corners. The four numbers separated by commas in FrameRect are boundary coordinates corresponding to the rectangles top-left and bottom-right corners.

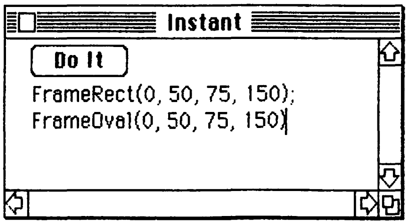
Using the coordinate notation described above, the new FrameRect command should look like this: FrameRect(50, 0)(150, 75). However, MacPascal requires that all four numbers be lumped together in the order top-left, bottom-right. As a result, the command looks like this: FrameRect(0, 50, 75, 150).

What happens if you drag the Drawing window to another location? Does the rectangle stay put, or move with the window? If the grid is stationary with respect to the Macintosh screen, the rectangle should not move. Yet if the grid is fixed solid in the Drawing window, both the grid numbers and any rectangle it contains should drag along with the window.

The Drawing window uses a local coordinate system. The numbered grid, and all drawings and text on that grid, adhere to the window, and don't give a hoot about any other window or the Macintosh screen as a whole.

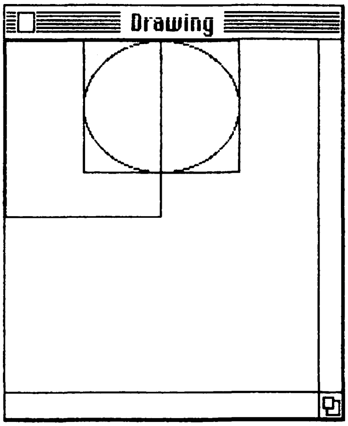
5.4 Ovals and lines, thick and thin

Insert a semicolon at the end of the first line and press *return* before adding this second line to the Instant window.

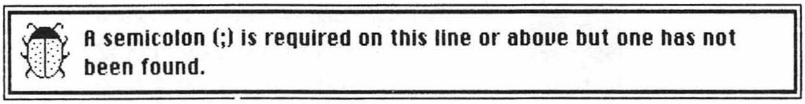


```
FrameRect(0, 50, 75, 150);
FrameOval(0, 50, 75, 150)
```

Click Do It to produce this drawing:



If you forget the semicolon after the first statement, you will get the thumbs down hand before the second statement and this bug message:



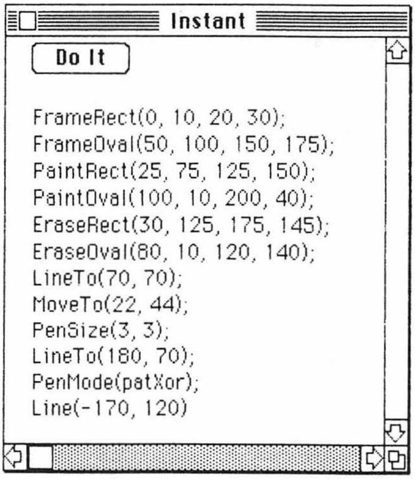
A semicolon (;) is required on this line or above but one has not been found.

The semicolon is a necessary evil in Pascal to separate statements. The only statements that do not require the line-ending semicolon are those preceding other words that punctuate Pascal, such as **begin, end,** and **until,** and those commencing an action such as **for..to..do, repeat..,** and **while..do.**

You can see that the Instant window is capable of doing multiple statements provided you separate them with semicolons.

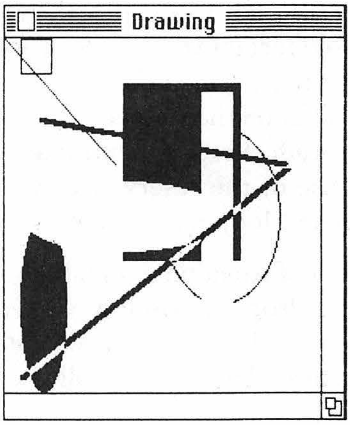
The coordinates of FrameOval define a rectangle in the same way as FrameRect. However, FrameOval draws an oval that fits just inside the specified rectangle.

Experiment individually with the statements in the following Instant window. Insert your own dimensions, large and small. Try a decimal number; try a negative number.



```
FrameRect(0, 10, 20, 30);
FrameOval(50, 100, 150, 175);
PaintRect(25, 75, 125, 150);
PaintOval(100, 10, 200, 40);
EraseRect(30, 125, 175, 145);
EraseOval(80, 10, 120, 140);
LineTo(70, 70);
MoveTo(22, 44);
PenSize(3, 3);
LineTo(180, 70);
PenMode(patXor);
Line(-170, 120)
```

Do It to produce the following drawing:



You may have discovered that if you draw to coordinates outside the visible region of the Drawing window, neither dragging nor resizing the window will succeed in recovering or exposing the "hidden" drawing.

Similarly, when you hide part of a drawing by dragging, resizing, or burying its window, the image cannot be recaptured. Unlike the Program window or the MacPaint program, MacPascal does not refresh drawings that have gone off the screen.

5.5 Structures in the instant window

At first glance, the Instant window might appear to be simply a second Program window. The difference lies in the ability to structure a program.

The Instant window has the single capability of performing a series of Pascal statements within or independent of a Pascal program. The Program window has the capabilities of structuring a program's statements into distinct blocks, naming the blocks, declaring and defining data within blocks, passing information between blocks, and executing blocks by calling their names.

The ability to structure a program makes programming code easier to create and understand. Pascal's popularity is due to its highly structured nature.

A note for the advanced Pascal programmer: You should be aware that the Instant window does not support its own *procedures, functions,* or the declaration of *variables* and *constants.* You can, however, call upon these structures as they appear in the Program window in midcourse of a program. The Instant window also allows the use of the reserved words **begin** and **end,** as well as conditional and looping statements.

MacPascal will format unacceptable structures in the Instant window like the Program window. However, after you click Do It, you will get the thumbs down sign at the first occurrence of the illegal structure. This will be followed by a bug message.

5.6 Prolonging the pleasure

The Instant window is ideal for experimenting with Pascal statements, formatting the Drawing and Text windows, and observing how statements and structures of your program are functioning. You can practice first, then install your tested code in the Program window. This saves you the time wasted by prematurely running an incomplete program.

Many a computer programmer knows the slow horror of premature compilation or even premature interpretation. (*Compilation* and *interpretation* are two different methods by which a computer language translates English into a machine-readable format. MacPascal uses the interpreted method.) It is doubtful the Macintosh will ever complain about your hastiness with a bug message such as this:

Not tonight, I have a headache.

Yet the fulfillment of a complete, uninterrupted, bug-free program run can be orgas . . . well, can make you feel good.

6 Observing: Different Pokes for Different Folks

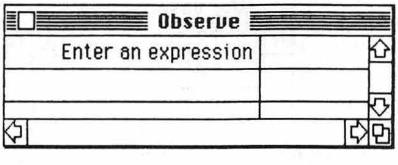
What's next

This chapter concludes Part 1 on the MacPascal operating environment. The Observe window displays the names and continually updated values of selected variables, and you will see how these values direct the construction of ComputerSewer.

6.1 The observe window

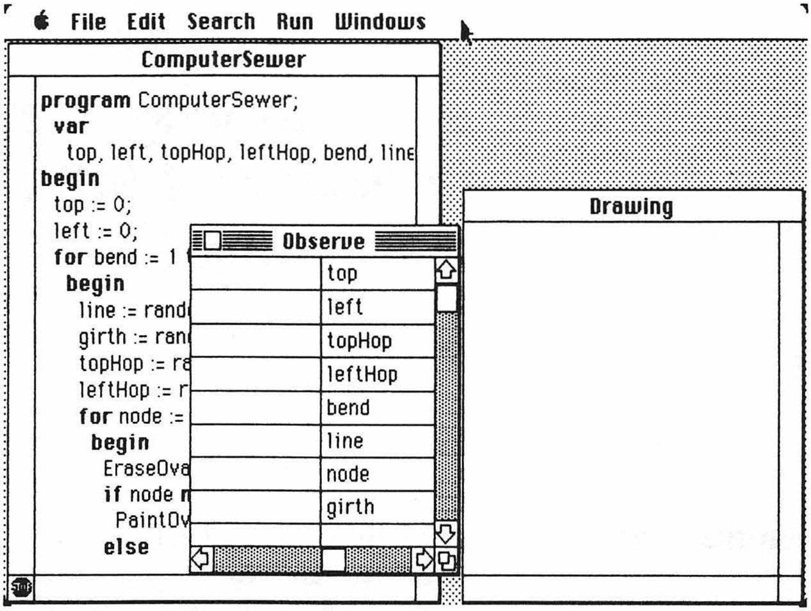
Experimenting with the Observe window will illustrate how a single name can be used to represent an entire series of values. The Observe window will also serve as a prime debugging device in your later programming efforts.

Close the Instant window from Chapter 5. Choose ComputerSewer from the Windows menu. Now choose Observe from the Windows menu.



Enter an expression

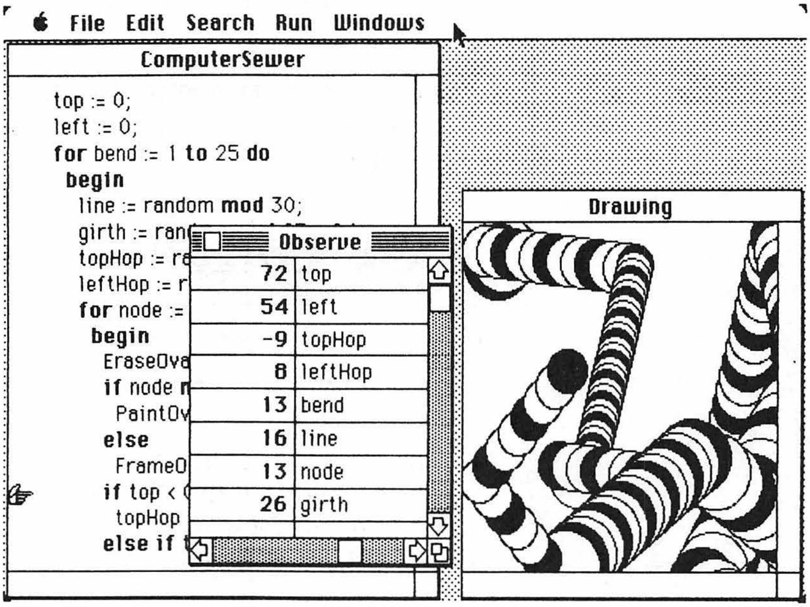
Position and resize the Observe window on top of the Program window. Following the illustration on the next page, type each word into the Observe window and press *return* to bring the insertion bar to the next line. Use the horizontal scroll bar to center the text in the Observe window.



```
program ComputerSewer;
 var
  top, left, topHop, leftHop, bend, line
begin
 top := 0;
 left := 0;
 for bend := 1
  begin
   line := rand
   girth := ran
   topHop := r
   leftHop := r
   for node :=
    begin
     EraseOva
     if node m
      PaintOv
     else
```

	top
	left
	topHop
	leftHop
	bend
	line
	node
	girth

Now select Step-Step from the Run menu. To the left of each box, numbers appear. The disk whirs, the numbers change, and ComputerSewer plods along with the pointing hand. Choose Halt from the Pause menu when you get bored.



```
 top := 0;
 left := 0;
 for bend := 1 to 25 do
  begin
   line := random mod 30;
   girth := ran
   topHop := r
   leftHop := r
   for node :=
    begin
     EraseOva
     if node m
      PaintOv
     else
      FrameO
☞    if top < (
      topHop
     else if
```

72	top
54	left
-9	topHop
8	leftHop
13	bend
16	line
13	node
26	girth

Exciting stuff, this Observe window. You may be thinking, “Ugghh, more number garbage,” but amazingly, Macintosh is able to decode this seeming nonsense into the home telephone numbers of Nastassja Kinski, Woody Allen, Princess Diana, Prince, and the cast of the “Benny Hill Show.”

Actually, you are looking at the changing values of your program’s *variables* as the program moves from line to line. A variable is a name given to a particular type of data that might vary as the program runs, hence the name *variable.*

The numbers you see in the Observe window change because some line or lines in the program code instruct the number to change. In sewer, the line *top := top + tophop* changes the value of the variable *top.* In English you might read this line, “Assign a new value to *top* equal to the current values of *top* plus *tophop.*”

The Observe window lets you examine both variables and expressions made up of variables, numbers, and arithmetic signs. If you are still unclear about *variables,* don’t worry: The sun and the moon are important too, but worrying doesn’t make them work better. Besides, Parts 2 and 3 are full of examples showing variables in action.

6.2 Observe at work

Programmers can name variables to describe their purpose. Whereas a cartoon animator might label a sequence of pictures “Road Runner drops anvil on Coyote,” a sewer programmer could label a sequence of oval locations *top.* Likewise, a sequence of different sewer thicknesses could be represented by a variable named *girth.*

The sewer is constructed from ovals with the top and leftmost part of the first oval drawn touching the top-left sides of the Drawing window. The variables *top* and *left* are initially set by the code *top := 0; left := 0.*

Choose Step-Step from the Run menu again. While ComputerSewer is running, pay attention to the Observe window. In particular, watch the values of *top* and *left* to see how they determine the location for the next node of sewer to be built. Click on Pause if you need more time to think things out.

Depending on how large you have resized the Drawing window, the sewer may have momentarily escaped off the bottom or right side of the screen. Or if you have resized the Drawing window to take up the entire screen, you will see that the sewer never draws onto the right half of the screen.

In both instances, the reason can be traced to the variables *top* and *left.* These variables grow as a result of the last two statements of ComputerSewer:

top := top + tophop;
left := left + lefthop

The Observe window shows these variables never grow much larger than 200. Since a fully opened Drawing window can accommodate coordinates 497 dots wide and 311 dots tall, the sewer at its farthest reaches will not come near the bottom-right corner. Similarly, a small Drawing window may not have room to show a sewer growing near its outer limits.

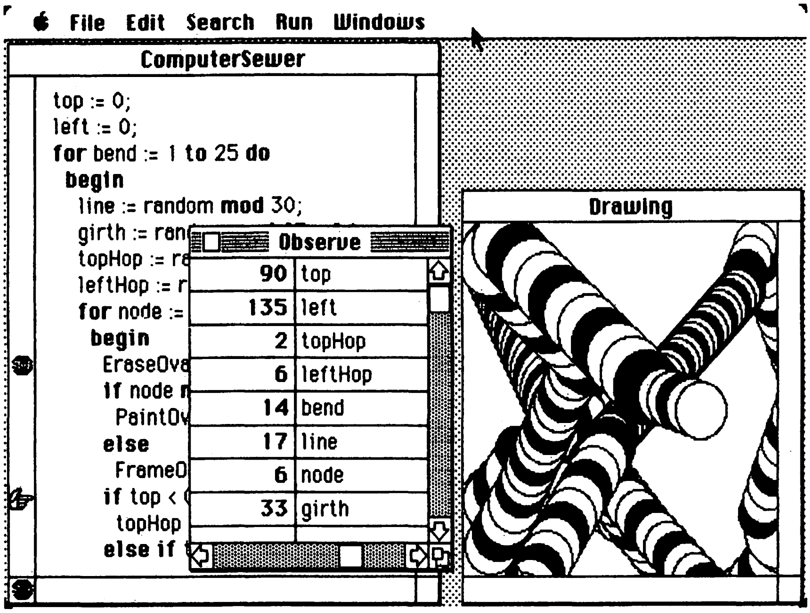
You should look at the program code and try to find why *top* and *left* do not extend much below 0 or over 200. Examine the **if..else** statements toward the bottom of the program. These statements catch *top* and *left* when they go below 0 or above 200, and force them to hop—using absolute values and the incrementing variables *tophop* and *lefthop*—in the opposite direction.

For example, the statement ***else if*** *top > 200* ***then*** checks to see if the value of *top* exceeds 200. If it does, then the following statement, *tophop := −abs(tophop),* is executed. *−abs* (short for *absolute value)* is Pascal notation which makes sure that *tophop* is assigned a nonpositive value. This ensures that the statement at the bottom of the program, *top := top + tophop,* will not allow *top* to grow any larger.

The letters *abs* are short for absolute value. Placed before a variable, *abs* will make certain the variable's value is not negative by removing any minus sign. *−abs* will make certain the variable's value is not positive by inserting a minus sign to all nonzero absolutes.

If *top = 203* and *tophop = 4,* then the execution of *tophop := −abs(tophop)* would change the value of *tophop* to *−4.* Now the execution of *top := top + tophop* will change the value of *top* to *199.*

Using Step-Step to observe variables becomes tedious. Inserting stop signs with Stops In enhances the power of the Observe window.



```
top := 0;
left := 0;
for bend := 1 to 25 do
 begin
  line := random mod 30;
  girth := ran
  topHop := r
  leftHop := r
  for node :=
   begin
    EraseOva
    if node r
     PaintOv
    else
     FrameO
    if top < (
     topHop
    else if
```

90	top
135	left
2	topHop
6	leftHop
14	bend
17	line
6	node
33	girth

1. Choose Stops In from the Run menu. Set a stop sign in the margin to the left of the line containing *EraseOval.* Remember, clicking the mouse in the margin will either set a new stop sign or, if one already exists, remove it.

2. Rather than using Go from the Run menu, hold down the shortcut *cloverleaf-g* keys to Go through a few loops.

3. Repeat step 2. You should notice in the Observe window that the values of *top* and *left* show where the sewer has stopped, and *tophop* and *lefthop* show the direction for the next segment.

4. Change the boundaries of the sewer by activating the Program window and then replacing the numbers 0 and 200 in

the four bottom **if** statements with −40 and 400, or for a claustrophobic mess, 10 and 60.

5. Repeat step 2 to watch the sewer grow with new boundaries.

6.3 Final rites for the sewer's Pascal mysteries

A few of ComputerSewer's Pascal statements have yet to be explained. Using the Observe window might help you to better understand these concepts.

Enlarge the Observe window to display all of the variables in the program. Each press of the *cloverleaf-g* keys will run the program until the next occurrence of the loop's stop sign. The sewer makes use of two loops, or repetitive structures. These are identified by the reserved words **for..to..do.**

The sequence of the loop *for bend := 1 to 25 do* could be read:

1. The variable *bend* shall be assigned the value of 1.
2. The program shall perform the statements between the following **begin** and its paired **end.**
3. *Bend* shall be incremented by one.
4. Sequence parts 2 and 3 shall be performed again and again until the value of *bend* has exceeded 25.

This loop repeats a series of instructions for every *bend* the sewer takes. Included in this loop are the random assignments that make ComputerSewer run differently each time Go is selected.

Knowing which **end** belongs to which **begin** is tougher than knowing your own **end** from your own **elbow.** Skip to Part 3 under **begin** if your elbow itches and you want to know where to scratch.

The second loop, *for node := 1 to line do,* is performed in every repetition of the *bend* loop. This loop repeats a series of instructions for every *node* in each straight line of sewer. It is in this second loop that the graphic commands *EraseOval, PaintOval,* and *FrameOval* actually draw onto the Drawing window.

A stop sign in the margin halts program execution and tells the Observe window to update its data. Step, Step-Step, and Go-Go also update. Go, alone without stops, will not provide a running tab of variable values.

With the descriptive names of ComputerSewer's variables, and their values illustrated in the Observe window, you may be able to piece together how the heck the program works.

*random **mod*** are reserved words that will assign to a variable a random number in the range including 0 and one less than the number following **mod.** *random **mod** 30* will assign a value of 0 to 29 to *line. random **mod** 19 - 9* assigns a number between −9 and 9 to the variables *tophop* and *lefthop.* Notice the subtraction occurs after the random number is determined.

Experiment by putting one or more stop signs elsewhere in the program. Try changing the numbers to put more *bends* in the sewer, or to make each *line* of sewer longer, or to extend the *hops* between each *node* of a sewer line. A simple change worth the effort is to make the sewer square. Replace the *Oval* with *Rect.*

If you can look at the Observe window and predict how the next segment of sewer will grow, you have reached a position of programming expertise deserving of the title Sewer King or Sewer Queen. Not everyone cares for aristocracy though, Mr. Moss among those. Thus, simply for getting this far in the text, here are some words of encouragement:

Congratulations, Sewer Punk

PART 2

Programming with Quickdraw and the Toolbox: The Language Guide

Each chapter in Part 2 begins with a sample program. The programs illustrate Quickdraw and the Toolbox, the user-friendly software built into every Macintosh.

You will see that Quickdraw and the Toolbox do much more than draw pictures. They can become a user's interface to the computer. Your programs will have the Macintosh qualities that make micemeat of programs run on any other computer.

Along the way, you will pick up the Pascal language. Line-by-line explanations tell how each program works. Each chapter adds a few new pieces to Pascal and the puzzle of computer programming.

By following the example programs, your first Pascal programs will:

- use the mouse to initiate graphics and answer questions
- manipulate windows
- send multistyled text to the printer
- store and retrieve data on disks
- connect your Pascal programs to MacPaint and MacWrite

And that's just the first few chapters. By using Quickdraw and the Toolbox the structure and grammar of the Pascal language will unfold.

Later chapters show you how to:

- create simple tones and full-frequency sound
- perform error-checking input routines
- format dollars and cents for business applications
- keep track of time with the system clock
- start a graphics-based filing system

7 Zen Pascal: Everything You Know Is Wrong

What's next

This chapter shows you the shell of a Pascal program. With this shell you can begin writing your own programs. You will see that programs can be written without months of monastic discipline, deprivation, dispossession or, worse yet, attending school.

■ **Topics:**

program

; {semicolons}

var {variables}

begin

end

7.1 Program AirHead;

```
program AirHead;
begin
end.
```

7.2 A savage journey into the head of Pascal

All Pascal programs must begin with the reserved word **program,** followed by a name chosen by the programmer, and a line-ending semicolon. This is what you see in the first line of the program above:

program AirHead;

Pascal doesn't give a fig about lines or carriage returns, but semicolons are taken seriously. Semicolons separate statements.

All Pascal programs must contain at least one **begin** and one **end.** The last word of every Pascal program is **end** and it is always followed by a period, like this: **end.**

That is all that Pascal requires. In summary, a Pascal program must have no less than:

1. The reserved words **program, begin** and **end.**
2. A made-up name chosen by the programmer.
3. A semicolon after the made-up name.
4. A period after the reserved word **end.**
5. All of the above inserted in the same order as shown in **program** Airhead.

7.3 Declarations

Most programs have a declaration section. **program** AirHead has none. If **program** AirHead used any variables, they would be declared beneath the line ***program** AirHead;* and above the line **begin.**

Just for the heck of it, take a look at **program** AirHead2 with a variable. Notice the reserved word **var. var** says to a program, "Hey, pal, listed below are the names and types of variables that I want to use in this program."

```
program AirHead2;
 var
  thoughts : integer;
begin
 thoughts := 0
end.
```

Throughout Part 2, Pascal terms will be introduced without pausing to define and give examples of each new word. The continuity of seeing a program presented without interruption can help you understand programs better than a series of definitions. Besides, Mr. Moss busted his chops devising Part 3, a Pascal-Quickdraw-

Toolbox dictionary; so if you want to know more about **var** and variables, flip to the alphabetical listing in Part 3.

The variables listed below **var** might be thought of as a grocery list. The variable name is equivalent to an item's brand name. The variable type is equivalent to an item's kind. Name and type are separated by a colon. Every listing, even the last, ends with a semicolon.

Here is an example of a possible variable declaration:

```
var
  rockyRoad : ice cream;
  snickers : candy;
  grandmaDora : cookies;
```

The sample programs in the next few chapters will use Pascal's predefined types such as *integer, real, char, string, text,* and *array.* Here is a sample variable declaration using predefined types:

```
var
  chocolateFix : integer;
  sugarIntake : real;
  bloatedFeeling : string;
  constipation : packed array[1..26] of char;
```

You can read up on each of Pascal's predefined types in Part 3. For now you should know that:

1. Each variable needs to be declared as to its **type.**
2. The declaration occurs following the reserved word **var,** in the format:

```
var
  variableName1 : typeA;
  variableName2 : typeB;
```

7.4 The main body

You might have guessed by now that a computer program is nothing more than a list of instructions. When you write a computer program, you are giving orders in the same manner that a parent gives orders to a child. A parent might say: "Go to the supermarket. Buy a loaf of whole wheat bread and a half gallon of low-fat milk. Here is three dollars, bring back the change."

The order of the instructions becomes important. You cannot buy the bread and milk until you go to the supermarket. You cannot get the change until you buy the bread and milk.

The order of a Pascal program is also important. You have to know where your instructions **begin** and **end.** And guess what: Pascal does not execute a program in a strict, linear, top-to-bottom order.

In what order does Pascal perform instructions? The answer is illustrated by following the *pointing hand.* Drooling teachers and leaden books can try to explain program flow till hell serves Häagen Dazs, but not as well as the pointing hand of MacPascal's Step commands.

Whenever the order of program execution becomes confusing, use the Step command to inch your way through the program. The Step command will perform your program, putting output in the Text and Drawing windows the same as the Run command, yet at a line-by-line pace.

Step through **program** AirHead. Though the program does absolutely nothing, you can see that it is an actual program. The main body of AirHead consists of two words: **begin** and **end.**

The main body has no special name or reserved word to state its presence. The main body requires only Pascal's delimiters, **begin** and **end.** These reserved words serve throughout the Pascal program as bookends to hold together two or more instructions as a unit. Only in the main body do they serve the added purpose of beginning and ending a Pascal program.

The main body resides at the end of the program, following all other parts of a program. In forthcoming chapters, you will be introduced to the building blocks of a program called **procedures** and **functions.** A block is simply a group of instructions that have been lumped together under an assigned name in order to accomplish a task.

In **program** AirHead, running the program with Step makes the pointing hand point to **begin.** The next Step points to **end.**

Remember, the pointing hand points to the instruction that will be performed next. One more call to Step will end the program.

All programs begin with the first **begin** in the main body. All programs end with the last **end** in the main body. The instructions between **begin** and **end** of the main body instigate all other activity.

From this chapter on, you will be instructing a computer to "go to the supermarket, buy bread and milk, and return with the change." If the computer comes back with ice cream and cookies, you will just have to shrug and say four of the most satisfying words in the English language:

At least I tried.

8 Of Mice and Mountains

What's next

This program flips a rectangular coin when the cursor arrow is on top of the coin. Move the mouse off the coin and you get heads if the coin is black or tails if the coin is white. The key command for a program to read the mouse is *getMouse,* a predefined procedure from the Macintosh Toolbox.

■ Topics

integer

frameRect

moveTo

writeDraw

repeat..until

button

getMouse

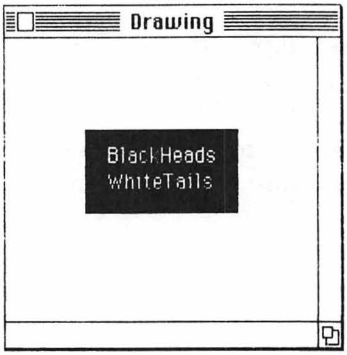
if..then

and

invertRect

8.1 Program CoinFlip 1;

```
program CoinFlip1;
 var
  x, y : integer;
begin
 frameRect(50, 50, 100, 150);
 moveTo(65, 70);
 writeDraw('BlackHeads');
 moveTo(65, 85);
 writeDraw('WhiteTails');
 repeat
  getMouse(x, y);
  if (x >= 50) and (x <= 150) and (y >= 50) and (y <= 100) then
   invertRect(50, 50, 100, 150)
 until button
end.
```



BlackHeads
WhiteTails

8.2 Declarations

Two variables, *x* and *y*, are declared to be of type *integer* under the reserved word **var**. The variable *x* will hold the horizontal coordinate and *y* the vertical coordinate of the tip of the mouse's cursor arrow.

8.3 Main

The instruction *frameRect(50, 50, 100, 150);* draws a rectangular outline in the Drawing window. The numbers in parentheses, known as *parameters,* correspond to the top, left, bottom and right sides of the rectangle, in that order.

moveTo(65, 70); puts the Quickdraw pen into position for text to be written. (65, 70) is a coordinate point of the Drawing window and is contained in the rectangle drawn above.

You never actually see the Quickdraw pen. It is an invisible tool that positions and draws Macintosh graphics. What you *can* see is the ink from the Quickdraw pen when you execute a drawing or *writeDraw* command.

writeDraw('BlackHeads'); inserts the text of its string parameter beginning at the pen location. The string parameter is the letters between the single quotation marks.

moveTo(65, 85) repositions the pen location to a point directly below where 'BlackHeads' was written.

writeDraw('WhiteTails'); inserts its text at the new pen location.

Now a **repeat** loop begins. All the instructions following **repeat** up to the bold-lettered, similarly-indented **until** are contained in the loop.

The loop will be repeated until the boolean Toolbox function named *button* is true. Boolean implies a true or false condition.

A Toolbox function is a group of instructions defined inside the MacPascal language. The Toolbox function *button* tests the status of the mouse button and returns a true or false value wherever the word *button* occurs. *button* returns the value *true* when, and only when, the mouse button is pressed.

getMouse(x, y) is the first instruction in the **repeat** loop.

getMouse(x, y) is a Toolbox procedure that reads the integer coordinates of the mouse's cursor, and returns the horizontal coordinate to its first variable parameter and the vertical coordinate to its second variable parameter. In program CoinFlip 1, the parameter variables are named *x* and *y*.

A Toolbox procedure, similar to a Toolbox function, is a group of instructions defined in the MacPascal language. The instructions are performed by calling the procedure's name. In the case of *getMouse(x, y)*, the Toolbox procedure assigns the integer coordinates of the mouse to the parameter variables *x* and *y*.

Following *getMouse* is an **if..then** statement. If the condition following **if** is true, then the action following **then** is performed. Otherwise, nothing is done and the program continues at the statement beyond the **then** action, which in CoinFlip 1 is: ***until button.***

The **if** condition is: *(x >= 50)* **and** *(x <= 150)* **and** *(y >= 50)* **and** *(y <= 100).*

The reserved word **and** means that the conditions on both sides of **and** must be *true* in order for the entire condition to be *true.*

Since *x* and *y* represent coordinate integers that were assigned in the *getMouse(x, y)* procedure, the **if..then** statement says: if the mouse's coordinates are within the specified rectangular area **then** perform the action *invertRect(50, 50, 100, 150).*

invertRect is a Quickdraw procedure that inverts the dots within the parameter rectangle. If they were black they become white; if they were white they become black. The rectangle's parameters are the same as the parameters of the *frameRect* procedure.

The area specified by the **if** condition happens to be the same area enclosed by the *frameRect* box drawn in the first statement. Matching the *x* and *y* coordinates with the top, left, bottom, and right parameters of *frameRect* deserves a few minutes of your time.

The **repeat** loop ends at: ***until button.*** Assuming the button is not being pressed, the *getMouse* procedure is performed again, the location of the mouse is checked by the **if** statement, and if the cursor is within the rectangle, the rectangle is inverted.

The best way to see how fast MacPascal performs this loop is to run the program with the mouse pointing inside the rectangle. Watch how fast the rectangle changes back and forth from white to black. Point the mouse outside the rectangle to stop the rectangle from inverting.

Are you quick enough to make the coin flip always stop at BlackHeads?

Press the mouse button to exit the **repeat** loop and end the program.

The mention of BlackHeads makes Mr. Moss think of his adolescence, where hope and frustration stand out years after the pimples have passed. Hope turns to resolve, frustration becomes sadness; Mr. Moss's adolescence lasted too long. You have a computer language to learn, but damned if Mr. Moss is going to let a chapter pass without a kindred message on priorities and patience.

Priorities? Patience? Lofty aims for a book on Pascal. Like the blind man playing Impeccable Warrior at the video arcade: "Blind man," says Mr. Moss's girlfriend, "How in heaven do you play these machines?"

"Very poorly," he answers. Quarters exhausted, he steps back, bumps her. "But I'm improving."

9 Procedures and True Boolean Confessions

What's next

This chapter's program performs the same trick as last chapter's program. But two things are different. First, the main body has been shortened by adding procedures to do its work. Second, the use of a boolean expression makes the program more readable.

Topics

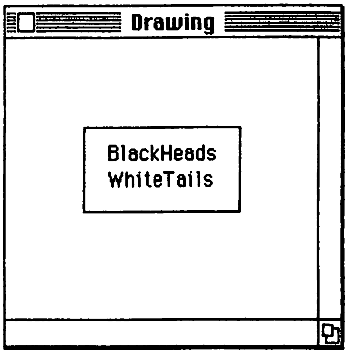
var {within procedures}

boolean

procedure

9.1 Program CoinFlip2;

```
program CoinFlip2;
 procedure frame;
 begin
  frameRect(50, 50, 100, 150);
  moveTo(65, 70);
  writeDraw('BlackHeads');
  moveTo(65, 85);
  writeDraw('WhiteTails')
 end;
 procedure flash;
  var
   x, y : integer;
   inBox : boolean;
 begin
  getMouse(x, y);
  inBox := (x >= 50) and (x <= 150) and (y >= 50) and (y <= 100);
  if inBox then
   invertRect(50, 50, 100, 150)
 end;
begin
 frame;
 repeat
  flash
 until button
end.
```



```
BlackHeads
WhiteTails
```

9.2 Declarations

The same two variables, *x* and *y*, are declared to be of type *integer* under the reserved word **var**, but now the declaration appears lower down in the program code, beneath the line: ***procedure** flash;*.

The declaration of a variable within a procedure or function block is only effective within the group of statements that compose the block. Outside the procedure or function, the variable is unknown.

The advantages of declaring a variable *locally* within a block rather than *globally* under the **program** heading, as in last chapter's CoinFlip 1, are twofold. First, memory space is freed when the variable's block is not being used. Second, the same variable name can be used other places in the program, helping to prevent a clutter of variable names all doing the same thing.

A third variable, *inBox,* of type *boolean* has been declared. Whereas type *integer* represents a whole number in the range −32,767 to 32,767, type *boolean* represents one of only two values: true or false. The value of *inBox,* once assigned, must be either true or false. Such is the nature of all boolean variables.

9.3 Main

The main body of CoinFlip2 consists of only five words, excluding the required **begin** and **end**. The ***repeat..until button*** instruction was introduced in the last chapter. The two new words, *frame* and *flash,* are procedure calls.

A procedure call is an instruction to direct program execution to a procedure block listed above. The instruction *frame;* directs program flow to *frameRect(50, 50, 100, 150);*, the first instruction of **procedure** frame.

Remember from Chapter 7, a **procedure** block is a group of instructions that have been clumped together under an assigned name in order to accomplish a task. The way to execute the instructions of a **procedure** is to call that **procedure** by its assigned name.

Since you must call a **procedure** (or **function**) in order to perform its statements, you can see why all programs start from the main body. The main body has no title and resides at the bottom of the program code. The main body begins with the first occurrence of the reserved word **begin** that is not a part of any other block.

The main body of a program should not be cluttered with obscure instructions. Treat the main body of your program like your favorite girlfriend or boyfriend—give the clearest, most concise phrasing of your intentions, and orderly calls to the procedures and functions you believe will satisfy both your needs.

And if the two of you are planning to spend the night discussing existential metaphysics, at least have the decency to stick the crummy code in an out-of-the-way **procedure.**

Whenever you are in doubt about how a program jumps from one instruction to another, MacPascal's pointing hand, available by running a program with Step or Step-Step, shows the path of program execution. Assuming you have entered CoinFlip2 into the Program window (as you should be doing with all of Part 2's programs), choose Step from the Run menu a few times—or the shortcut *cloverleaf-S*—and watch the pointing hand move from *frame;* to *frameRect(50, 50, 100, 150);*.

9.4 Procedure frame;

Program flow goes from the main body and procedure call *frame;* to *frameRect(50, 50, 100, 150);* and on through the five instructions of **procedure** frame.

The commands within **procedure** frame are the same as those in the previous chapter. The task of the procedure is to frame the rectangular coins and write the strings 'Blackheads' and 'WhiteTails' within the rectangles.

When the last command, *writeDraw('WhiteTails'),* has been performed, program execution continues where it left off before **procedure** frame was called. The next instruction is **repeat** in the main body.

9.5 Procedure flash;

The **repeat** loop now contains one command, the procedure call *flash.* **procedure** flash will be repeated until the mouse button is pressed making the *button* function return true.

procedure flash performs the mouse reading and interpreting tasks. The *getMouse* procedure remains unchanged. The **if** condition has been changed slightly.

Instead of inserting the long **and..and** condition between **if** and **then,** the same within-the-given-area condition is assigned to a boolean variable with the descriptive name of *inBox.*

The variable *inBox* is assigned a true or false value by the line: *inBox := (x >= 50)* ***and*** *(x <= 150)* ***and*** *(y >= 50)* ***and*** *(y <= 100);*. Both sides of this assignment are boolean expressions. The reserved word **and** joins all four of the expressions in parentheses into a single boolean expression that has a true value only if all the individual expressions are true.

The variable *inBox* becomes true if mouse's coordinate point (x, y) is contained within the rectangle's boundaries. Otherwise *inBox* is assigned the value of false.

There are two important aspects of assignment statements:

1. Assignments are made using the colon/equals sign (:=). The equals sign alone (=) is used for tests of equality.
2. The type of the variable must match the type of its assignment. Boolean variables cannot be assigned integers or anything else but a boolean value.

If you forget either of these aspects of assignment statements, MacPascal will assuredly remind you with outlined text or a bug message.

The boolean variable *inBox* makes the **if..then** statement much easier to understand. If the mouse is within the box, then *inBox* is true and the *invertRect* command is performed. If the mouse is not within the box, then *inBox* is false and the *invertRect* command is skipped over.

Following the **if..then** action, **procedure** flash ends and program flow drops back to the main body. However, if the button is not being pressed, the procedure is immediately called again because *flash* is the only statement in the **repeat** loop.

A press of the mouse button drops out of the **repeat** loop and ends the program.

10 Rect and Point: Types Not for Your Sister

What's next

This program performs exactly the same trick as programs in the last two chapters. But one important new concept is introduced. Rectangles and points will not be identified by a series of integers. They will be assigned names. These names will become variables of the predefined Quickdraw types *rect* and *point.*

■ Topics

point

rect

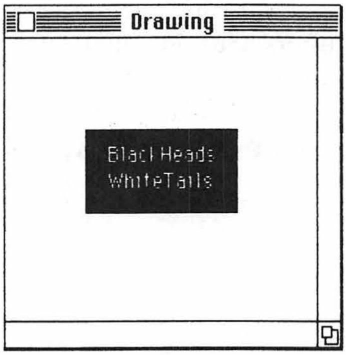
setPt

setRect

ptInRect

10.1 Program CoinFlip3;

```
program CoinFlip3;
 var
  r : rect;
 procedure frame;
 begin
  setRect(r, 50, 50, 150, 100);
  frameRect(r);
  moveTo(65, 70);
  writeDraw('BlackHeads');
  moveTo(65, 85);
  writeDraw('WhiteTails')
 end;
 procedure flash;
  var
    x, y : integer;
    pt : point;
 begin
  getMouse(x, y);
  setPt(pt, x, y);
  if ptInRect(pt, r) then
   invertRect(r)
 end;
begin
 frame;
 repeat
  flash
 until button
end.
```



10.2 Declarations

As in the last chapter, variables *x* and *y* are declared to be of type *integer* under the reserved word **var** within **procedure** flash. The boolean variable *inBox* has been omitted.

Also declared in *flash* is *pt,* a variable of type *point.* Using a variable of type *point* will replace the need for integer parameters each time a point's coordinates are referenced.

The easiest way to remember about parameters is to think of the *frameRect* command. The four integers in *frameRect(50, 50, 100, 150)* are a parameter list, and each number is a parameter dictating on which coordinate axis to draw a side of the rectangle (top, left, bottom, right).

A variable of type *point* requires only two parameters. You will see below that the Quickdraw procedure *setPt* assigns the variable *pt* to two integer variables by listing them consecutively in its parameter list: *setPt(pt, x, y).*

A global declaration of *r*, a variable of type *rect,* requires four parameters. The variable name *r*, assigned its value by *setRect(r, 50, 50, 150, 100),* replaces the need for integer parameters each time the rectangle is referenced.

point and *rect* are predefined in Quickdraw. As such, these types can be used with the same ease as Pascal's predefined types like *integer* and *string.* Because Quickdraw, the Toolbox, and MacPascal are all computer language tools, the only time you will need to differentiate among them is when you try Pascal programming on a computer other than Macintosh.

10.3 Main

The main body of CoinFlip3 is identical to Chapter 9's CoinFlip2.

The adage *If It Ain't Broke, Don't Fix It* applies wisely for mainframe COBOL crankers, though MacPascal artists would more likely be overheard saying:

Hell, my date won't be here for an hour.
In that time I can rewrite this sucker from scratch.

10.4 Procedure frame;

The first instruction performed after *frame;* in the main body is *setRect(r, 50, 50, 150, 100);* in **procedure** frame. This Quickdraw procedure assigns the variable in the first parameter, in this case *r*, with the four side parameter integers: *left, top, right,* and *bottom,* respectively.

Sorry, fans, but the order of the side parameters of *setRect(r, 50, 50, 150, 100)* is *left, top, right,* and *bottom.* This is different from the order of integers placed directly in the parameter list of any shape-drawing procedure such as *frameRect.* In *frameRect(50, 50,* *100, 150),* the order of the side parameters is: *top, left, bottom,* and *right.*

frameRect(r); draws the identical rectangular outline of *frameRect(50, 50, 100, 150)* used in Chapters 8 and 9. The single variable *r* of type *rect* represents all four integers.

The remaining instructions of **procedure** frame have not changed. When the last command, *writeDraw('WhiteTails'),* has been performed, program execution continues where it left off before **procedure** frame was called. The next instruction is **repeat** in the main body.

10.5 Procedure flash;

procedure flash still performs the mouse reading and interpreting tasks. The *getMouse* procedure remains unchanged.

setPt(pt, x, y); assigns the variable of its first parameter, in this case *pt,* with the two integers of its second and third parameters. The latter parameters are *x* and *y*, the horizontal and vertical coordinates returned from the *getMouse(x, y)* procedure. As a result, the variable *pt* of type *point* contains the two integers indicating the position of the mouse cursor.

The next instruction is a beauty. *ptInRect(pt, r),* as the condition of the **if..then** statement, is a Quickdraw boolean (true/false) function that determines if point *pt* is contained in rectangle *r.*

If *pt* is in *r, ptInRect* returns true, and the instruction *invertRect(r)* is performed. If *pt* is not in *r*, the function returns false, and the action is not done.

The *ptInRect* function will not accept integers directly inserted into its parameter list. It accepts only a type *point* and a type *rect* parameter. So even though the shape-drawing procedures accept integer parameters, the use of *setPt* and *setRect* to create *point* and *rect* types is strongly recommended. Besides, it is easier to write *invertRect(r)* than it would be to write *invertRect(50, 50, 100, 150).*

This chapter is the last you will see of the rectangular coin flip. You no longer need to scrounge in your pants pockets for a nickel to make life's important decisions. Of course, if you enjoy reaching into your pants pockets, don't let technology put a crimp in your fun.

11 Babies and the Input/Output Function

What's next

This program asks a question and waits for the user to type an answer. The program will call a **function** to read the keyboard, then issue an appropriate response. Also, the Text window will be resized from within the program.

Topics

string

setTextRect

showText

write

writeln

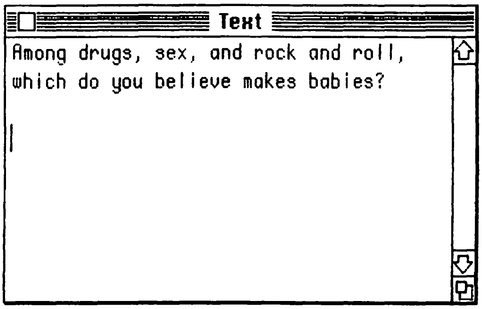
if..then..else

function

readln

11.1 Program NotTheStork;

```
program NotTheStork;
 var
  txWindow : rect;
 function getAnswer : string[20];
  var
   s : string[20];
 begin
  readln(s);
  getAnswer := s
  end;  {end getAnswer}
  procedure question;
  begin
    write('Among drugs, sex, and rock and roll, ');
    writeln('which do you believe makes babies?');
    writeln;
    if getAnswer = 'sex' then
      writeln('Mr. Moss quessed this, too.')
    else
      begin
        write('Either your Walkman is on too tight or ');
        writeln('the pipe between your ears needs cleaning.')
      end
  end;  {end question}
begin
  setRect(txWindow, 100, 100, 400, 250);
  setTextRect(txWindow);
  showText;
  question
end.  {end NotTheStork}
```



```
Among drugs, sex, and rock and roll,
which do you believe makes babies?
```

11.2 Declarations

txWindow, a variable of type *rect,* is declared globally. Remember, a global declaration occurs beneath the **program** heading, and global variables can be used throughout the program. A local declaration occurs beneath a **procedure** or **function** heading, and can be used only within its block.

The Quickdraw type *rect* was discussed in the last chapter. Variables of this type contain the integer coordinates that define a rectangle. *txWindow* will be used to set, then display, a resized Text window.

The only other variable declared in NotTheStork is *s*, a variable of type *string.* The string variable *s* will hold the answer typed by the user.

A string is one or more characters held together by single quotation marks. Any character can be inside a string except the single quotation mark, which is reserved for identifying the string's beginning and end.

The variable declaration of a string should also contain an integer in brackets stating the maximum number of characters the string can contain. *s : **string**{20}* declares that the variable *s* is a string of not more than 20 characters.

The purpose of stating the size of the string in brackets is to conserve memory space. If the number is omitted, a default size of 255 is assumed, and the variable will take up 255 characters worth of space even if it is only 2 characters long.

11.3 Main

The main body of NotTheStork consists of four procedure calls. The first three, all Quickdraw procedures, concern the Text window. The last call performs the question and answer task.

You should be familiar with the *setRect* procedure from the last chapter. In *setRect(txWindow, 100, 100, 400, 250);*, coordinates are assigned to the type *rect* variable *txWindow.*

setTextRect(txWindow); is the Quickdraw procedure call that sets the size of—but does not draw—the Text window. The size is determined by its type *rect* parameter. The four coordinates assigned to the variable *txWindow* by the *setRect* command are passed on as a single parameter of *setTextRect.*

showText; is the Quickdraw procedure call that draws the Text window with whatever dimensions are currently assigned to the Text window rectangle.

question is the call to **procedure** question in the program code. Program execution continues at the first instruction of this procedure.

11.4 Procedure question;

The first instruction of **procedure** question is: *write('Among drugs, sex, and rock and roll, ');*.

The characters between the single quotation marks of Pascal's write command are written onto the Text window just as they appear between the single quotation marks. Characters between single quotation marks are called a *literal string,* or simply, a *literal.*

When creating a literal, be careful not to use the single quotation mark as part of your literal. When you need to use an apostrophe, use the slanted apostrophe mark that is on the top-left key of your keyboard.

The second instruction, *writeln(‘which do you believe makes babies?’);* also displays the characters between single quotation marks on the Text window. Both *write* and *writeln* require their parameters to be parenthesized.

The difference between *write* and *writeln* is that *writeln* also sends an invisible end-of-line character after the last literal character. The consequence of this end-of-line character is that any subsequent characters sent to the Text window will start a new line.

The third instruction, *writeln;,* has no characters, parentheses, or single quotation marks. This command sends the end-of-line character for the purpose of beginning a new line without displaying any characters. The result is a blank line on the Text window.

You don’t have to worry whether each line of characters you send to the Text window will spill ouside the boundaries of the window. The Text window will format text into new lines whenever the edge of the window has been reached. If text extends beyond the bottom of the window, the vertical scroll bar becomes active, and the beginning text lines scroll off the top of the window. Use the scroll bar or resize the window to recapture the hidden text.

Unlike the Drawing window, the Text window does not lose its contents when it is covered over by another window or resizing, nor will the Text window write invisibly outside the window’s coordinates.

The remaining instructions of **procedure** question are part of an **if..then..else** statement. The boolean condition *getAnswer = 'sex'* will be explained in the next section. If the boolean condition of the **if** statement is evaluated as true, then the literal 'Mr. Moss guessed this, too.' is written to the Text window. If the boolean condition is false, the two literal strings following **else** are written to the Text window.

You should notice that the *write* and *writeln* commands following **else** are enclosed by a **begin..end.** The purpose of bracketing—or gluing together—the *write* and *writeln* commands between **begin** and **end** is so both commands will be performed only if the **else** condition is true. Without the **begin..end** brackets, the **else** condition would apply only to the *write* command, and the latter *writeln* command would be performed regardless of whether the **else** condition was true or not.

11.5 Function getAnswer : string[20];

In section 11.4 above, the boolean condition *getAnswer = 'sex'* of the **if..then** statement needed more explanation. Since *getAnswer* was never declared as a variable under any **var** heading, there must be some other way for *getAnswer* to be assigned a string value such as 'sex.'

The block of code titled **function** *getAnswer : **string{20}*** performs this task.

A **function** resembles a procedure in that it contains its own declarations, its own statements, and a name by which it is called from elsewhere in the program. A **function** differs from a procedure in that a function must return a single value to the function call.

You might think of a function's name as a variable that assigns itself a value through its own block of statements. Like a variable, the function's name has a declared type. A colon (:) separates the function name from its type in the title.

This is the function title in NotTheStork: ***function** getAnswer : **string**{20};*. **function** getAnswer will return a value of type **string**[20], a string of not more than 20 characters. The name *getAnswer* serves both as a call to the function and as the variablelike name of the returned value.

Now you can see that the **if..then** condition *if getAnswer = 'sex' then* is both calling **function** getAnswer and returning with a string value that will be compared with the literal 'sex.'

function getAnswer contains only two statements. The first, *readln(s),* causes the program to wait until input is received from the Macintosh keyboard.

The input is recognized as complete when the end-of-line *return* key is pressed. When *return* has been pressed, the characters input from the keyboard are assigned to the parameter variable, in this case, the string variable *s*.

The second instruction, *getAnswer := s,* assigns the value of the string variable *s* to the string type result of **function** getAnswer. Just like a variable, a function must have a value assigned to it.

Remember, assignments—setting one value equal to another—are done with the colon and equals sign (:=). The equals sign (=) alone is used for a boolean (true/false) test of equality.

Use Step to go through NotTheStork line by line. This will reaffirm how the name *getAnswer* in the **if..then** statement directs the pointing hand to the **function** getAnswer block. After you execute the *readln(s)* command you will still have to type an answer in from the keyboard.

For a little twerp of a program, this is a long and somewhat difficult chapter. Do not be misled:

Programming is not simple, though many programmers are.

Nor is programming always fun. The bug messages, outlined code, and glitched-out glop can get on your nerves. Still, you should be taking the programs in this book and twisting them around so that they do what you want them to do. At the least, change the window sizes and string literals. If you abhor drugs and rock and roll, insert tofu and poetry. If you abhor sex, insert . . . nah, just skip to the next chapter.

12 The Mouse Yes, the Keyboard No

What's next

Do not be frightened by the long program in this chapter. Almost every instruction you have seen before in some manner. Best of all, you will see how to use the mouse to answer yes or no questions—all within a subroutine that you can reuse in any of your own programs.

■ Topics

hideAll

setDrawingRect

showDrawing

if..then..else {nested}

penSize

textSize

textFont

frameRoundRect

function {boolean}

or

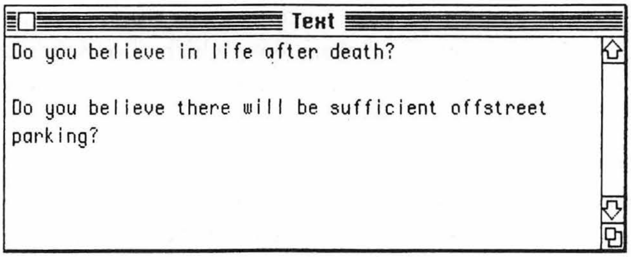
not

sysBeep

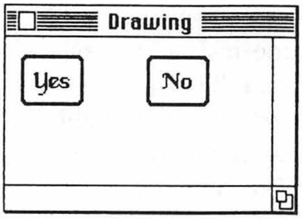
while..do

12.1 Program LifeAfterDeath;

```
program LifeAfterDeath;
 procedure windows;
  var
   txWindow, drWindow : rect;
 begin
  hideAll;
  setRect(txWindow, 60, 60, 455, 180);
  setRect(drWindow, 40, 225, 225, 325);
  setTextRect(txWindow);
  setDrawingRect(drWindow);
  showText;
  showDrawing
 end;    {end windows}
 function getAnswer : boolean;
  var
   x, y : integer;
   inBox : boolean;
   pt : point;
   okay, notOkay : rect;
 begin
  penSize(2, 2);
  textSize(14);
  textFont(5);
  moveTo(18, 30);
  writeDraw('Yes');
  setRect(Okay, 10, 10, 50, 40);
  frameRoundRect(Okay, 9, 9);
  moveTo(100, 30);
  writeDraw('No');
  setRect(NotOkay, 90, 10, 130, 40);
  frameRoundRect(NotOkay, 9, 9);
  inBox := false;
  repeat
   getMouse(x, y);
   setPt(pt, x, y);
   if button then
    begin
     inBox := ptInRect(pt, okay) or ptInRect(pt, notOkay);
     if not inBox then
      sysBeep(15)
    end
  until inBox;
  getAnswer := ptInRect(pt, okay);
  while button do                      {wait for mouse up}
   ;
 end;   {end getAnswer}
 procedure questions;
 begin
  writeln('Do you believe in life after death?');
  writeln;
  if getAnswer then
   begin
    writeln('Do you believe there will be sufficient offstreet parking?');
    writeln;
    if getAnswer then
     write('Write Mr. Moss about his mosquito-free Everglades land.')
    else
     write('Breathe exhaust, heretic.')
   end
  else
   write('Bite on a bug, heretic.')
 end;  {end questions}
begin
 windows;
 questions
end.  {end LifeAfterDeath}
```



```
Do you believe in life after death?

Do you believe there will be sufficient offstreet
parking?
```



12.2 Declarations

txWindow and *drWindow,* both variables of type *rect,* are declared in **procedure** windows. Just as *txWindow* was used in the last chapter to set, then display a resized Text window, *drWindow* will do the same for the Drawing window.

The six other variables in LifeAfterDeath are declared in **function** getAnswer. The types of each of these variables should be familiar to you from previous chapters. Four of the six variables even retain the same names. Only two *rect* variables, *okay* and *notOkay,* are new.

12.3 Main

The main body of LifeAfterDeath consists of two procedure calls. Both call procedures listed in the program code.

Procedures and functions should be placed in a program *before* any statements that call it. This presents no problems for the main body of a program, which is always placed at the end of a program. However, when a procedure or function calls another procedure or function, care must be exercized that the block being called precedes the statement that calls it. In other words, statements cannot call *forward* to procedures and functions.

There is an exception to this rule—a special declaration called a *forward declaration*—which allows statements to precede the blocks they call, but such declarations are not implemented in this book.

12.4 Procedure windows;

From *windows;,* program execution jumps to *hideAll;,* the first instruction of **procedure** windows. *hideAll* is a Quickdraw procedure that clears the Macintosh screen of everything but the menu bar.

The next six instructions are all Quickdraw procedure calls performing the same task as the main body of last chapter's NotTheStork program. The *setRect* commands assign coordinates to *txWindow* and *drWindow,* variables of type *rect. setTextRect* and *setDrawingRect* assign those dimensions to the Text and Drawing windows. *showText* and *showDrawing* display those windows on the Macintosh screen.

procedure windows gives LifeAfterDeath a clean, sharp look, uncluttered by program code and overlapping windows.

12.5 Procedure questions;

The second instruction of the main body is a call to **procedure** questions. This procedure consists of *write* and *writeln* statements, and two **if..then..else** statements, nested such that the second **if** statement begins before the first one has ended.

The *write* and *writeln* statements were explained in the last chapter. The only reason some of the string literals in NotTheStork and LifeAfterDeath are presented in consecutive *write* and *writeln* statements rather than a single long *writeln* statement is to prevent a long string literal from running off the right margin of the Program window. Such a string would still work okay, but is not as pleasant to view.

You should also be familiar with the **if..then..else** statements. The action performed after **then** or **else** will be either a single statement or a multiple statement block bracketed by **begin** and **end**.

As the MacPascal indentations suggest, each **end** is paired with the nearest preceding unpaired **begin**. A similar rule exists for **if..then..else** statements. Each **else** is paired with the nearest preceding unended **if**. Look at the examples in Part 3 under **begin** and **if** for more information on this topic.

procedure questions does little more than write the string literals that make up the questions and answers in LifeAfterDeath. The only additional task of the procedure is to call **function** getAnswer to provide the boolean branching condition of the two **if..then..else** statements. The true or false value returned by **function** getAnswer will determine whether the **if** statement will branch to the action following **then** or the action following **else**.

Note that the second, more deeply indented, *if getAnswer **then*** instruction is part of the action taken only if the first call to *getAnswer* returns with a true value. The second **if..then..else** instruction is skipped over when the first call to *getAnswer* returns false and program flow branches to the paired **else** action, *writeln('Bite on a bug, heretic.')*.

This chapter really needs a sidetrack here—something that has absolutely nothing to do with Pascal or programming. Mr. Moss wants to talk about his girlfriend. Seems that Mr. Moss always wants to talk about his girlfriend. Tonight at around six-thirty she is going to come over his house and they are going to take a long walk around the neighborhood. Mr. Moss's girlfriend knows the names of plants and flowers and trees and all sorts of good information. Mr. Moss considers himself lucky that he can, at least, spot poison oak.

The sunsets have been beautiful lately. Mr. Moss and his girlfriend have been going to the marina to watch the sky change colors. They hold hands or walk with their arms around each other's back as if they were both seventeen years old.

Summer is almost over. The cool autumn evenings ought to be nice. Mr. Moss's girlfriend is kind, generous, and sweet. Whatever darkness there might have been earlier in his life, Mr. Moss looks forward to tonight.

12.6 Function getAnswer: boolean;

function getAnswer works much like its counterpart of the same name in the last chapter. Its purpose is to return an answer to the calling procedure in order for a branching decision to be made.

In the last chapter, **function** getAnswer returned a string, and that string was compared to the literal 'sex' to determine if the boolean condition was true or false.

In LifeAfterDeath, **function** getAnswer returns a boolean result, so no comparison is necessary. The function name alone—***if*** *getAnswer* ***then***—calls **function** getAnswer and returns the boolean result necessary to make a branching decision.

The declarations and statements in the revised *getAnswer* do much more than its predecessor. Rather than demand that the user answer a question by typing on the keyboard, a more elegant method—one which befits the elegance of Macintosh—is to point the mouse at an answer box and click within the box to select the answer.

This is exactly what **function** getAnswer does. Two rounded-corner boxes are drawn in the Drawing window—one with *yes,* the other with *no,* printed inside the boxes. The first eleven instructions—every one a Quickdraw procedure call—perform this task.

See Part 3 to learn more about any of these Quickdraw calls. You have used most of these calls before, including *setRect,* the procedure that defines a variable of type *rect. frameRoundRect* works like *frameRect* except that two additional parameters are required to determine the roundness of the rectangle's rounded corners.

The remainder of **function** getAnswer resembles the program CoinFlip3, where the *ptlnRect* procedure is used to determine whether the mouse is pointing within a specified rectangle. Here, the same procedure is used to determine if the button has been pushed while the mouse is pointing in the *yes* box, the *no* box, or neither box.

Probably the most mysterious of all the instructions in **function** getAnswer is: *inBox := false.* It is the first command after the eleven Quickdraw calls. This command is an *initialization.* The boolean variable *inBox* is assigned, that is, *initialized* with, a value of false so that the loop

repeat
.. {action}
until *inBox*

continues cycling until a command within the loop changes the value of *inBox* to true.

Until a variable is specifically assigned, or initialized with, a value, it is considered undefined. An undefined variable can take unpredictable values, and its use will, more often than not, cause unwanted results.

The **repeat..until** loop performs these tasks:

1. Reads the coordinates of the mouse.
2. Assigns these coordinates to variable *pt* of type *point.*
3. Calls the Toolbox *button* function to see if the button has been pressed.
4. If the button has been pressed, branches to the block following **then** and
5. assigns *inBox* a value. *Inbox* is true if *pt* is located in either rectangle *okay* **or** *notOkay,* otherwise *inBox* is false and the **if not** action causes the *sysBeep* noise.

6. If button has not been pressed or if *inBox* was assigned false in step 5, the boolean condition *inBox* following **until** remains false and the loop is repeated starting at step 1.

or joins two boolean expressions to a single boolean value. The value is true if either or both expressions are true, otherwise the value is false.

not negates the boolean value of whatever expression it precedes. If the reserved words **or** or **not** are confusing to you, read more about them in Part 3.

When the mouse button has been pressed while pointing in the *yes* or *no* box, the **repeat** loop is exited. The next instruction, *getAnswer := ptInRect(pt, okay),* assigned **function** getAnswer its result value—true if point *pt* is contained in the *yes* rectangle *okay,* false if not.

The last instruction of **function** getAnswer, ***while button do;,*** is a delay mechanism necessary only because the Macintosh computer is so fast in recognizing mouse input that a program needs to allow for the slow touch of the user.

while button do; is a loop that performs no statements. The semicolon signifies the end of the loop, and since there are no statements between **do** and the semicolon, it is an empty loop. Nonetheless, if the mouse button is being pressed, the computer (think of the pointing hand of MacPascal) cycles around and around from the semicolon to ***while button do,*** repeating the loop until the *button* function reads that the mouse button is up and returns a value of false.

The **while..do** loop works nearly the same as the **repeat..until** loop. The difference is that the **while** loop interprets its boolean condition before performing any action, whereas the **repeat** loop always performs its action at least once before it interprets its boolean condition.

When the ***while button do*** loop is exited, the program flow returns from **function** getAnswer back to **procedure** questions. The boolean result of the function determines which branch of the **if** loop is performed and, consequently, which text is written on the screen.

13 The Printer Prints

What's next

This short program sends text to be printed on—what else?—the printer. Mr. Moss has been using an Apple Imagewriter printer, though there is a good chance this program would work the same on any Macintosh-compatible printer.

Topics

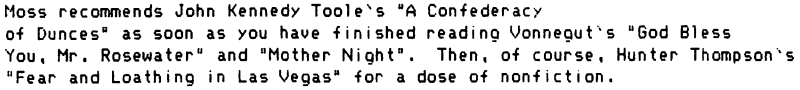
text {files}

rewrite

writeln {to files}

13.1 Program WordsInPrint;

```
program WordsInPrint;
 var
  f : text;
  s1, s2, s3, s4 : string[60];
begin
 s1 := 'Mr. Moss recommends John Kennedy Toole`s "A Confederacy ';
 s2 := 'of Dunces" as soon as ';
 s3 := 'you have finished reading Vonnegut`s "God Bless ';
 s4 := 'You, Mr. Rosewater" and "Mother Night".  ';
 rewrite(f, 'PRINTER:');
 write(f, chr(27), 'c', chr(27), 'p');
 writeln(f);
 writeln(f, s1);
 writeln(f, s2, s3);
 writeln(f, s4, 'Then, of course, Hunter Thompson`s ');
 writeln(f, '"Fear and Loathing in Las Vegas" for a dose of nonfiction.')
end.
```



```
Moss recommends John Kennedy Toole's "A Confederacy
of Dunces" as soon as you have finished reading Vonnegut's "God Bless
You, Mr. Rosewater" and "Mother Night". Then, of course, Hunter Thompson's
"Fear and Loathing in Las Vegas" for a dose of nonfiction.
```

13.2 Declarations

WordsInPrint introduces a new variable type called *text.* The variable *f* is declared to be of type *text.*

Pascal does not recognize printers (or disk drives) directly. Pascal *does* recognize variables and types. So the best method of using a printer from Pascal is twofold:

1. Declare a variable to be of the *text* type.
2. Open the variable as a file organizer to the printer.

You never need to assign characters to the *text* variable *f.* Instead, you will use *f* as a file organizer that directs characters line by line from the computer to the printer.

s1, s2, s3, and *s4* are string variables whose space allotment has been set to 60 characters. The characters in these strings will be sent to the printer via the file organizer variable *f.*

13.3 Main

This program is short and direct, so there is no need to break up tasks into procedures or functions. All statements are contained in the main body of the program, bracketed by a single **begin** and **end.**

The four string variables are assigned their strings by the first four statements. The string variables could have been named Manny, Moe, Jack, and Curly, but *s1, s2, s3,* and *s4* was easier.

The fifth instruction is: *rewrite(f, 'PRINTER:');*. This Pascal procedure opens the connection between the file organizer *f* and the external device named 'Printer:'.

Three rules of printing:

1. The printer can only be activated by the file command *rewrite.*
2. The proper name for the printer must appear as shown—with colon and single quotation marks—though the letters of *'printer:'* can be upper or lower case.

3. The file organizer (variable) must be of type *text,* and be included as the first parameter of rewrite.

The remaining six instructions of WordsInPrint are *write* or *writeln* commands, all using *f*, the file variable, as the first parameter.

write(f,chr(27),'c',chr(27),'p'); is an awful-looking instruction that performs a useful printer task. The command sends control codes to the printer that set the Imagewriter to a proportionally-spaced pica print type.

The first parameter, *f*, lets the *write* command know that all subsequent parameters are to be sent to the rewrite-designated **file**, in this case, the printer. The control codes specified by the cryptic *chr(27)* and a character in single quotation marks send nonprinting information to the printer that regulates the printer hardware. The next chapter presents more information on directing the printer with control codes.

writeln(f); sends a line feed to the printer. The single parameter, file organizer *f*, sends its information to the printer. Since there are no text and no control codes, the only information sent is the line feed implicit in the *writeln* command.

As you saw in the two previous chapters, a *writeln* command with no parameters sent a line feed to the Macintosh screen. The blank line makes adjoining text easier to read. Sending a line feed to the printer also performs the handy task of straightening the paper before any printing is done.

writeln(f, s1); sends the string *s1* to the printer. If the file variable *f* had not been included, the string would have been sent to the Macintosh screen.

Take a moment to think through the concept of a file organizer. First, under var, you declared a variable *f* of type *text.* Then, you created (opened) the connection between *f* and the printer with *rewrite(f,'PRINTER:');*. Now, in the *writeln* commands, you are using *f* as a file organizer to route strings to the printer.

writeln(f, s2, s3); sends both strings to the printer. This shows you can combine more than one string parameter in a single *writeln*(or *write*) command, as long as you separate them with a comma. The line feed of *writeln* occurs after the last string is printed, not after each string.

The last two instructions show that string literals can also be sent to the printer. The string *s4* is printed, followed by the literal between single quotation marks. The final *writeln* command sends the second half of the literal. The *writeln* command works the same as when it is sending data to the Macintosh screen, except now the file organizer, inserted as the first parameter, directs output to the printer instead.

Besides Toole, Vonnegut, and Thompson, authors near the abyss worth checking out are: Kate Braverman, Brett Singer, William Kotzwinkle, Nathanael West—and one who has gone over—R. A. Lafferty.

14 Printing in Style Without the Alligator

What's next

Here is another program that makes use of the printer. You will experiment with control codes to see eleven different print styles, in addition to picking up some key Pascal pointers.

■ Topics

const

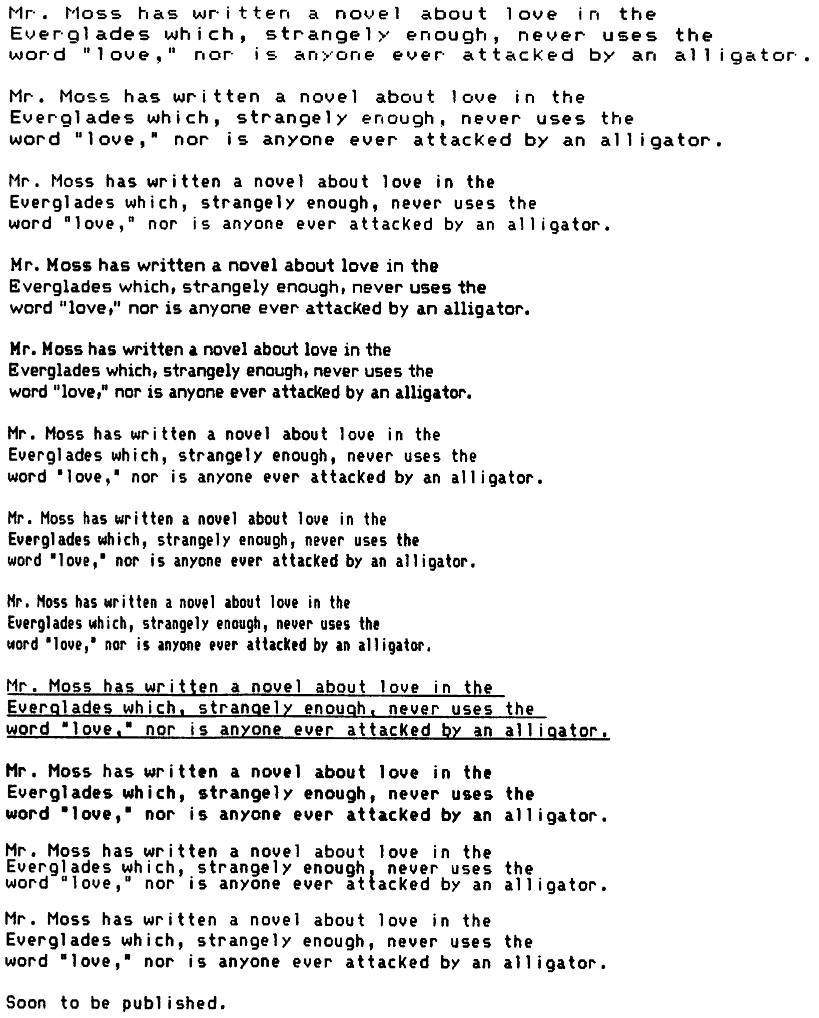
for..to..to

chr

case..of

14.1 Program PrintStyles;

```
program PrintStyles;
 var
  s1, s2, s3 : string[70];
 procedure assignLines;
 begin
  s1 := 'Mr. Moss has written a novel about love in the ';
  s2 := 'Everglades which, strangely enough, never uses the ';
  s3 := 'word "love," nor is anyone ever attacked by an alligator.'
 end;
 procedure printLines;
  const
   esc = 27;
  var
   i, code : integer;
   f : text;
 begin
  rewrite(f, 'printer:');
  for i := 1 to 12 do
   begin
    write(f, chr(esc), 'c');
    case i of
     1 :
      code := 110;
     2 :
      code := 78;
     3 :
      code := 69;
     4 :
      code := 112;
     5 :
      code := 80;
     6 :
      code := 101;
     7 :
      code := 113;
     8 :
      code := 81;
     9 :
      code := 88;
     10 :
      code := 33;
     11 :
      code := 66;
     12 :
      code := 99
    end;    {end case}
    write(f, chr(esc), chr(code));
    writeln(f);
    writeln(f, s1);
    writeln(f, s2);
    writeln(f, s3)
   end;    {end for loop}
  writeln(f);
  writeln(f, 'Soon to be published.')
 end;
begin
 assignLines;
 printLines
end.
```



```
Mr. Moss has written a novel about love in the
Everglades which, strangely enough, never uses the
word "love," nor is anyone ever attacked by an alligator.

Mr. Moss has written a novel about love in the
Everglades which, strangely enough, never uses the
word "love," nor is anyone ever attacked by an alligator.

Mr. Moss has written a novel about love in the
Everglades which, strangely enough, never uses the
word "love," nor is anyone ever attacked by an alligator.

Mr. Moss has written a novel about love in the
Everglades which, strangely enough, never uses the
word "love," nor is anyone ever attacked by an alligator.

Mr. Moss has written a novel about love in the
Everglades which, strangely enough, never uses the
word "love," nor is anyone ever attacked by an alligator.

Mr. Moss has written a novel about love in the
Everglades which, strangely enough, never uses the
word "love," nor is anyone ever attacked by an alligator.

Mr. Moss has written a novel about love in the
Everglades which, strangely enough, never uses the
word "love," nor is anyone ever attacked by an alligator.

Mr. Moss has written a novel about love in the
Everglades which, strangely enough, never uses the
word "love," nor is anyone ever attacked by an alligator.

Mr. Moss has written a novel about love in the
Everglades which, strangely enough, never uses the
word "love," nor is anyone ever attacked by an alligator.

Mr. Moss has written a novel about love in the
Everglades which, strangely enough, never uses the
word "love," nor is anyone ever attacked by an alligator.

Mr. Moss has written a novel about love in the
Everglades which, strangely enough, never uses the
word "love," nor is anyone ever attacked by an alligator.

Mr. Moss has written a novel about love in the
Everglades which, strangely enough, never uses the
word "love," nor is anyone ever attacked by an alligator.

Soon to be published.
```

14.2 Declarations and definitions

You have seen all of the **var** types used in previous chapters. Integer and string types should be old hat by now. The variable *f* of type *text* was explained in detail in the last chapter. *f* is the file organizer that directs output to the printer.

Some variables are declared locally in **procedure** printLines because they are only going to be used in **procedure** printLines. However, the string variables, *s1, s2,* and *s3,* are used in both of the program procedures. So rather than declare these variables twice, a single global declaration beneath the **program** heading makes them available for use anywhere in the program.

Beneath the heading of **procedure** printLines, and above the title **var**, is a new section of a Pascal program titled **const**. Beneath **const** is the definition: *esc = 27;*.

const is short for *constant.* A constant is similar to a variable except its value can never change. With a constant, you equate a name with a value. Later, you can substitute that name for the value in the program code.

Unlike variable assignment statements that use the colon/equals sign (:=) for notation, a constant definition uses the equals sign (=) alone. The **const** definition equates a name with a value. Each definition under **const** is followed by a semicolon.

The use of constants (defined by **const**) adds clarity to your program. Names are more apt to be descriptive and self-documenting than values.

By defining under **const** *esc = 27;*, each occurrence of the number 27 in **procedure** printLines can be substituted with the letters *esc.* As a result, the control code *chr(esc)* is more easily understood as the *escape character* than would be *chr(27).*

Another important advantage of using constants is the ease with which you can revise Pascal constants. For example, if after you have written a program, you discover that the correct *escape code* number is 32 instead of 27 (it's not), you would have to revise only the **const** definition. Yet if you had the number 27 written throughout your code, you would have to edit every occurrence of 27 to 32.

The definition **const**, like the declaration **var**, can be inserted globally beneath the **program** heading, or locally beneath a **procedure** or **function** heading. The former allows the constant to be used anywhere in the program, whereas the local definition can conserve memory space and enhance the building-block nature of a Pascal program.

14.3 Main

The main body of PrintStyles calls the two procedures listed above. Notice that the names of the procedures tell you what the procedures will do.

If you write a program, and then forget about it for a couple days, the descriptive names of constants, variables, procedures, and functions, help immensely in trying to figure out what the heck you have done.

You think you will remember how a program works, but you won't. Biologists say that in each person, millions of brain cells die every minute; and a visit to your neighborhood users group might convince you that programmers lose more than most. Then, of course, there is the word of Mr. Moss who says:

> You will forget your Pascal techniques far more readily than those used in the bedroom.

14.4 Procedure assignLines;

procedure assignLines contains three assignment statements, giving strings values to the string variables *s1, s2,* and *s3*. Since there is nothing new to examine in this procedure, take a moment to review the use of semicolons in a procedure.

The procedure name and the **end** of a procedure are always followed by semicolons. Because **begin** and **end** serve as brackets for Pascal instructions, neither **begin** nor the statement immediately preceding **end** requires a semicolon.

The text of the three strings in PrintStyles might have you wondering whether Mr. Moss's novel concerns computers. For the most part, no; however there is a short, dreamy sequence in which a programmer ingests a program and spends the next eight hours playing an adventure game from the inside out.

14.5 Procedure printLines;

procedure printLines performs a task very similar to the program in the last chapter. The variable *f* of type *text* works as a file organizer to send strings to the printer. The command that opens the connection between *f* and the printer is: *rewrite(f, 'PRINTER:');*.

The *writeln* commands at the end of **procedure** printLines are nearly identical to those in the last chapter. The file organizer *f* as the first parameter directs the strings to be written on the printer rather than the screen.

To examine how printLines works first look at the program's output. If you have a printer and have typed PrintStyles into the Program window, run the program. In this way you can have a paper copy of the different styles.

The first thing you will notice as you look at PrintStyle's output is that the same text is repeated twelve times. Not surprisingly, the line of code: *for i := 1 to 12 do* is responsible.

Similar to **repeat..until** and **while..do, for..to..do** is a looping mechanism. Loops allow a program to run the same instructions more than once without rewriting the code for each repetition. The method by which you want the loop to exit determines which looping mechanism is appropriate.

The **repeat** and **while** loops evaluate a *boolean* condition to determine when to exit. The **for..to..do** loop uses integers to count out an exact number of repetitions. When the last repetition has been completed, the loop is exited.

The format for the **for..to..do** loop requires that a variable of an ordered type such as *integer* or *char* be used as a counter. Then, expressions of that same type are inserted on both sides of the reserved word **to** to establish the counter's beginning and end points.

The sequence goes like this:

1. The counter variable is assigned the beginning value.
2. The instruction(s) following **do** is (are) performed.
3. The counter is automatically incremented by one and checked against the loop's endpoint integer.
4. Unless the endpoint has been exceeded, sequence parts 2 and 3 are repeated. If the endpoint has been exceeded, the loop is exited.

MacPascal requires that the variable used as counter must be declared at the same level, that is, locally, to the **for..to..do** loop. An attempt to use a global variable as the counter will produce an error message.

The instructions contained within the loop of **for..to..do** are bracketed by **begin** and **end**. Whenever the action of a **for** loop contains more than one instruction, the multiple statements must be bracketed.

You should pay special attention to just where the **for..to..do** loop ends. A curly bracket command {end for loop} will help you.

The first instruction of the **for..to..do** loop is *write(f, chr(esc),'c');*. This sends a control code to the printer that sets the printer to its standard setting (12 characters to an inch, pica type).

Remember from the **const** definition that *esc* equates to the integer 27. The parameter *chr(27)* is a Pascal function that converts numbers to their assigned character equivalents. *chr(27)* assigns an invisible character (one not represented on the Macintosh keyboard) to the printer, which in conjunction with the third parameter, instructs the printer to adopt a certain print style.

The third parameter of the *write* statement is the literal *'c.'* When preceded by the invisible *escape*[chr(27)] character, the literal 'c' tells the printer to reset the print style to its original settings. The literal 'c' could have been written as *chr(99)* because 99 is the decimal equivalent of the character 'c.'

Information about printer control codes, including many not illustrated in this chapter, can be found in Appendix D of Part 3. The *Imagewriter User's Manual* offers further detail about the operation of the printer. More information about the **chr** function can be found in Part 3.

The next instruction of PrintStyles is: *case i of.* This is a Pascal structure that closely resembles the **if..then** conditional structure. The **case** structure examines the variable *i* between the reserved words **case** and **of**, then selects the appropriate action from the **case** list that follows.

For example, if the value of *i* equals 6, then the action labeled 6 in the **case** list is performed. In that instance, the variable *code* would be assigned the value 101.

Since the **case** selector, the variable *i*, is the same variable used as the counter in the **for..to..do** loop, every **case** option will be performed as the **for** loop makes its twelve repetitions. The variable *code* will, one at a time, be assigned each of the values listed in the **case** list.

The purpose of assigning *code* these integer values becomes apparent in the first instruction following the **case** list. But first, take notice that the **case** list must end with an **end;**.

The instruction *write(f,chr(esc),chr(code));* sends to the printer twelve different control codes—one for each value of *code* that the **case** statement has assigned on the twelve passes through the **for..to..do** loop.

writeln statements send a line feed and the three strings to the printer in each repetition of the loop. This is why the same three strings are repeated twelve times in PrintStyles' output.

Remember, the different styles occur because the **case** statement has assigned a new value to *code* on each pass, and the variable *code* is then used as the invisible control code that sets the printer mechanism.

The final line of **procedure** printLines sends the literal 'Soon to be published.' to the printer. Until then, Mr. Moss suggests lots of sunsets, long walks, fiber in your diet, and bringing out a kindness that surprises even you.

15 Filing Away Regrets on a Disk

What's next

This little program shows you how to store information on a disk. In Part 1 you saw how to save the *program code* onto a disk, but here you will see how to save data *created within* a program. The data you save will be represented by its own MacPascal desktop icon under a name chosen by the program user. The next chapter's program will show you how to open the datafile and display its contents.

■ Topics

array

newFileName

rewrite {to disk files}

close

15.1 Program WriteRegrets

```
program WriteRegrets;
 var
  regrets : array[0..4] of string[75];
 procedure toDisk;
  var
   i : integer;
   dataName : string[75];
   regretFile : text;
 begin
  dataName := newFileName('type in  new file name');
  rewrite(regretFile, dataName);
  for i := 0 to 4 do
   writeln(regretFile, regrets[i]);
  close(regretFile)
 end;
```

```
begin
 regrets[0] := 'The only regrets in the life of Mr. Moss--thus far:';
 regrets[1] := 'Being shy with Nedra at age sixteen.';
 regrets[2] := 'Being shy with Linda at age eighteen.';
 regrets[3] := 'Being shy with Twila at age twenty-five.';
 regrets[4] := 'Being careless with Rebecca at age twenty-eight.';
 toDisk
end.
```

15.2 Declarations

You should be familiar with three of the four variables declared in **program** WriteRegrets. Again you will be using a variable of type *text.* Like the file organizer *f* used in the last two chapters, the variable *regretFile* is also a file organizer, though *regretFile* will be used to organize a disk file instead of the printer.

The new **var** declaration is: *regrets:* ***array****{0..4}* ***of string****{75};*. You know that *string{75}* is a series of up to 75 characters notated between single quotation marks. ***array****{0..4}* ***of string****{75}* is a fancy, shortcut method of creating five different strings, each with up to 75 of its own characters. The numbers in brackets *{0..4}* indicate the range of the array, in this case, five elements numbered 0, 1, 2, 3, and 4.

Each of these five strings has the name *regrets.* The way to tell one string from another is by indicating its element number immediately after the name *regrets. regrets{0}* contains the first string of the array. *regrets{1}* is the second string. *regrets{4}* is the last. Remember, each one of these strings can be assigned its own characters between single quotation marks.

Look at the five string assignment statements in the main body of **program** WriteRegrets. Notice that the array element *regrets{0}* has an assignment distinct from *regrets{1},* and so on. By declaring the variable *regrets* as an ***array****{0..4}* ***of string****{75},* you have at your access the equivalent of five new variables.

You might want to think of an array as a line of mailboxes on a post office wall. They are numbered consecutively, and they are all of an identical type, but each contains its own private mail.

15.3 Main

The main body consists of five string assignments to the array variable *regrets* and a procedure call. As you can see from the text of the five strings, there are three *shy's* and one *careless.* Behind both is much sadness.

Much of what Mr. Moss says derives from his regrets. For young people playing with their first romances, courage and caution are difficult to balance. In less poetic terms—being lonely sucks, and so does a broken heart.

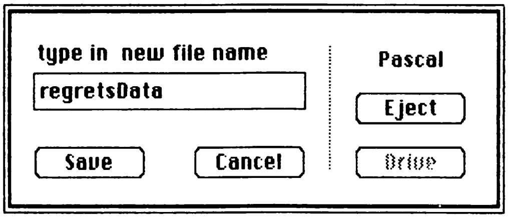
The courage to touch and the wisdom to be careful are more elusive than programming skills. If you don't know how to touch—try, because getting older doesn't make it any easier. If you know how to touch, make sure you also know about kindness and gentleness and how not to make babies.

And if you *are* going to mess around—even for the first time—especially for the first time—do it with someone you really, really like.

15.4 Procedure toDisk;

The first instruction makes use of the powerful Macintosh Toolbox procedure *newFileName.* The call to *newFileName* will cause a dialog box to appear on the screen prompting the user to type in a file name for the datafile to be created on disk. Whatever name is typed will be returned by the function as a string value.

The instruction *dataName := newFileName('type in new file name');* assigns the function's returning string to the variable *dataName.* Here is an example of a dialog box after the user has typed in a datafile name.



type in new file name

regretsData

The parameter of the function *newFileName* is a string literal. You will see the purpose of this literal when you run the program:

it prompts the user to type in a name for the datafile. *newFileName* performs three tasks:

1. uses its parameter string to prompt the user to type a datafile name on the keyboard
2. returns as a string value whatever name has been typed
3. creates an empty data file on disk represented by the datafile name and a Pascal desktop icon.

The second instruction, *rewrite(regretFile, dataName);*, opens a connection between the newly created *dataName* and the file organizing variable *regretFile.*

Remember, just as the printer required a file organizer to direct output to it instead of the Macintosh screen, so does the disk drive require a file organizer to direct input and output to it. In both cases, the two steps for using a file organizer are:

1. Declare a variable (the organizer) of a file type, such as *text.*
2. Open a connection between the organizer and file/device name.

The file/device name for the printer was *'PRINTER:'*. The file/device for a disk drive is the datafile string *dataName* that was returned by function *newFileName.*

You do not have to use function *newFileName* in order to create a datafile. You can assign a datafile name from within the program or you can write your own instructions to prompt the user to provide a name. An example of how to assign a datafile name from within the program can be found in Part 3 under *text.* Creating a file without *newFileName* requires a *rewrite* statement much like the one used in the last two chapters except, instead of *'PRINTER:'*, its second parameter uses *'diskName : datafileName'* (where the programmer must insert the disk's actual name and datafile's desired name). The name of your disk can be found in the Macintosh desktop beneath the disk icon. The original name of your disk was *Pascal,* but yours might have been changed.

You were introduced to the **for..to..do** loop in the last chapter. Here it is again. This time the *counter* variable will be used as the array's subscript, the bracketed number that identifies the individual elements of the array. That is the reason why the range of the **for** loop (*for i := 0 **to** 4*) is identical to the range of the array (***array** {0..4} of **string**{75}*).

The action performed within the **for** loop is a single instruction: *writeln(regretFile, regrets{i});*. This instruction will be executed five times. Each pass will write the string assigned to the array element *regrets{0}* through *regrets{4}* onto the disk file.

Remember, the first parameter, *regretFile,* is the file organizer that sends the contents of the second parameter to a disk instead of to the Macintosh screen. The *rewrite* command opened the connection between file organizer and the disk's datafile.

The final instruction of **procedure** toDisk, *close(regretFile),* closes the connection between the file organizer and the disk's datafile. It is always a good idea to *close* a file connection when you are finished using it. Though MacPascal will automatically close any open files when a program terminates, there might come a time when your programs use *two or more* file connections. Closing a file will protect it from being written on with data meant for another file.

16 Recalling Regrets from a Disk

What's next

This program, even shorter than the last chapter's, shows you how to retrieve information that has been stored in a disk datafile. The datafile that was created in the last chapter will be opened, and its contents displayed in the Text window.

Topics

oldFileName

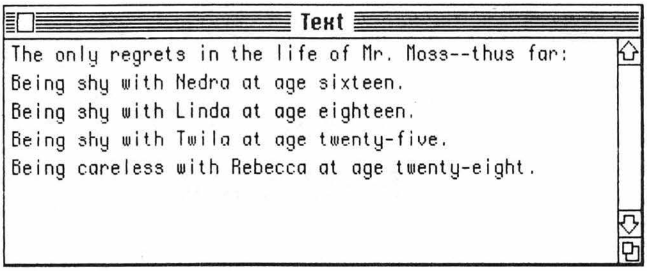
reset

not

eof

16.1 Program ReadRegrets

```
program ReadRegrets;
 var
  i : integer;
  dataName : string[75];
  regrets : array[0..4] of string[75];
  regretFile : text;
begin
 i := 0;
 dataName := oldFileName('select file name');
 reset(regretFile, dataName);
 while not eof(regretFile) do
  begin
   readln(regretFile, regrets[i]);
   writeln(regrets[i]);
   i := i + 1
  end;
 close(regretFile)
end.
```



```
The only regrets in the life of Mr. Moss--thus far:
Being shy with Nedra at age sixteen.
Being shy with Linda at age eighteen.
Being shy with Twila at age twenty-five.
Being careless with Rebecca at age twenty-eight.
```

16.2 Declarations

The variables declared in ReadRegrets are the same as those declared in last chapter's WriteRegrets. The only difference is in their placement. Since ReadRegrets does not use a procedure block, the variables are declared globally beneath the **program** heading.

Since ReadRegrets performs a single, simple task, the use of procedure blocks is not warranted.

Whenever your program, or a part of your program, requires the retrieval of information from a datafile, you should examine the type of the variables that *inserted* the data. Pascal has an affinity for *matching types* so you should be prepared to read from datafiles using that same type with which the datafile was written.

Hence, *regrets* is declared as an ***array{0..4} of string{75}*** and the file organizer declared as type *text* in both WriteRegrets and ReadRegrets.

16.3 Main

The first instruction assigns an initial value of 0 to the integer variable *i*. Until a variable is initialized, its value is undefined. The use of an undefined variable causes unpredictable results.

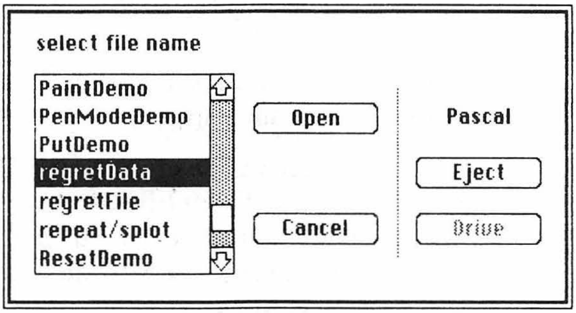
Mr. Moss's girlfriend causes unpredictable results, too. You might even say she is an undefined variable. Her spontaneity is enchanting. When she says the three most important values in life are awareness, intimacy, and spontaneity, you ought to take note because she knows more about important values than any computer ever built.

Mr.Moss has a list of the five most important values in life. They are: health, diversion, friends, intimacy, and children. Only one item in Mr. Moss's list matches with an item in his girlfriend's list.

But, hi ho, it's a good one.

The second instruction in **program** ReadRegrets is: *dataName := oldFileName(‘select file name’);*. *oldFileName* is a Toolbox function call similar to the function *newFileName* used in the last chapter. But instead of prompting for the user to input a new file name, *oldFileName* produces a dialog box containing all of the disk's file names. From this list, the user can select the appropriate file name.

Since the purpose of **program** ReadRegrets is to read the datafile created by **program** WriteRegrets, the correct file to select from the dialog box is same one the user typed in the dialog box produced by *newFileName.*



The string *‘regretData’* is returned by function *oldFileName* and assigned to the variable *dataName.* The parameter of function *oldFileName(‘select file name’)* works just like the parameter of *newFileName*—the literal is placed in the dialog box to help instruct the user what to do.

The next instruction, *reset(regretFile, dataName),* opens the connection between the file organizer *regretFile* and the user-selected datafile *dataName.*

reset is a companion Pascal procedure to *rewrite.* Whereas *rewrite* creates an empty datafile ready for writing, *reset* opens an existing datafile in order for it to be read. See Part 3 under *reset* and *rewrite* to find out more about file opening procedures.

There is a third MacPascal procedure that opens files; it is called *open.* The procedure *open* will either create a new datafile or open an existing one. However, the use of *open* is limited to *random-access files.* You can see examples of reading and writing to a random-access file in Part 3 under the definitions of *open* and *seek.*

Briefly, the files used in this chapter and the last are sequential-access files. When the file was opened by *rewrite* or *reset,* the first component of the file is number 0, and each sequential write or read operation affects component number 1, then component number 2, and so on.

Random-access files will not automatically increment to the next component with each write or read operation. To access any component other than component 0, a program must use the Pascal procedure *seek* to direct the file organizer to the desired component.

The definitions and examples in Part 3 offer more information on MacPascal's file capabilities.

The fourth instruction of **program** ReadRegrets is: ***while not*** *eof(regretFile)* ***do.*** The **while..do** loop will read each component of the datafile and write the string into the Text window. The loop will be performed *if* and *until* the boolean condition *eof(regretFile)* is **not** true.

The reserved word **not** reverses the boolean condition necessary for the loop to make its branching decision.

The boolean condition *eof(regretFile)* uses the Pascal function *eof* to determine whether the file organizer has reached the end of the datafile. The *eof* function returns a result of true when the file organizer has read (or written to) the last component of a datafile.

Since **program** WriteRegrets wrote five components into datafile, the **while not..do** loop will repeat five times before the function *eof(regretFile)* returns true and the loop is exited. In each repetition, the instruction *readln(regretFile, regrets{i})* will assign a component of the datafile to an element of the array *regrets*.

The assignment statement *i* = *i* = *1* increments the array subscript in brackets in each repetition of the loop. In this way the elements of the array are incremented at the same rate as the *readln* statement automatically increments along the datafile.

close(regretFile) closes the connection between the file organizer and the dataFile. Regrets are useful to the extent they give you courage to make new decisions and the caution not to make the same mistakes twice.

17 Trading Licks with MacWrite

What's next

This chapter's program retrieves the text of a MacWrite document. MacWriteRead will use the MacPascal Text window to display the contents of any document created by the MacWrite word processor. This program will also retrieve text from a document created by the Text Editor program found in the Tools folder of the Pascal disk.

■ Topics

oldFileName

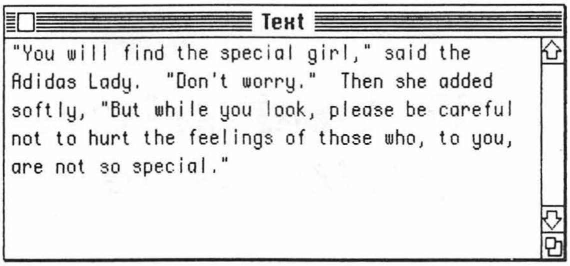
reset

eof

eoln

17.1 Program MacWriteRead;

```
program MacWriteRead;
 var
  giveName, docName : string[75];
  line : string;
  macWriteFile : text;
begin
 giveName := 'Select MacWrite file name';
 docName := oldFileName(giveName);
 reset(macWriteFile, docName);
 while not eof(macWriteFile) do
  begin
   readln(macWriteFile, line);
   writeln(line)
  end;
 close(macWriteFile)
end.
```



"You will find the special girl," said the Adidas Lady. "Don't worry." Then she added softly, "But while you look, please be careful not to hurt the feelings of those who, to you, are not so special."

17.2 Declarations

You won't find much new anywhere in this chapter, including the **var** declarations. Reading text from a MacWrite file is even simpler than reading text from your own disk files. Notice that the new name for the variable of type *text,* the file organizer, is *macWriteFile.*

MacPascal can read MacWrite documents only when those documents are saved using MacWrite's Text Only option. When you save a document from MacWrite, the Text Only option appears in the Save/Save As dialog box. As a result, the fancy print styles, fonts, and pasted-in graphics are not transferable to MacPascal.

In MacWrite, the datafile is called a document. Since Pascal files can contain any type of data—strings, integers, real numbers, arrays, records—the name *datafile* is more appropriate than *document.*

The MacWrite document uses only *string* type data. Whereas the last two chapters stored and retrieved data from the array *regrets,* **program** MacWriteRead requires only the declaration of *line,* a variable of type *string.*

The file-finding function *oldFileName* will again be used. Just for a change of pace, the literal used as the function's parameter will be replaced by the string variable *giveName.* The first instruction of MacWriteRead will assign a string value to *giveName.*

17.3 Main

The main body of MacWriteRead is remarkably similar to the main body of last chapter's RegretRead. The primary difference is in the composition of the datafile.

regrets was an array variable where each sequential component of the datafile was assigned to an element of the array. The datafile consisted of five components.

line is a string variable where each line of text in the datafile (document) is assigned to the variable *line.* The datafile consists of lines of text all in a single component.

The first two instructions of MacWriteRead prompt the user to select the file name of a MacWrite document. Function *oldFileName* produces a dialog box from which to choose a MacWrite file name. You can use the Eject oval of the dialog box if your MacWrite document is on another disk. The function will return the selected document name, and be assigned to the datafile *docName.*

The instruction *reset(macWriteFile, docName);* will open a connection between the file organizer *macWriteFile* and the datafile *docName.*

Remember, the Pascal command *reset* must be used when accessing an existing datafile. The command *rewrite* will create an empty datafile, effectively erasing the contents of any other file using the same file organizer name.

while not *eof(macWriteFile)* ***do*** creates the loop that exits upon finding a true value for the end-of-file function *eof.*

Just as the file organizer parameter directs *readln* and *writeln* statements to external devices (printer/disk), the same parameter directs the *eof* function to check the status of a datafile. The check for an end-of-file condition must be directed to a particular device's datafile, or else the function will assume the default device of the Macintosh screen.

A true value for the end-of-file condition is triggered by an invisible end-of-file marker. The last character of the last component of all datafiles is the end-of-file marker. Thus, when a *readln* or *writeln* statement encounters the marker at the end of a file, the *eof* function will return true. Otherwise, *eof* will return false.

program MacWriteRead also makes implicit use of the *eoln* (end-of-line) function. While not actually calling the *eoln* function, the instruction *readln(macWriteFile, line);* takes input from the datafile up until it reaches an *eoln* marker, assigning the input to its string parameter *line.* The *eoln* marker is also an invisible character. The *eoln* marker is set by the carriage return key and marks the *end-of-line* of files organized by type *text.*

After the *readln* instruction retrieves a line of text from the MacWrite document and assigns it to the string variable *line,* the next instruction *writeln* displays the string in the Text window.

The *readln* and *writeln* commands are repeated within the **while not..do** loop. Each repetition retrieves another line of text from the MacWrite document. When the *readln* statement encounters an end-of-file marker at the end of the datafile, the *eof* function is set to true, the loop exited, and the file connection closed.

You will find that MacWrite and MacPascal's Text Editor programs are able to read your program files and the text files your programs create. This should convince you that text files follow a standard protocol. Graphics present more of a challenge when trying to exchange data from one program to another, though as you will see in the next chapter, MacPascal makes some graphics exchanges very simple.

The MacWrite document used in this chapter came from Mr. Moss's private library. He refuses to divulge the identity of the Adidas Lady except to say she is over seventy, plays tennis, drives a sixteen-year-old Chrysler Newport, lives alone (she lost her family in the Holocaust), and modestly describes herself as:

A Cosmopolitan girl with a Racing Form face.

18 Trading Licks with MacPaint

What's next

This chapter's program fills the Drawing window with a familiar graphics display, then saves the window's contents as a MacPaint document. As a MacPaint file, you can use any of the tools of MacPaint to "touch up" your Pascal displays.

■ Topics

const {string}

procedure {nested}

eraseOval

paintOval

frameOval

abs

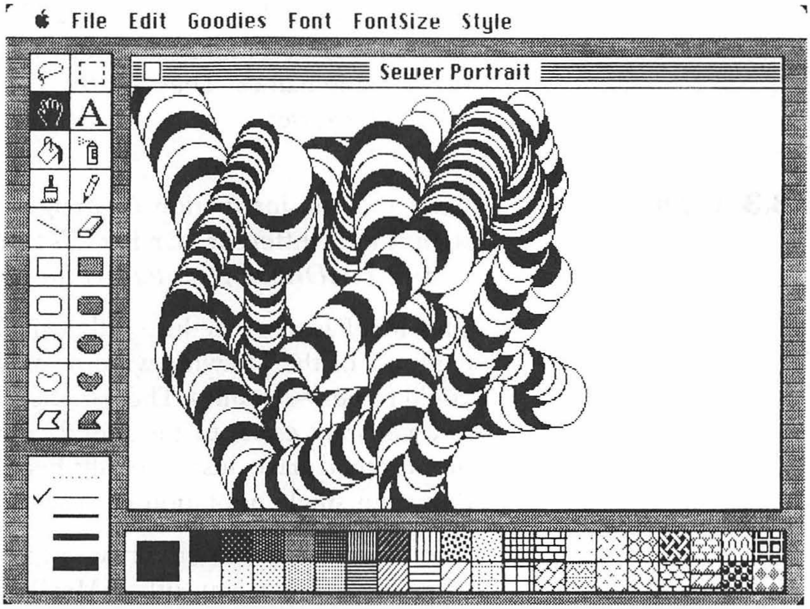
random

mod

saveDrawing

18.1 Program SaveTheSewer;

```
program SaveTheSewer;
  const
    macPaintTitle = 'Sewer Portrait';
  var
    top, left, bend : integer;
  procedure createSewer;
    var
      topHop, leftHop, line, node, girth : integer;
    procedure drawSewer;
      const
        min = 20;
        max = 250;
    begin
      EraseOval(top, left, top + girth, left + girth);
      if node mod 3 = 1 then
        PaintOval(top, left, top + girth, left + girth)
      else
        FrameOval(top, left, top + girth, left + girth);
      if top < min then
        topHop := abs(topHop)
      else if top > max then
        topHop := -abs(topHop);
      if left < min then
        leftHop := abs(leftHop)
      else if left > max then
        leftHop := -abs(leftHop);
      top := top + topHop;
      left := left + leftHop
    end;   {end drawSewer}
  begin
    line := random mod 30;
    girth := random mod 25 + 24;
    topHop := random mod 19 - 9;
    leftHop := random mod 19 - 9;
    for node := 1 to line do
      drawSewer;
  end;   {end createSewer}
begin
  top := 0;
  left := 0;
  for bend := 1 to 25 do
    createSewer;
  saveDrawing(macPaintTitle)
  end.   {end ComputerSewer}
```



18.2 Definitions and declarations

There are two **const** sections in SaveTheSewer. The global constant *macPaintTitle* is defined as the string literal 'Sewer Portrait'. The constant name can now substitute for the literal anywhere in the program.

The other **const** section is local to **procedure** drawSewer. You might have already noticed that **procedure** drawSewer begins before **procedure** createSewer has ended (or really even begun). This technique of nesting procedures will be explained later in this chapter.

The constants *min* and *max* have been defined with drawSewer to have values equal to 20 and 250 respectively. There are two reasons for defining these constants:

1. The constant names *min* and *max* help clarify that their integer values will be used to specify a minimum and maximum range.
2. Should you at a later time decide you want to alter the minimum or maximum values, it will be simpler to change the **const** definition than to search for and change each occurrence of the integers in the program code.

All of the variables declared in SaveTheSewer are of type *integer.* You should be familiar not only with the *integer* type, but also the variable names. They are the same variables used in Part 1's ComputerSewer.

18.3 Main

Only the last instruction of **program** SaveTheSewer makes it significantly different than Part 1's ComputerSewer. That instruction is: *saveDrawing(macPaintTitle).*

This MacPascal procedure call is all that is necessary to save the contents of the Drawing window as a MacPaint document (or if you'd rather—datafile). The parameter *macPaintTitle* was defined as a constant equal to the literal 'Sewer Portrait'; however, the literal could have just as easily been inserted as the parameter (between single quotation marks).

The MacPaint document created by *saveDrawing* is represented with an icon like any other MacPascal datafile. However, this icon differs from other MacPaint icons, and the document cannot be opened directly through the icon. Instead, you must open MacPaint, Close the current window, then choose Open from the File menu. The resulting dialog box will allow you to choose 'Sewer Portrait' from among its document options.

This chapter could end right here. You have seen how to send a MacPascal display to MacPaint, and unfortunately, the process of getting a MacPascal program to retrieve a Macpaint document is too tricky to tackle here. But SaveTheSewer has been written a bit differently than ComputerSewer, and these differences deserve attention.

The first three instructions are the same in the main body of both programs. Of course, in ComputerSewer all the code is contained in the main body, whereas SaveTheSewer has been broken down into procedures. As a result, these three instructions occupy the top of ComputerSewer and the bottom of SaveTheSewer.

18.4 Procedure createSewer;

The fourth instruction of SaveTheSewer is the procedure call *createSewer;*. Below the heading ***procedure** createSewer;* are the procedure's **var** declarations, but below the declarations is the heading ***procedure** drawSewer;*. You might be wondering: Where does the body of **procedure** createSewer **begin** and **end**?

The easiest way to figure this out is to examine the Program window's indentations. The **begin** and **end** of all blocks, even the main body, are indented the same as the block's heading.

The **begin** immediately following ***procedure** drawSewer;* is indented the same as the heading of drawSewer. Consequently, the block of statements following this **begin** belong to drawSewer. The **end** of drawSewer follows the rules of all **begin..end** brackets. An **end** is matched to the immediately preceding unended **begin**.

To make the pairing even clearer, a comment between curly brackets {end drawSewer} has been added. You should use comments within your programs wherever there might be a point of confusion: either for yourself at a later time or for someone else trying to understand your programming.

The first instruction of **procedure** createSewer is: *line := random **mod** 30;*.

Run SaveTheSewer using Step if you have any doubts about the sequence of program flow. The pointing hand of MacPascal will show you in what order a program executes its instructions.

Random and ***mod*** were explained in Part 1 and can also be found in Part 3, so they won't be covered here. But in a sentence, the integer variable *line* will be assigned a random value of between 0 and 29.

The instruction *girth := random **mod** 25 + 24;* assigns *girth* a random integer value of between 24 and 48. The addition of 24 is performed *after* the *random **mod** 25* value is ascertained. (This makes a lot of sense, because otherwise you might just as well have written *random **mod** 49,* which produces a value between 0 and 48.)

The next two instructions are also *random **mod*** assignments, followed by the loop ***for** node= 1 **to** line **do**.*

Notice that the range of the **for** loop allows for integer variables as well as integers.

The action performed by the **for** loop is the procedure call *drawSewer.* Program flow continues at the first instruction of **procedure** drawSewer.

18.5 Procedure drawSewer;

The instructions of *drawSewer* are the same as those performed in ComputerSewer. The interesting characteristic of the procedure is that it is *nested* inside of **procedure** createSewer. *drawSewer* would work as well had it been placed before *createSewer,* just as ComputerSewer worked without any procedures.

The primary advantage to nesting is the enhancement of Pascal's block design. The purpose of blocks is to make programming easier and more efficient by grouping instructions by the task they perform. Likewise, a design is enhanced when you group together blocks by the task they perform. Part of creating a sewer is drawing the sewer, so it makes sense to have the *draw* procedure be part of the *create* procedure.

A caution to nesting arises because a nested procedure is *local* to its host procedure, and therefore cannot be used from outside the host block. The concept of *local* versus *global* applies to blocks in the same way as it does to definitions and declarations. Since *drawSewer* is only called from *createSewer,* its locality presents no problems.

The Quickdraw commands *eraseOval, paintOval,* and *frameOval* are explained in detail in Part 3. The building of the sewer is accomplished by looping through the erase, paint, and frame commands, and changing the location parameters on each pass.

The *abs* (absolute value) function used in the assignment statements of drawSewer force turnaround values to the *topHop* and *leftHop,* the sewer's growth variables. This assures that values outside the *min* and *max* window range will redirect to within the range.

Here it is at the close of another chapter and Mr. Moss has yet to put in his two cents' worth. A whole chapter without any sidetrack might disappoint those who don't give a damn about Pascal, and are reading this book for spiritual uplifting and interpersonal growth. Proselytizing philosophy is fodder for the dim of mind, but at least here you pick up a computer language.

The few words Mr. Moss does find pertinent fall somewhere near the asphalt's middle white line. On one shoulder is enough fever, fear, and decay to make a person want to puke in all directions at once. On the other are the giving, brown eyes of Rollo, the salt-sweet tastes from the neck of Mr. Moss's girlfriend, fathomless colors in daytime and in dreams, "Late Night with David Letterman," grilled cheese and avocado sandwiches, and the invention of the electric blanket.

No, Mr. Moss has nothing to say in this chapter. Maybe your soul needs saving, but here all you'll find out is how to SaveTheSewer. When Mr. Moss needs *truth* he walks down a crowded sidewalk and overhears bits of conversation. Once, he heard—or thought he heard—a young man with a coal miner's accent say:

I not only believe in God,
I believe I can get revenge.

19 QuickSounds for the Hard of Herring

What's next

The programs in the next four chapters create sounds using the *note* procedure. This chapter explains the three parameters of the procedure, allowing you to experiment with different sounds in a very simple Pascal program.

■ Topics

sysBeep

note

19.1 Program NoteTaker;

```
program NoteTaker;
  const
    amp = 150;
    time = 30;
  var
    freq : integer;
begin
  note(1000, 200, 45);
  note(29830, 111, 83);
  note(-19005, amp, 140);
  sysBeep(60);
  note(7, 255, 99);
  for freq := 1 to 8 do
    note(freq * 1000, 85, time)
end.
```

19.2 Definitions and declarations

The constant names *amp* and *time* are equated with the values *150* and *30,* respectively. The variable name *freq* is declared to be an *integer* type.

amp, time, and *freq* will be used to provide values to the Toolbox procedure *note. amp* is short for amplitude. *freq* is short for frequency. *time* will represent the duration of a sound.

19.3 Main

procedure note requires three integer parameters. The order of the integers in the parameter list distinguish their purpose. The format for the procedure is: *note(frequency, amplitude, duration);.*

The value for *frequency* can be any integer value. The boundaries for an *integer*-type value are: [−32676..32676]. However, *frequency* cannot have a value of zero [0] because MacPascal uses the integer in a division and a divide by zero error will result.

The values for *amplitude* and *duration* must be integers in the range: [0..255].

Execution of the note procedure causes a single square-wave tone to be produced. Frequency is measured in hertz, a wave measurement equal to one cycle per second. Amplitude defines a range of sound intensity. Duration indicates the number of sixtieths of a second that the tone will last.

Macintosh sound capabilities are much more extensive than those provided by the *note* procedure. As you might imagine, the more complex sound, speech, and music syntheses require more complex programming techniques. The *MacPascal Reference Manual* offers more information on multichannel sound generation.

The main body of NoteTaker shows the *note* procedure using a wide range of parameter values. Sometimes values are inserted directly in the parameter list. Other times, a variable or constant is substituted for an integer.

Also included is the Toolbox procedure *sysBeep.* Its single parameter is a duration integer that increments by .022 seconds. The statement *sysBeep(60)* produces a square-wave tone lasting about 1.3 seconds. You should recognize this square-wave tone as the same you hear when the Macintosh is turned on.

The last two statements constitute a **for** loop, a quick means of hearing a series of different frequencies with a minimum of code. The frequency parameter is incremented by 1000 with each pass of the loop. The star symbol (*) signifies multiplication in Pascal.

Regarding this chapter's title: Herring is one of nature's more intriguing foods. This sea creature is most commonly found in bottles on supermarket shelves. Served on top of crackers with sour cream and onions, it is a snack you will likely remember throughout one day and part of the next. Mr. Moss strongly recommends you add your own sour cream and onions, even if you have the opportunity to purchase it already prepared.

Similar advice applies to another magical food. More important than Pascal, schmascal, says Mr. Moss's mom:

> You never know how long the chopped liver has been sitting in the deli case. Better you should make it yourself and know it's fresh.

20 Pick-Me-Ups at the Frequency Bar

What's next

This program lets the user test a wide range of Macintosh's sound frequencies simply by pressing the mouse button. Graphics and sound are combined, as the frequency bar will display in hertz the tone being produced. The drawing window is sized from within the program, and an *end* oval is drawn for a friendly program exit.

■ Topics

fillRect

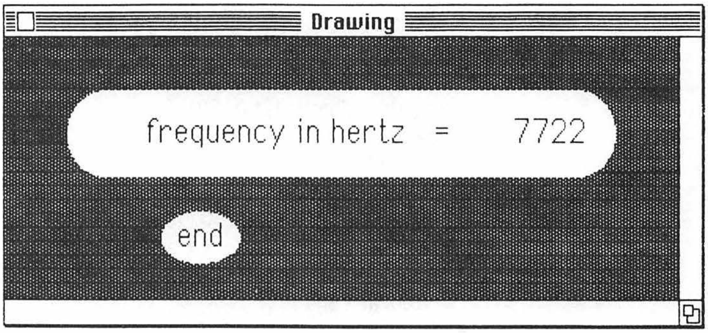
globalToLocal

eraseRoundRect

eraseOval

20.1 Program FrequencyBar;

```
program FrequencyBar;
 var
  drWindow, r1, r2 : rect;
  pt : point;
 procedure makeBar;
 begin
  setRect(r1, 40, 30, 390, 80);
  eraseRoundRect(r1, 50, 50);
  moveTo(90, 60);
  textSize(18);
  writeDraw('frequency in hertz  =');
  setRect(r2, 100, 100, 150, 130);
  eraseOval(r2);
  moveTo(110, 120);
  writeDraw('end')
 end;
 procedure soundNote;
  const
   amp = 200;
   time = 20;
   range = 26;
  var
   x, y, freq : integer;
 begin
  getMouse(x, y);
  setPt(pt, x, y);
  if ptInRect(pt, r1) then
   begin
    freq := (x - 39) * range;
    eraseRect(40, 300, 60, 380);
    moveTo(300, 60);
    writeDraw(freq);
    while button do
     note(freq, amp, time);
   end
 end;
begin
 hideAll;
 setRect(drWindow, 30, 100, 475, 265);
 setDrawingRect(drWindow);
 showDrawing;
 globalToLocal(drWindow);
 fillRect(drWindow, dkGray);
 makeBar;
 repeat
  soundNote
 until ptInRect(pt, r2) and button
end.
```



20.2 Definitions and declarations

You should be familiar with all the **const** and **var** listings. The constants *amp* and *time* are used the same way as in the last chapter. The constant *range* will help correlate the mouse position to the range of frequencies.

The variable *drWindow* of type *rect* will establish the dimensions of the drawing window. The other two variables of type *rect, r1* and *r2,* will determine the rounded rectangular area of the frequency bar and the oval area of the *end* button.

The variable *pt* of type *point* is assigned with **procedure** setPt using the mouse's current coordinates.

20.3 Main

The first four instructions format the Macintosh screen with the single enlarged Drawing window. You last saw these window drawing commands used in Chapters 11 and 12.

The fifth and sixth instructions are new. First, look at the sixth instruction: *fillRect(drWindow, dkGray);.*

fillRect is similar to the other shape-drawing procedures—frameRect, paintRect, and eraseRect. The dimensions provided by the type *rect* parameter *drWindow* determine the area to be filled with the pattern indicated by the second parameter, in this case, *dkGray.*

However, the dimensions provided by the variable *drWindow* might not plot on the Macintosh screen exactly as you imagined. The coordinates are identical to those set four instructions earlier by *setRect(drWindow, 30, 100, 475, 265),* yet the coordinate maps have changed.

The *setRect* command plots points based on a coordinate map where point (0,0) is the upper-left corner of the Macintosh screen. The *fillRect* command plots points based on a coordinate map where point (0,0) is the upper-left corner of the Drawing window.

The intent of the *fillRect* command is to fill the entire Drawing window with the dark gray pattern. This could be accomplished by the command *fillRect(0, 0, 165, 445).* This provides the same sized rectangle as the *setRect* command with a top-left corner point that is the top-left corner of the Drawing window.

Alternatively, **program** FrequencyBar uses the Toolbox procedure *globalToLocal.* This procedure converts a point expressed in global coordinates (the Macintosh screen) to one expressed in local coordinates (the Drawing window). The corner points of the *rect* variable *drWindow* are converted onto the coordinate map of the Drawing window.

As usual, difficult concepts are best illustrated by experimentation. Try running FrequencyBar after you have removed the command *globalToLocal(drWindow).*

The remainder of the main body performs procedure calls to the program code. The *soundNote* procedure is encased in a **repeat..until** loop whose exit condition is: *ptInRect(pt, r2)* ***and*** *button.* As you might expect, the parameter variable *r2* is the *end* oval, so that if the mouse is pointing within this oval **and** the button is being pressed, the loop is exited and the program ends.

20.4 Procedure makeBar;

Let it be known right from the start that Mr. Moss rarely frequents bars. Bars are for getting sloshed among sloshed people, a recreation Mr. Moss chooses to forego. As for meeting people, the singles scene at drinking establishments gives Mr. Moss the willies.

The ineptitude of single people trying to meet a mate is nothing less than an American cultural disgrace. Mastering Pascal is a cinch compared to the task of finding a good life partner. Dear Abby, whose credentials far outweigh Mr. Moss's, says to put your friends on the lookout and involve yourself in community activities. All that Mr. Moss can add is:

Be brave,
be considerate,
ask for a phone number,
and floss your teeth every day.

The commands in **procedure** makeBar are straightforward: Create a type *rect* variable, draw it, position the Quickdraw pen within the rectangle, set the text size, and write out the string. The same set of procedures works for the *end* oval as for the rectangle called *frequency bar,* except that *textSize(18)* does not need to be repeated.

The two shape-drawing commands are *eraseRoundRect* and *eraseOval.* Since the entire Drawing window was filled with a dark gray pattern, the erase commands draw their shapes in the white pattern. The same effect could have been achieved by using the *fillRoundRect* and *fillOval* procedures, specifying as the second parameter of each, the pattern constant *white.*

The rounded-corner rectangular shapes require three parameters. The latter two integers specify an oval shape within the rectangle that determines the degree of roundness.

The *writeDraw* command allows the same versatility as the *writeln* command, including numeric parameters and colon/format modifiers. The Quickdraw procedure *drawString* will also draw text into the Drawing window, though its parameter must be type *string.* Part 3 has more information on this procedure.

20.5 Procedure soundNote;

This procedure reads the position of the mouse, and if the mouse button is pressed while the mouse is pointing inside the frequency bar rectangle, a note is played. The horizontal coordinate of the mouse (x-axis) is used in an equation to determine the frequency (first parameter) of the note procedure. Also, the value of the first parameter is displayed on screen within the bar.

The conversion of the mouse's x coordinate to the variable *freq* is performed by the assignment statement *freq := (x−39) * range;.* The logic behind this equation is that, within the rectangular area of *r1,* the range of possible x coordinate values is: [40..390]. Look at the first *setRect* command in **procedure** makeBar to verify this.

By subtracting 39 from x, the range becomes: [1..351]. The constant *range* equated with the integer 26 is an arbitrary multiplicand which magnifies the range of *freq* values to: [26..7236]. This gives the frequency bar a much broader spectrum of notes. A horizontal movement of a single dot will increase or decrease the frequency response by 26 hertz.

The instruction *eraseRect(40, 300, 60, 380);* is necessary to erase the prior frequency value before a new value is drawn onto the screen. The *writeDraw* command inserts the integer value of *freq* at the pen location specified by *moveTo(300,60),* but makes no provision to erase the current screen dots. The *eraseRect* instruction makes sure the new numbers are not drawn on top of old numbers.

You might need to experiment to find the rectangle occupied by text in the Drawing window. Text is drawn in the Drawing window to the right of the pen location, with the left end of the text's base line at the pen's location.

21 Passing Parameters on E Street

What's next

This chapter's program executes the *note* command seven times, each time sounding a note of a different frequency. The method by which the frequency is assigned introduces a new and powerful Pascal concept.

Topics

procedure {with parameters}

note

21.1 Program EStreet;

```
program EStreet;
 procedure hitIt (eStreet : integer);
 begin
   note(eStreet, 225, 70)   {eStreet freq range: (-32767..32767)}
 end;                       {but freq cannot equal zero}
begin
 hitIt(400);
 hitIt(8000);
 hitIt(-19000);
 hitIt(32000);
 hitIt(555);
 hitIt(17);
 hitIt(9100)
end.
```

21.2 Declarations

Look at **program** EStreet. Do you see any declarations? There appears to be a declaration on the same line as the title to **procedure** hitIt: ***procedure** hitIt (eStreet : integer);*.

The name *eStreet* is not declared anywhere else in the program, and it does have the colon/type (: integer) format that is characteristic of **var** declarations. You might come to the conclusion that this is an alternative way of declaring a local variable.

This assumption would be correct, but variables declared by this method serve a distinct purpose. They are a communication device between the procedure call and the procedure instructions.

21.3 Main

The main body of EStreet consists of seven calls to **procedure** hitIt. In each case, a number in parentheses accompanies the procedure name. This number is the procedure call's parameter.

You have used parameters many times while calling Quickdraw, Toolbox, and Pascal procedures, but here, for the first time, you are using parameters in the procedures listing in the program code.

The parameter stated in parentheses is passed along to the procedure while being assigned to the variable name *eStreet.* Since *eStreet* has been declared to be of type *integer,* you can rightly assume that the parameter in the procedure call must be an integer value.

21.4 Procedure hitIt (eStreet : integer);

There is only one instruction in **procedure** hitIt, the call to the Toolbox procedure *note.* The amplitude and duration parameters of *note* are given the integer values of 225 and 70. The frequency parameter will be assigned by values passed from the main body to the parameter name *eStreet.*

The first call to **procedure** hitIt sends a value of 400 to the *eStreet* parameter. As a result, a note is sounded that has a frequency value of 400, a amplitude of 225, and a duration of 70.

The second call to **procedure** hitIt passes along a value of 8000 to *eStreet.* The third call passes a value of − 19000, and so on through the seven calls to *hitIt.*

Once the parameter name *eStreet* has been sent a value from the procedure call's parameter list, the name *eStreet* works just like a local variable declared with **var**. Better yet, the value of *eStreet* has already been assigned.

By passing parameters, Pascal's block structure is greatly enhanced. Not only can you group together instructions that perform a certain task, you can also send data to those instructions. Procedures can retain the advantages of local declarations (clarity, modularity) and still receive communications from other blocks of the program.

In case you have any friends who think the BASIC computer language is hot stuff, ask them if they can name blocks of statements *and* pass data to those blocks.

Basic programmers tend to be an ornery lot because of all the times they are told where they ought to GOTO.

22 A Springsteen Type Concert

What's next

This program closely resembles the last chapter's. The procedure call *hitIt* will pass along a frequency parameter, and seven notes will be played. But instead of passing an integer, the parameter will be a new type defined within the program. The use of a **type** declaration can make a program easier to understand.

■ Topics

type

ord

22.1 Program BornToRun

```
program BornToRun;
  type
    band = (theBigMan, theBoss, max, danny, garry, theProfessor, nils);
  procedure hitIt (eStreet : band);
    var
      amp, time, freq : integer;
  begin
    amp := random mod 200 + 50;
    time := random mod 80 + 20;
    freq := (ord(eStreet) + 1) * 600;
    note(freq, amp, time)
  end;
begin
  hitIt(max);
  hitIt(theBigMan);
  hitIt(danny);
  hitIt(nils);
  hitIt(theProfessor);
  hitIt(garry);
  hitIt(theBoss)
end.
```

22.2 Definitions and declarations

A new definition heading called **type** is listed directly below the **program** heading. The **type** heading follows many of the same rules as the **const** and **var** headings. **type** definitions can be either global or local, and each listing must be ended with a semicolon.

Pascal requires that definitions and declarations of the same block be placed in the following order:

const

type

var

procedure or **function**

Information on two additional definitions, **uses** and **label,** can be found in Part 3.

A **type** definition can take a few different formats. The one you see in BornToRun is an *ordered list.* The name *band* is equated with the entire list of names. Punctuation requires that the list be stated between parentheses, with a comma separating each item on the list. The equals sign (=) equates the name *band* with the ordered list.

The reason that the list of names is called an *ordered* list is because the position of each name has a numeric significance. *theBigMan* is the first name on the list, and later on in the program, the number 0 will be associated with *theBigMan.* (Pascal likes to begin counting with the number zero.) Likewise, *nils* is the seventh name on the list, and the number 6 will be associated with *nils.*

Nils Lofgren, guitarist for Bruce Springsteen's E Street Band, put out some fine music of his own with a band called *Grin.* Grin never sold many records, but their fine music helped bring Nils and Bruce together.

Partnerships such as Lennon/McCartney, Jagger/Richards, and Jobs/Wozniak have done well: add to the list Andy Hertzfeld, Burrell Smith, and Bill Atkinson, the core behind the Macintosh computer. The Macintosh team—not much larger than the E

Street Band—showed an artistic side of technology that the suit-and-tie clones of corporate America could not even envision.

22.3 Main

The main body of BornToRun consists of the same seven procedure calls to *hitIt* you saw in last chapter's *EStreet*. Only the parameter list of each procedure has been changed. Instead of an *integer* parameter, you are sending to *hitIt* a *band*-type parameter.

Now *band* might not seem to be as logical a generic type as *integer*, but that is exactly what you accomplished when you defined *band* beneath the definition heading **type**. Whereas an *integer* type consists of the set of whole numbers between [− 32676..32676], the *band* type consists of the set of names [*theBigMan..nils*].

The first instruction of the main body, *hitIt(max),* passes the name *max* to the parameter *eStreet. eStreet* has been declared to be of type *band* in the **procedure** heading, and is being passed the value of *max.*

22.4 Procedure hitIt(eStreet : band);

The first two instructions of *hitIt* set values for the variables *amp* and *time.* Rather than giving the amplitude and duration of the *note* procedure constant values, **procedure** *hitIt* uses Pascal's predefined function/operator *random* ***mod*** to assign a range of random values to these variables.

amp will be assigned a value in the range [50..249]. *time* will be assigned a value in the range [20..99].

The next instruction of **procedure** hitIt assigns a value to the variable *freq: freq := ord(eStreet) + 1) * 600;.*

The Pascal function *ord* returns the *ordinal number* of its parameter. (Pascal uses a standard assignment of ordinal numbers called the ASCII character set, about which you can find more information under *ord* in Part 3.) Ordinal numbers are always of type *integer.*

The ordinal number of an item in a program-defined **type** such as *band* is determined by the position of the item in the ordered list. The first item has an ordinal value of 0, the second item has the ordinal value of 1, and so on through the seventh item, whose ordinal value is 6.

When **procedure** hitIt is first executed, the value that has been passed to *eStreet* is *max.* The Pascal function *ord(max)* returns the integer 2. *max* occupies the third position in the ordered list of type *band,* thus its ordinal number is 2. (Remember, in lists and files, Pascal counts beginning with the number 0.)

Now that *ord(eStreet)* has returned with a value of 2, an addition and multiplication is performed. (2 + 1) * 600 produces a frequency value of 1800.

The Toolbox *note* procedure is then executed with the assigned values of *freq, amp,* and *time.* The next time *note* is executed, *amp* and *time* will again be assigned random values, and *freq* will be assigned the ordinal value of *theBigMan (0)* plus 1 times 600. The second note will have a frequency value of 600 since (0 + 1) * 600 = 600.

23 Detecting Shady Characters

What's next

The program InputTesting is composed largely of a single function called *checkChar.* All input entered from the keyboard is filtered through this function so that only designated characters are accepted and written onto the screen. Weeding out bad input before it enters a program is one of the best ways of making your program bug-free.

■ Topics

type

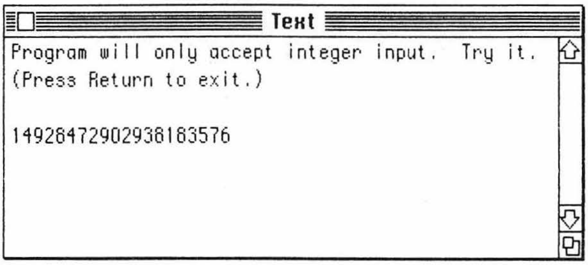
set of

function {with parameters}

get

23.1 Program InputTesting

```
program InputTesting;
 type
  keys = set of char;
 var
  fineNumber : keys;
  ch1 : char;
 function checkChar (fineKey : keys) : char;
  var
   fine : boolean;
   ch : char;
  begin
   repeat
    ch := input^;
    get(input);
    fine := ch in fineKey;
    if not fine then
     sysBeep(5)
    else
     write(ch)
   until fine;
   checkChar := ch
  end;
 begin
  fineNumber := ['0'..'9'];
  writeln('Program will only accept integer input.  Try it.');
  writeln('(Press Return to exit.)');
  writeln;
  repeat
   ch1 := checkChar(fineNumber)
  until eoln
 end.
```



```
Program will only accept integer input.  Try it.
(Press Return to exit.)

14928472902938183576
```

23.2 Definitions and declarations

A new kind of **type** definition is used in InputTesting. The name *keys* is equated with *set of char*. You have used predefined types such as *integer* and *char*, and you have defined your own **type** with an ordered list of *band* members. Now you are defining your own **type** as a group of, as yet, unspecified characters.

The difference between type *char* and **type** *set of char* is that a variable of type *char* can only represent a single character at a time, whereas a variable of type *set of char* can represent a group of characters at the same time.

The letters ['a','b','c','x','z'] are a set of characters. The uppercase letters from ['A'..'Z'] are also a set of characters. The following are all examples of legitimate sets of characters:

[‘a’..‘z’] {all the lowercase letters from ‘a’ to ‘z’}
[‘2’..‘6’,‘8’] {the numbers 2,3,4,5,6 and 8}
[‘$’,‘ + ’,‘*’,chr(32)] {the symbol characters plus the return key}

Any single character that can be represented as type *char* can be included in a variable of type *set of char.* The characters assigned to a variable of type *set of char* must be enclosed by brackets [].

Individual characters can be notated in one of three ways:

1. directly, in which case they must be placed between single quotation marks[‘’]
2. within an ordered list that abbreviates using two-dot notation [..]
3. through ASCII representation: [*chr(ordinal) number)*]

The **var** declarations of **program** InputTesting include *fineNumber,* a variable of the newly defined type *keys.* The actual assignment of a set of characters to the variable *fineNumber* occurs in the main body.

ch1 is declared a global variable of type *char.* This variable will be assigned the result of **function** checkChar.

Two variables are declared locally with *checkChar. ch* is a variable of type *char* which holds the current character typed in from the keyboard. *fine* is a boolean variable which is assigned a true value if the character entered from the keyboard is a member of the designated set of allowable keys.

23.3 Main

The first instruction assigns a value to *fineNumber,* the variable of type *set of char.* The right side of the assignment statement shows that *fineNumber* has a value of the numeric characters ‘0’ through ‘9’.

program InputTesting will only allow these keyboard characters to appear on the Macintosh screen. If any other character is pressed, a beeping sound will result without the character ever appearing on screen.

The **repeat..until** *eoln* loop at the bottom of the main body contains the **function** call to *checkChar*. The function will return a single character to be assigned to the variable *ch1*. If the *eoln* condition is true, that is, the line return character (*return* key, *chr(32)*) has been pressed, the loop is exited and the program ends.

You should take note that the call to **function** checkChar contains a parameter. The value of *fineNumber* will be passed along to **function** checkChar in the same way that parameters were passed to **procedures** in the last two chapters.

Though the advantages of passing parameters are less apparent in short programs, the result is **functions** and **procedures** that are more self-sufficient. Self-sufficient program blocks enhance the clarity of longer programs accomplishing many tasks.

On clarity, Mr. Moss reflects:

> The beauty of clear writing derives in large part from contrast to the pea soup fog within which our brains operate.

23.5 Function checkChar (fineKey : keys) : char;

checkChar uses two variables and the *fineKey* parameter to check for valid keyboard input. But the first task is to wait for input from the keyboard.

Early in Part 2, you were introduced to Pascal's *readln* command, which accepted input from the keyboard. However, one of the characteristics of *readln* is that as soon as an entry from the keyboard is recognized, the character is displayed on screen. *checkChar* uses another method of getting input from the keyboard, one that does *not* automatically display the keyboard entry.

This alternative method of keyboard entry is shown in the first two instructions of *checkChar:*

```
ch := input^;
get(input);
```

The assignment statement *ch := input^;* assigns a character from the keyboard *buffer* to the variable *ch.* The keyboard buffer is a holding place where each keypress is kept while waiting for further instruction. Pascal accesses the keyboard buffer through a predefined file called *input.* The file name *input,* followed by the caret sign(^), represents the file's buffer.

The second instruction, *get(input);,* advances the file buffer so that the next character input can be read from the keyboard. More information on the uses of Pascal's *get* procedure can be found in Part 3.

The remaining instructions of **function** checkChar explain themselves. The boolean variable *fine* is assigned a true value if the character *ch* is **in** the set of acceptable characters represented by the parameter *fineKey.*

fineKey and *fineNumber* are assigned the same values, the set of characters ['0'..'9']. The difference between the two is that *fineKey* is the parameter name given in the heading of **function** checkChar, while *fineNumber* is the global variable assigned in the main body whose value is then passed on to *fineKey.* Though the use of parameters is not required by this example, the technique is useful in many other applications.

A false value of *fine* will result in *sysBeep(5)* sounding its disapproval. The **else** condition, a true value of *fine,* will result in the character being written onto the Macintosh screen. Only when an acceptable character is entered will *checkChar* be assigned the value of *ch,* and the **function** exited.

24 Stringy Strings and Split Ends

What's next

The program StringTesting expands upon the keypress filter used in the last chapter. The designated characters that pass through function checkChar are used to create *character-tested* strings. Remember, input-checking techniques are an effective way of making a program run without error.

■ Topics

var {parameters}

chr

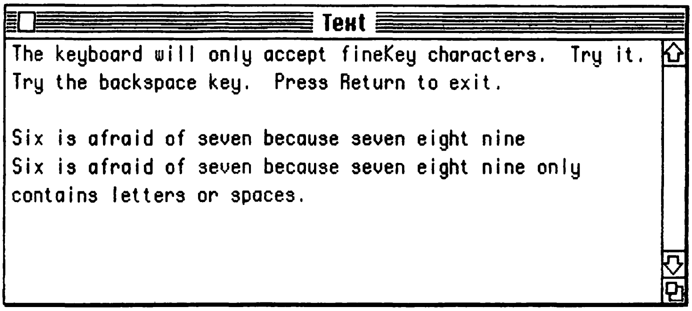
length

delete

concat

24.1 Program StringTesting;

```
program StringTesting;
 type
  keys = set of char;
 var
  fineKey : keys;
  sp : char;
  s : string;
 function checkChar (fineKey : keys) : char;
  var
   fine : boolean;
   ch : char;
  begin
    repeat
      ch := input^;
      get(input);
      fine := ch in fineKey;
      if not fine then
        sysBeep(5)
      else
        write(ch)
    until fine;
    checkChar := ch
  end;
  procedure fineString (var s : string;
                  fineKey : keys);
    var
      newChar, bs : char;
      okayKey : keys;
  begin
    bs := chr(8);     {ascii code for backspace key}
    repeat
      if length(s) <> 0 then
        okayKey := fineKey + [bs]
      else
        okayKey := fineKey;
      newChar := checkChar(okayKey);
      if (newChar = bs) then
        delete(s, length(s), 1)
      else
        s := concat(s, newChar)
    until eoln
  end;
begin
  writeln('The keyboard will only accept fineKey characters.  Try it.');
  writeln('Try the backspace key.  Press Return to exit.');
  writeln;
  sp := chr(32);     {ascii code for space bar}
  fineKey := ['a'..'z', 'A'..'Z', sp];
  fineString(s, fineKey);
  writeln;
  writeln(s, ' only contains letters or spaces.');
end.
```



```
The keyboard will only accept fineKey characters.  Try it.
Try the backspace key.  Press Return to exit.

Six is afraid of seven because seven eight nine
Six is afraid of seven because seven eight nine only
contains letters or spaces.
```

24.2 Definitions and declarations

You have seen all of the types used in StringTesting in previous chapters. The program-defined **type** *keys* is again equated with *set of char.* The variables *fineKey* and *okayKey* of type *keys* will be used to designate acceptable keypresses just as *fineKey* and *fineNumber* did in the last chapter.

The variables *fineKey, sp,* and *s* are declared globally because they are used in the program's main body. The only other occasion for using global variables is when the variable will be used in many of the program's **procedures** and **functions.** Use of a global variable will avoid the need for repeating the same local declaration in each block.

24.3 Main

The fourth instruction of the main body is: *sp := chr(32);.*

The Pascal function *chr* returns the character whose ordinal number is given as the function's parameter. The table of ASCII values in Appendix C, Part 3, shows that the character associated with the number 32 is the *space bar* key. Since the *space bar* key does not have a literal character of its own, it must be referenced by its ASCII code number. The assignment statement shown above allows the program to substitute *sp* instead of the unwieldy *chr(32)* whenever the *space bar* character is mentioned.

The next instruction assigns the variable *fineKey* to the set of characters made up of all letters, both lower and upper case, as well as the space bar character, *sp.* Remember from last chapter that all characters not included in *fineKey (fine Number)* got beeped and never appeared on the screen.

Once a character set has been assigned to *fineKey,* the variable is passed on to **procedure** fineString as a parameter: *fineString(s, fineKey);*.

The parameter list of the procedure call uses two parameters: *fineKey* passes the designated character set to the procedure, and *s* passes the string created in **procedure** fineString *back to* the procedure call.

The parameter *s* is a special kind of parameter that has not been used before. A parameter that passes data *back to* the procedure call is titled a ***var** parameter.* The reserved word **var** serves as the title for both variable and **var** parameter declarations.

The two kinds of parameters are sometimes called *value parameters* and *variable parameters.* The value parameter sends data *to* a block. The variable parameter (**var** parameter) sends data *back from* a block. Since the word *variable* is used so often in Pascal, you might find it less confusing to simply refer to the two kinds as:

1. the plain parameter, which needs no title and sends data to a block
2. the **var** parameter, which requires **var** in the block heading and returns data from the block to the block call

24.4 Procedure fineString (var s: string; fineKey: keys);

The *char* type variable *bs* is assigned the value *chr(8)* (8 is the ordinal number of the *backspace* key) in the same manner that the variable *sp* was assigned to the *space bar* key. Since the *backspace* key has no literal character of its own, it must be referenced by its ordinal number.

The string parameter *s* is initialized to an empty string, notated by two consecutive single quotation marks. The remainder of **procedure** fineString is contained in a **repeat..until** loop. The purpose of this loop is to add characters, one by one, to the **var** parameter *s*. The string's completion occurs when the *return* key has been pressed (and the *eoln* function becomes true).

In order to accomplish the construction of a string, three of Pascal's predefined string commands are used. The *length* function checks the number of characters in the parameter string *s* and if the string has no characters, the *backspace* key (bs) is not permitted. However, for any length of *s* other than zero, the *backspace* key is added to the set of characters in *fineKey.*

The variable *okayKey,* also of type *keys,* will be sent as the parameter to **function** checkChar. *okayKey* will be assigned the same value as *fineKey* if *s* is an empty string. Otherwise, *okayKey* will be assigned the value of *fineKey* plus the character *bs.* When *okayKey* includes *bs,* the program allows the user to backspace over any unwanted characters.

function checkChar is called from the instruction: *newChar := checkChar(okayKey);. checkChar* is the exact same function in **program** StringTesting as it was in **program** InputTesting. The *okayKey* parameter that is sent to the **function** determines which keypresses will be beeped and which will be written onto the screen, then assigned to variable *newChar.*

The last part of **procedure** fineString is an **if..then..else** condition that checks to see if the character returned from **function** checkChar is the *backspace* key. If *newChar = bs* then the *delete* procedure removes from string *s* one character beginning at the position given by *length(s).*

The *delete* procedure requires three parameters. The first is the string from which characters are to be deleted. The second is the numeric position of the first character to be deleted. The third is the number of characters to be deleted.

If *newChar* has been assigned any character other than the backspace character, that character is concatenated (added) onto string *s,* and *s* is assigned its new, one-character-longer value.

The *concat* function requires two or more parameters. The contents of each parameter are joined together and returned by the function as a single string value. The order in which the parameters are joined is the same order in which they appear in the parameter list.

In this way, each new character that has been tested by **function** checkChar is joined to the existing characters of string *s.* The construction of the string is completed when the *eoln* character, the *return* key, has been pressed.

procedure fineKey concludes and the **var** parameter *s* is returned from the **procedure** back to the procedure call in the main body. The string assigned to variable *s* can be used like any other string. The last instruction of the main body is a *writeln* statement that sends to the screen the string contents of *s* as well as the string literal 'only contains letters or spaces.'

Pascal offers many other predefined **procedures** and **functions** that help a programmer manipulate strings. You will find definitions and examples of these as you browse through the unfamiliar terms of Part 3.

But do not try to find the term *Split Ends* in any Pascal glossary. Only the *Fear and Loathing Guide* has the breadth to acknowledge the cruxes of the human condition.

There are many theories to the problem of split ends, whose nasty trademark is frizzy, unmanageable hair. Diet, weather, shampoos, and stress could all be contributing factors. Pascal offers no solution. Mr. Moss can only share the hope with others that modern science might one day find an answer to this follicular bane.

25 Formatting Dollars with Common Cents

What's next

This program presents numbers so that they appear on screen as dollar and cent amounts. Numeric data will be formatted to two decimal places, then converted to strings in order that dollars signs ($) and commas be inserted. The conversion occurs in a single procedure that can be relocated into any program.

■ Topics

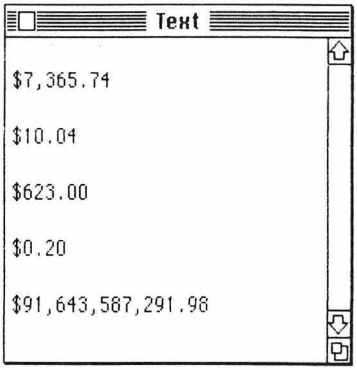
extended

stringOf

insert

25.1 Program DollarCents;

```
program DollarCents;
 var
  answer : array[1..5] of string[20];
  i, cost : integer;
 procedure convertMoney (amount : extended;
          var stAmount : string);
  const
   width = 15;
  var
   len : integer;
  begin
   stAmount := stringOf(amount : width : 2);
   while stAmount[1] = ' ' do
    delete(stAmount, 1, 1);
   len := length(stAmount);
   while len > 6 do
    begin
     insert(',', stAmount, len - 5);
     len := len - 3
    end;
   insert('$', stAmount, 1)
  end;
begin
  convertMoney(7365.74, answer[1]);
  convertMoney(10.035, answer[2]);
  cost := 623;
  convertMoney(cost, answer[3]);
  convertMoney(0.2, answer[4]);
  convertMoney(91643587291.98495, answer[5]);
  for i := 1 to 5 do
   begin
    writeln;
    writeln(answer[i]);
   end
end.
```



```

$7,365.74

$10.04

$623.00

$0.20

$91,643,587,291.98
```

25.2 Definitions and declarations

The global declarations are used only to show examples of the dollar and cents formatting. **procedure** convertMoney is self-contained, in that it does not use any global or outside references.

The variable *answer* is declared as type *array[1..5] of string[20];* The five elements of *answer* will be assigned the formatted dollar-string values by **procedure** convertMoney's **var** parameter. Remember, **var** parameters *send back* data from a block to the block call.

The constant *width* and the variable *len,* local to **procedure** convertMoney, are used because they add clarity. *width* refers to the number of character-spaces an expression will occupy when displayed on the screen. *len* substitutes for the cumbersome function call *length(stAmount).*

A new type, *extended,* is introduced in **program** DollarCents. The parameter *amount* in **procedure** convertMoney is declared to be of type *extended. extended* is one of MacPascal's four real-types that represent a range of real numbers, which for the purpose of DollarCents means numbers that allow the use of a decimal point.

See Part 3 under *real, extended, double,* and *computational* to find out more about real-types.

25.3 Main

Five of the first six instructions of the main body call **procedure** convertMoney. The first parameter of each call serves to send data *to* the procedure. The second parameter receives data *from* the procedure. If you look up to the heading of **procedure** convertMoney, you will see that the first parameter (which has no title) is *amount : extended;,* while the second parameter (titled **var**) is ***var** stAmount : string.*

The call to *convertMoney* will send a numeric value to the procedure, then take back the converted value as array elements of a *string* type.

The **for..to..do** loop at the end of the main body will display the formatted contents of the array *answer.*

The reason for storing the string results in an array (as opposed to creating five different variable names) is the ease with which a **for** loop can write out the contents of the array. The **for** loop's counter (i) serves the double purpose of the array's subscript ([i]). If five different variable names had been used, you would have needed to write out all five names in a *writeln* statement to display their contents.

25.4 Procedure convertMoney (amount : extended; var stAmount : string);

The first instruction of *convertMoney* converts the parameter *amount* from type *extended* to type *string: stAmount := stringOf(amount : width : 2);*.

The MacPascal function *stringOf* works similarly to the *write* procedure except that instead of displaying its parameter on the screen, *stringOf* returns the characters of its parameter as a string value. In DollarCents, this function will take a real-type number as its parameter and return a string value. *stAmount* will be assigned the function's string result.

The parameter of the *stringof* function deserves special attention. The real-type value *amount* has been appended with *(: width : 2)*. These are *colon modifiers* that can be applied to any *write, writeln, writeDraw,* or *stringOf* statement.

Colon modifers are *integer* type values that specify:

1. : minimum field width
2. : number of decimal places

The purpose of specifying field width and decimal places is to help a programmer format the display of output. If a list of output expressions all occupy the same number of character spaces, and are extended the same number of places to the right of the decimal point, much of the formatting task is already done.

Colon modifiers must be placed in the order shown (: field width : decimal places) and must have a value greater than zero. A decimal place modifier cannot be used without a field width modifier.

In DollarCents, the field width value of 15 ensures a minimum of 15 character spaces to represent the value of *amount.* The decimal place value of 2 causes the value of *amount* to be notated using two decimal places to the right of the decimal point.

Version 1.0 of MacPascal has a bug with some uses of colon modifiers. To sidestep this bug, do not use a *minimum* width modifier any smaller than the *actual* width of your largest expression. Though a minimum field width is supposed to accommodate enough characters as necessary to represent an expression's value, a field width smaller than an expression's actual width causes errors in the decimal places modifier.

Now that *amount* has been converted from type *extended* to type *string,* leading spaces can be deleted, and commas and a dollar sign can be inserted. Two **while..do** loops accomplish the leading spaces and commas task.

Leading spaces are deleted by the instructions:

```
while stAmount{1} = ' ' do
  delete(stAmount, 1, 1);
```

The subscript *{1}* of the string *stAmount* is a method for notating the first character of the string value. *stAmount{2}* represents the second character of *stAmount,* and so on.

Remember, the field width modifier of *amount* specified a minimum of 15 character spaces, yet most values of *amount* require far fewer spaces. As a result, the leading spaces of *stAmount* are blank. The **while** loop checks to see if the first character of *stAmount* is blank (' ') and if so, Pascal's *delete* procedure deletes that space character. The loop continues until all leading spaces are deleted.

The second **while** loop counts the number of characters in *stAmount* and inserts commas after every third character to the left of the decimal point. See if you can figure out how this loop works after reading the following note.

The predefined procedure *insert* requires three parameters. The first is the string to be inserted. The second is the result string, which will take in the insertion. The third is an integer, which determines the character position at which the first parameter will be inserted into the second parameter (an integer value of 1 will cause the first parameter to be inserted before the first character of the result string, a value of 2 will cause an insertion before the second character, and so on).

The last instruction of **procedure** convertMoney inserts a dollar sign into the result string *stAmount* before the first character. The *insert* procedure returns *stAmount* as its own **var** parameter, with the dollar sign appended.

This final value of *stAmount* is returned by **procedure** convertMoney to the procedure call in the main body. The **var** parameter *stAmount* is sent back to the procedure call as an array element of *answer.* The array elements, each of type *string,* are displayed on the Macintosh screen in their dollar and cents format.

Much of Mr. Moss's acumen can be attributed to his training at the University of Pennsylvania's Wharton School of Finance (Wanton, for short). Mr. Moss strongly urges anyone with the opportunity to pursue a higher education, not to squander it. The wealth of experience offered within a setting of young men and women living together, bonded by the pursuit of knowledge, far outweighs the occasional inconvenience of attending classes and reading books.

26 Financial Programming: To the Bone

What's next

The program in this chapter illustrates a common business application. A short invoice is prepared from the retail sale of merchandise. The computation involves items purchased, price, tax, and a pay-by-installment plan. You will use Pascal's arithmetic tools, a new type definition, and last chapter's *convertMoney* procedure to format a dollar amount.

■ Topics

type {string subset}

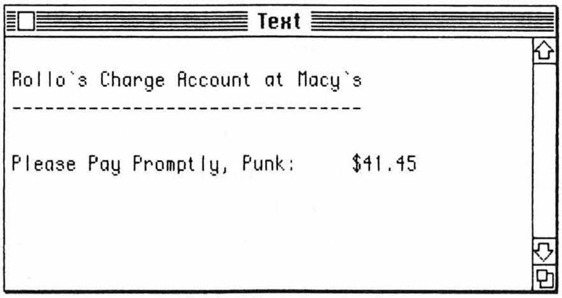
record

computational

[+,−,*,/]

26.1 Program RolloAtMacys;

```
program RolloAtMacys;
 type
  str20 = string[20];
 var
  amountDue : extended;
 procedure convertMoney (amount : extended;
          var stAmount : str20);
  const
   width = 15;
  var
   len : integer;
 begin
  stAmount := stringOf(amount : width : 2);
  while stAmount[1] = ' ' do
   delete(stAmount, 1, 1);
  len := length(stAmount);
  while len > 6 do
   begin
    insert(',', stAmount, len - 5);
    len := len - 3
   end;
  insert('$', stAmount, 1)
 end;
 procedure calculateBill (var amountDue : extended);
  const
   salesTax = 0.065;
   installments = 10;
  type
   boneRec = record
     units : computational;
     unitCost : computational
    end;
  var
   bones : boneRec;
   fleaPowder, subTotal : computational;
   total : extended;
 begin
  bones.units := 800;        {number of bones}
  bones.unitCost := 49;      {in cents ($0.49)}
  fleaPowder := 279;         {in cents ($2.79)}
  subTotal := (bones.units * bones.unitCost) - fleaPowder;
  total := (subTotal / 100) * (1.00 + salesTax);
  amountDue := total / installments;
 end;
 procedure printBill (amountDue : extended);
  var
   rolloOwes : str20;
 begin
  writeln;
  writeln('Rollo's Charge Account at Macy's');
  writeln('--------------------------------');
  writeln;
  write('Please Pay Promptly, Punk:   ');
  convertMoney(amountDue, rolloOwes);
  writeln(rolloOwes);
 end;
begin
 calculateBill(amountDue);
 printBill(amountDue)
end.
```



```
Rollo's Charge Account at Macy's
--------------------------------

Please Pay Promptly, Punk:       $41.45
```

26.2 Definitions and declarations

At the top of the program, beneath the **type** heading, is the definition: *str20* = ***string****[20];*. In this way, *str20* becomes a subset of the *string* type. Any variable declared of **type** *str20* must be a string of 20 or fewer characters.

Two purposes are served by creating **type** str20. First, it is shorter and easier to write *str20* than ***string****[20]*. Second, and more significantly, *str20* can now be used as a type in a parameter list. Pascal does not allow string subscripts in a parameter list so ***string****[20]* would not have been permitted. ***string*** alone would have been okay, except that its default length is 255 characters, which is a waste of space for variables that will never exceed 20 characters.

program RolloAtMacys introduces two other new words: **record** and *computational*. Both appear locally in the definition section of **procedure** calculateBill:

```
type
  boneRec = record
    units : computational;
    unitCost : computational
  end;
```

A **record** is a type composed of two or more fields, each with their own type. The fields of a record work like variables. You might think of a **record** as a suitcase full of assorted variables. The name of the **record** serves as the suitcase's handle.

Just like a suitcase, the purpose of a record is to keep your possessions (variable fields) organized when you move from one place to another. When two variables are intimately associated with one another, you should consider joining them as fields of a **record**.

procedure calculateBill has done this. The fields *units* and *unitCost* represent the number purchased of a piece of merchandise and the price of that merchandise. When these fields are multiplied, a dollar amount due for the purchase is derived.

The fields of a record can be accessed individually. First, a variable must be declared of the record type. Second, the field name must be appended to the variable name: *recordVariableName.fieldName.*

A single period separates the variable name from the field name. The first two instructions of **procedure** calculateBill illustrate this notation as both fields of the **record** boneRec are assigned values. Notice that under **var**, the variable *bones* has been declared of type *boneRec.*

Both fields of **record** boneRec are of type *computational. computational* is one of MacPascal's predefined real-types, designed specifically for financial applications. The computational type offers a large range of values without any loss of precision. See Part 3 for more information on MacPascal's four real-types.

26.3 Main

The main body of RolloAtMacy's consists of two procedure calls. Both calls use a single parameter. **procedure** calculateBill returns as a **var** parameter *amountDue.* **procedure** printBill takes the value of *amountDue* and displays the formatted value in an invoice.

Borrowing from the renowned economist Wimpy, Rollo offers this insight to the infrastructure of modern civilization:

> I would gladly pay you Tuesday for a bone today.

26.4 Procedure calculateBill (var amountDue : extended);

This procedure assigns values to the record fields of *bones* and the variable *fleaPowder,* then performs simple arithmetic to determine the invoice's *amountDue.* The first three assignments have comments to explain the values being assigned. You should notice that computational values do not use decimal points, therefore all money figures are represented in cents.

subTotal is assigned the product of the record field *units* and the record field *unitCost,* less the cost of *fleaPowder. fleaPowder* is subtracted from the cost of bones because Rollo is returning the flea powder and Macy's is giving credit toward the purchase of bones.

total is assigned the product of a division and an addition. First, *subTotal* is divided by 100 (in order to convert from type *computational* cents to type *extended* dollars). Second, *salesTax* is added to 1. Third, the results of the division and the addition are multiplied to produce a dollar value with tax added.

amountDue is assigned the *total* bill divided by the number of *installments. amountDue* is then returned as the **var** parameter of **procedure** calculateBill.

26.5 Procedure printBill (amountDue : extended);

printBill consists of six *writeln* statements that display invoice information in the Text window. The only other instruction in *printBill* is the procedure call to *convertMoney,* the same procedure used in the last chapter to format numbers into a dollar format.

The call to *convertMoney* uses two parameters. The first sends the *extended* type *amountDue* to the procedure. *convertMoney* returns a string as its **var** parameter. This string value returns to the procedure call with the name of the second parameter, *rolloOwes.* In a dollar and cents format, *rolloOwes* is written to the Text window.

27 A Time for Mac, a Time for Martha

What's next

This program shows you how to communicate date and time information between the Macintosh clock and your Pascal programs. The Text window will display a digital display of date and time, while the Drawing window will display the second hand of an analog clock face.

■ Topics

dateTimeRec

with..do

writeln {with modifiers}

insetRect

fillOval

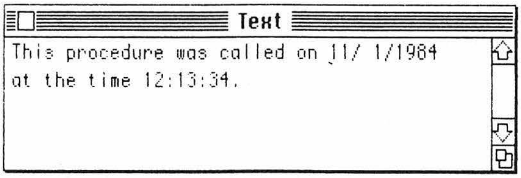
tickCount

fillArc

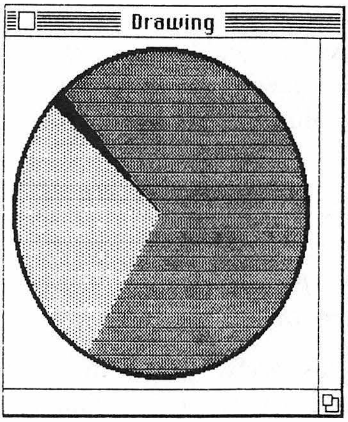
paintArc

27.1 Program SecondHand;

```
program SecondHand;
  var
    currentTime : dateTimeRec;
  procedure textDisplay (currentTime : dateTimeRec);
  begin
    with currentTime do
      begin
        writeln('This procedure was called on ', month : 2, '/', day : 2, '/', year : 2);
        writeln('at the time ', hour : 2, ':', minute : 2, ':', second : 2, '.');
        writeln
      end
  end;
  procedure drawingDisplay;
    const
      sixDegree = 6;
      ticks = 3600;
    var
      face : rect;
      perSec, arcSec : integer;
      starter : longint;
  begin
    setRect(face, 5, 5, 195, 195);
    penSize(3, 3);
    frameOval(face);
    insetRect(face, 3, 3);
    fillOval(face, gray);
    perSec := 0;
    starter := tickCount;
    while not (tickCount = starter + ticks) do
      begin
        getTime(currenttime);
        if currentTime.second <> perSec then
          begin
            perSec := currentTime.second;
            arcSec := perSec * sixDegree;
            fillArc(face, arcSec - sixDegree, sixDegree, ltGray);
            paintArc(face, arcSec, sixDegree)
          end       {end if..then}
      end        {end while..do loop}
  end;      {end procedure drawingDisplay}
begin
  getTime(currentTime);
  textDisplay(currentTime);
  drawingDisplay;
  textDisplay(currentTime)
end.
```



```
This procedure was called on 11/ 1/1984
at the time 12:13:34.
```



27.2 Definitions and declarations

program SecondHand uses a new Toolbox data type called *dateTimeRec.* Since the type *dateTimeRec* has already been defined as a Toolbox **record** type, a Pascal program can simply declare a variable of that type. After that, any field of the record can be accessed just as the **record** boneRec was accessed in the last chapter.

Of course, you need to know the field names of *dateTimeRec* in order to access them. You could find this information in Part 3, but here they are anyway: *year, month, day, hour, minute, second, dayOfWeek : integer.*

All the fields of the **record** type *dateTimeRec* are of type *integer.* The integers that represent date and time data are easy to interpret. For example, the *month* field uses 1 to represent January, 2 to represent February, and so on. The *year* field contains all four digits of the year. The *dayOfWeek* field gives a number from 1 to 7, representing Sunday through Saturday.

The variable *currentTime* is declared of type *dateTimeRec.* The fields of *dateTimeRec* can be accessed by using the notation *currentTime.fieldName* where *fieldName* is replaced by *month, day, hour,* etc.

The variable *face* is declared of type *rect.* This will be used to set the oval for the clock face. You have used type *rect* variables many times before. What you might not have known is that the *rect* type, like *dateTimeRect* is a **record** type. *rect* is a Quickdraw **record** of four integer fields—*top, left, bottom,* and *right.* You could access any field of a *rect* type using the notation *rectVariableName.fieldName.* For instance, *face.bottom* has been assigned the value of 195 in the *setRect* command.

Quickdraw's *rect* type is actually a variant record defined either as four integers as stated above or two fields of type *point.* The fields of type *point* are *topLeft* and *botRight,* corresponding to the corner points of the rectangle.

The variables *perSec* and *arcSec* are assigned field values of *currentTime. currentTime* is assigned its value (all field values at once) as the **var** parameter of the Toolbox's *getTime* command.

The variable *starter* is declared of type *longint.* This variable is assigned the value returned by the Toolbox function *tickCount. tickCount* is a timer function that measures elapsed time in sixtieths of a second.

27.3 Main

The main body consists of four procedure calls. The first call is to the predefined Toolbox procedure *getTime.* The remaining three calls are to procedures *textDisplay* and *drawingDisplay.* Notice that *textDisplay* is called twice, once before and once after the call to *drawingDisplay.* By watching the Text window, you can see that the parameter *currentTime* sends new data on the second call.

The call *getTime(currentTime)* assigns the fields of the **record** *dateTimeRec* to the **var** variable *currentTime* with data from Macintosh's internal clock. In the next instruction, the field values contained in *currentTime* are passed as a parameter to **procedure** textDisplay.

There is also a MacPascal procedure to *set* the data of the Macintosh clock called *setTime.* You can read about this procedure in Part 3, though more likely you would want to use the Control Panel option from the Apple menu to set the date and time.

No matter the perspective, it seems that time tends to make people old, that old tends to make people weak, and that weak tends to make people die. For its cruel tendencies, time must have something fun to offer in return.

Mr. Moss would hate to become philosophical this late in the text, therefore he has no choice but to relate his plans for tomorrow night with his girlfriend. At seven-thirty, they will meet at his house, then walk to an ice cream store a half mile away. They don't know what they will talk about, yet they know it will be interesting because each likes the way the other talks. At home, after the ice cream has been finished, they will touch each other, and at a point in their touching, before they go to sleep, time will fizz and tingle and pay back all it owes.

27.4 Procedure textDisplay (currentTime : dateTimeRec);

The first instruction of *textDisplay* introduces **with**, a new Pascal reserved word: ***with** currentTime do.*

Beneath this statement are three *writeln* instructions bracketed by **begin** and **end.** Notice that the *writeln* statements use the field names of **record** dateTimeRect without referencing the variable name *currentTime.* The reason the prefix *currentTime* (and the separating period) has been omitted is because the **with..do** instruction automatically affixes the variable name *currentTime* to all the field names within the **begin..end** brackets.

The **with..do** command is simply a shortcut. The same effect could be accomplished by omitting ***with** currentTime **do,*** and instead using the full *currentTime.fieldName* notation. However, you should be able to see that **procedure** textDisplay would be quite a bit lengthier (and less easy to read) without the **with..do** shortcut. If only one field of *dateTimeRec* was being accessed, it would be shorter to affix *currentTime* to the field name.

The *writeln* statements of **procedure** textDisplay also make use of the colon modifiers. Only the first modifier, specifying minimum width, is necessary since all field values are integers, and there are no decimal places. If the width modifier is omitted, the integers will make up 6 character spaces, the default width size used by MacPascal and the text display would look awkward.

27.5 Procedure drawingDisplay;

The first five instructions of *drawingDisplay* call Quickdraw procedures in order to frame a thick oval, then fill inside the frame with a gray pattern. The *setRect* command sets the rectangle into which the oval frame will fit. The *penSize* command sets the black line thickness of the oval frame. *frameOval* draws the oval. *insetRect* takes the type *rect* variable *face* and shrinks it, horizontally and vertically, by the dimensions given in the second and third parameters respectively. Then the *fillOval* command fills the area within, but not covering, the framed oval.

The instruction *perSec := 0;* is an initialization. The variable *perSec* will be used if the *second* field of the **record** has changed since the previous *getTime* call.

The variable *starter* is assigned the value returned by the *tickCount* function. Though *starter* will represent the time elapsed (in sixtieths of a second) since the Macintosh was started, the usefulness of *starter* is its use as a base. When a subsequent call to *tickCount* returns a value 3600 (the constant *ticks*) larger than *starter,* the **while not..do** loop is exited.

The **while not..do** loop is repeated for as long as it takes *tickCount* to increment by 3600 ticks (about 60 seconds). In this time, *getTime* is called and the field values are assigned to *currentTime.* Each time a new value of the field *currentTime.second* is detected (by comparing it to *perSec*), the four instructions of the **if..then** condition are performed.

The **if..then** condition updates *perSec,* then assigns an integer value to *arcSec* based on the number of elapsed seconds multiplied by the constant *sixDegree.* The value of *arcSec* will determine the number of degrees at which an arc should be drawn. This simulates the position of a clock's second hand. Each increment of 6 degrees corresponds to 1 second (6 degrees * 60 seconds = 360 degrees per minute).

The last two instructions of **procedure** drawingDisplay create the sweep motion of the clock's second hand. The previous second's arc is filled by *fillArc,* then the current second's arc is painted by *paintArc.*

All the *arc* commands—*frame, paint, invert, erase,* and *fill*—require two degree parameters. These determine the starting angle of the arc and the number of degrees it will extend. The second and third parameters of *fillArc* and *paintArc* accomplish this task. The *fillArc* command has a fourth parameter which determines the fill pattern.

The arc-drawing commands should draw sixty arcs, or one complete revolution, before the *tickCount* function returns a value equal to *starter* plus *ticks.* At this point, the loop and procedure are exited. The main body calls **procedure** textDisplay again, updating the Text window with the most recent date and time data received from *getTime.* The new Text window display should show the time one minute later than the first display.

28 Files for the Child in You

What's next

This chapter's program presents an illustrated filing system using fast, random-access commands. The spirit of Macintosh graphics is upheld as the user retrieves file data by pointing and clicking the mouse.

Topics

record {of record}

array {of record}

file of

open

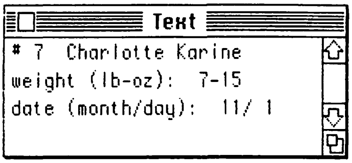
seek

filepos

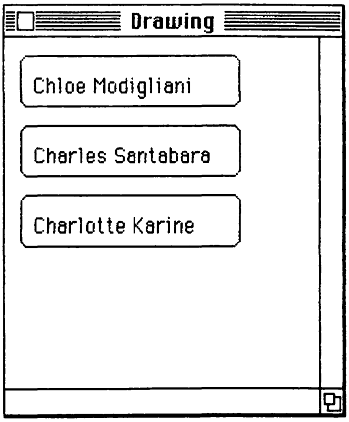
28.1 Program BabiesAreUs;

```
program BabiesAreUs;
 const
  cribs = 10;
  empty = '';
 type
  babyRec = record
     name : string[30];
     weight : string[5];
     time : dateTimeRec
    end;
```

```
var
  baby : array[0..cribs] of babyRec;
  dataName : string[50];
  babyFile : file of babyRec;
procedure addRecord;
  var
    currentTime : dateTimeRec;
begin
  getTime(currentTime);
  baby[0].name := '';
  baby[0].weight := '';
  baby[0].time.month := 0;
  baby[0].time.day := 0;
  baby[1].name := 'Chloe Modigliani';
  baby[1].weight := '7-7';
  baby[1].time.month := currentTime.month;
  baby[1].time.day := currentTime.day;
  baby[4].name := 'Charles Santabara';
  baby[4].weight := '6-8';
  baby[4].time.month := currentTime.month;
  baby[4].time.day := currentTime.day;
  baby[7].name := 'Charlotte Karine';
  baby[7].weight := '7-15';
  baby[7].time.month := currentTime.month;
  baby[7].time.day := currentTime.day
end;
procedure insertRecord;
  const
    instruct = 'type a new datafile name';
  var
    i : integer;
begin
  dataName := newFileName(instruct);
  open(babyFile, dataName);
  for i := 0 to cribs do
    begin
      seek(babyFile, i);
      if baby[i].name <> empty then     {empty string constant ''}
        babyFile^ := baby[i]
      else
        babyFile^ := baby[0];
      put(babyFile)
    end;    {end for loop}
  close(babyFile)
end;    {end procedure insertRecord}
procedure retrieveRecord;
  const
    instruct = 'select the file to read';
    var
      line, i, x, y : integer;
      doBox : array[1..cribs] of rect;
      pt : point;
      inBox : boolean;
      tot : babyRec;
  begin
    line := 30;
    dataName := oldFileName(instruct);
    open(babyFile, dataName);
    for i := 1 to cribs do
      begin
        seek(babyFile, i);
        if babyFile^.name <> empty then
          begin
            setRect(doBox[i], 10, line - 20, 150, line + 10);
            line := line + 40;
            frameRoundRect(doBox[i], 11, 11);
            moveTo(doBox[i].left + 8, doBox[i].bottom - 8);
            writeDraw(babyFile^.name)
          end
      end;
    repeat
      getMouse(x, y);
      setPt(pt, x, y);
      if button then
        begin
          i := 0;
          repeat
            i := i + 1;
            inBox := ptInRect(pt, doBox[i]);
          until inBox or (i = cribs);
          if not inBox then
            sysBeep(7)
        end
    until inBox;
    seek(babyFile, i);
    tot := babyFile^;
    write('*', filepos(babyFile) : 2, ' ');     {writes out file number}
    with tot do
      begin
        writeln(name);
        writeln('weight (lb-oz):  ', weight);
        writeln('date (month/day):  ', time.month : 2, '/', time.day : 2)
      end;
    close(babyFile)
  end;
begin
  addRecord;
  insertRecord;
  retrieveRecord
end.
```



```
* 7  Charlotte Karine
weight (lb-oz):  7-15
date (month/day):  11/ 1
```



```
Chloe Modigliani
Charles Santabara
Charlotte Karine
```

28.2 Definitions and declarations

Under the global **type** heading is the record definition of *babyRec.* The third field of *babyRec* is *time,* whose type is the predefined **record** type *dateTimeRec.* Hence, you have defined a **record** of a **record** type.

This means that each value of **type** babyRec will contain all the field values (*year, month, hour,* etc.) of type *dateTimeRec.* A variable of **type** babyRec will also contain the two *string* types, *name* and *weight.*

Pascal allows all sorts of nestings such as the above. In BabiesAreUs the variable *baby* is an **array** of a **record.** Each element of the **array** body is a **record** defined by *babyRec.*

The nesting of Pascal's structures is not easy for a beginner to understand. You can reread the definitions till your head spins, but the assignments in **procedure** addRecord should make the nesting process more clear.

Understanding assignments is easier than understanding structures. For example, you will probably get lost trying to figure the value of the *month* field of the predefined **record** dateTimeRec of the program-defined **record** babyRec contained in the **array** baby. But by studying **procedure** addRecord's assignments (with the period notation separating fields), you can work backward from the actual data itself to the Pascal structures that hold it.

program BabiesAreUs contains a declaration of a new variable type: *babyFile : file of babyRec;*.

The reserved words **file of** serve to connect a variable name with an external device, in this case, the disk drive. The declaration makes *babyFile* the file organizer through which data is communicated to and from disks. The data organized through the variable *babyFile* will be of type *babyRec,* but the primary significance of the **file of** declaration is the association with the disk drive.

Files can be created of any Pascal type except other files. Files of *integer, string, array,* and *record* types all use the same format: **file of** type. There is one special-purpose file-type you have already used in previous chapters called *text.* Since *text* is a standard, predefined file-type, the reserved words **file of** are not used in its declaration.

28.3 Main

The main body contains three simple procedure calls. Their names help indicate their tasks. In the first procedure, data is added to the record fields through assignment statements. In a general use application, a programmer would prompt the user to input data through *read* commands, or better yet, through the string-testing routine illustrated in Chapter 24.

The second procedure inserts the filled-in records into a file on disk and creates a graphic display for each insertion. The name of the datafile is selected by the user through use of the Toolbox file procedure *newFileName.* Since *BabiesAreUs* automatically calls *newFileName* in order to create a new file, a major enhancement to the program would be to ask the user to choose between creating a new datafile or add/delete/change/observe an existing datafile.

The third procedure presents a graphic, titled display of the available records, waits for the user to make a selection with the mouse, then retrieves the selected data using a fast, random-access retrieval instruction. The program ends after the data has been displayed in the Text window, though an easy enhancement allows the user to select other records to view before exiting the program.

28.4 Procedure addRecord;

The first instruction of *addRecord* fills the fields of the *currentTime* with data gotten from the Macintosh clock. Only the *month* and *day* fields will be used in *addRecord.* The other date and time fields will be ignored.

The remaining instructions assign values to four elements of the **array** baby. Notice that the first expression of each assignment is: *baby{subscript}.*

The subscript distinguishes which element of the array is being assigned a value. In this example, the element of the array is composed of a **record**. Thus, element and **record** refer to the same data.

All assignments with the subscript of [0] belong to the same array element and **record**. The assignments with the subscript of [1] belong to another array element/**record,** and so on. Though the array has been declared to allow for 11 elements [O..crib], **procedure** addRecord only fills in four of these elements.

The subscripts are given nonconsecutive numbers to illustrate a random-access filing system. You will find out more about this in the next section.

The *name, weight,* and *time* field names in *addRecord* distinguish which field of the **record** is to be assigned a value. You saw the period notation used in the last two chapters. In the case of the *dateTimeRec* fields, the period notation must be used twice: once to separate the record name from the *time* field, and again to separate the *time* field from the *month* and *day* fields.

The right side of the date assignments are values returned by the *getTime* procedure and its *currentTime* parameter. These will be integer values, as all of the predefined record *dateTimeRec* are of type *integer.*

28.5 Procedure InsertRecord;

The file system used in *insertRecord* differs from the type *text* files you used in earlier chapters in one major respect. The file organizer *babyFile* is able to directly access a component by first seeking a component position in the file, then inserting or retrieving the data through the *babyFile^* buffer variable.

The file organizer's buffer variable is the crucial connection between your program and the data stored on disk. The buffer variable has the same name as the file organizer, with a caret sign (^) appended to the last character. The nature of files is such that only one component of a file can be accessed at a time, and this is accomplished by positioning the file organizer along the file, then assigning data to and from the file organizer's buffer variable.

The Toolbox procedure *newFileName* creates a disk datafile and produces a dialog box prompting the program user (with the *instruct* string parameter) to make up a datafile name. The datafile is then opened with the instruction: *open(babyFile, dataName);*.

The *open* instruction is similar to the *rewrite* and *reset* instructions you used in earlier chapters. *open* makes the connection between the file organizer variable *babyFile* and the disk datafile *dataName.*

open opens both new and existing files for random access to the file's components. *rewrite* opens new files for sequential access, while *reset* opens existing files for sequential access. All three procedures serve the purpose of connecting the file organizer variable name (first parameter) to the datafile name (second parameter). See Part 3 for more specific information and additional examples.

The first command in the***for** i := 0 to cribs **do*** loop is: *seek(babyFile, i);*.

This command positions the file organizer *babyFile* to the i^{th} position (component place) of the disk file. The first repetition of the **for** loop will **seek** the 0^{th} position (first component place) of the file.

The conditional actions following *if baby{i}.name <> empty **then*** assign a value to the file organizer's buffer variable, *babyFile^*. If *baby{i}.name* has a value that *is not equal (<>)* to the constant *empty* (")—that is, a name has been assigned to record field *name* for the *i^{th}* element of the **array** baby—then the entire record value of *baby{i}* is assigned to the buffer variable *babyFile^.*

The **else** action of the **if..then** condition (performed when no name is in the record's *name* field) assigns initializing or empty values to the buffer variable. The initializing values, shown in the first four assignments of **procedure** addRecord in the *0^{th}* element of the *baby* array, are necessary to fill the unused spaces in a file so that the random-access process can keep count of component positions.

The last instruction of the **for..to..do** loop is: *put(babyFile).*

This inserts the contents of the buffer variable into the file at the current component position. The parameter of Pascal's *put* procedure is the file organizer (*babyFile*), not the organizer's buffer variable (*babyFile^*). Since Pascal's files can only access one component at a time, the file commands *put* and *get* operate implicitly on the buffer variable.

The **for..to..do** loop performs the *seek,* the conditional *babyFile^* assignment, and the *put* instructions for all values of *i* from 0 to *cribs* (a constant equated to 10). In this way, **procedure** insertRecord creates a disk file with eleven components, three of which are assigned informative record data.

28.6 Procedure retrieveData;

The second instruction of *retrieveData* calls the Toolbox procedure *oldFileName.* This produces a dialog box on the screen prompting the user (with the *instruct* string parameter) to select a datafile name. In **program** BabiesAreUs, the user should select the same datafile name that the user assigned in the *newFileName* call of **procedure** insertRecord.

The *open(babyFile, dataName)* command is identical to that used in **procedure** insertRecord. Remember, *open,* unlike *rewrite* and *reset,* is used to open both new *and* existing datafiles, and the files' components are randomly accessed. A connection is made between the file organizer *babyName* and the disk datafile *dataName.*

The loop ***for** i := 1 **to** cribs **do*** creates a graphic display in the Drawing window by:

1. using *seek* to advance the file organizer
2. testing whether the *name* field of the buffer variable (*babyFile^.name*) is an empty string

3. if the *name* field is not (<>) empty, then setting and drawing a rounded rectangle with the field's string value written inside.

Each rectangle created by the *setRect* command is an **array** element given the name *doBox{i}.* The integer *i* serves both as the *seek* command's file position parameter and the rectangle's **array** subscript. In this way, the *babyRec* data and its graphic display are linked.

The integer variable *line* is used to determine the top and bottom sides of each rectangle. After each *setRect* command, *line* is incremented so that the new rectangle is positioned below the prior rectangle.

The *moveTo* and *writeDraw* commands in the **for** loop insert the *babyRec* field *name* within the framed rectangles. Notice that the *moveTo* parameters use the *variableName.fieldName* notation to access the *left* and *bottom* fields of the type *rect* **record** of four boundary integers.

The **repeat..until** loop reads the mouse action of the user. A second **repeat..until** loop inside the first determines (by the value of *i* upon the loop's exit) within which rectangle in the array of *doBox* rectangles the mouse button has been pressed. The *ptlnRect* function returns a true value only when the mouse button has been pressed with the mouse pointing inside a drawn rectangle.

When *inBox* has been found to have a true value, the outer **repeat..until** loop is exited. The subscript of *doBox{i}* is used by the instruction *seek(babyFile, i);* to position the file organizer to the selected component position.

At this point, the organizer's buffer variable, *babyFile^*, is assigned to the type *babyRec* variable *tot.* Now the data contained in the selected file component can be displayed in the Text window.

The first *write* instruction displays the component file position. The MacPascal function *filepos* returns an integer noting the *i*th position of the file organizer. The buffer variable contains the data of the component indicated by *filepos.*

The other three *writeln* statements of **procedure** retrieveRecord are bracketed following a **with..do** command. The use of **with..do** provides a shortcut so that the variable *tot* does not need to be affixed before every field name of the *tot* record. The colon modifiers are added to the *month* and *day* fields so that the Text window does not display extra character spaces before each integer.

The purpose of assigning the buffer variable *babyFile^* to the variable *tot* is that the **with..do** structure of MacPascal does not allow the buffer variable to be used as the **record** variable.

The last instruction of **procedure** retrieveRecord, *close(babyFile),* closes the connection between the file organizer and the datafile. Closing a datafile is good practice even when the program is about to end, and all files will be automatically closed.

Part 2 ends here.

Part 3's programs reveal more from Mr. Moss, but his textual heresy has found a comfortable place to exit. Enjoy your programming and your time **not** programming. See if you can make your Macintosh do something nice for someone else.

Create a program to make a child smile. Then, when you really have your act together, create a child to smile at your program.

PART 3

Referencing MacPascal: The Whiz Kid's Encyclopedic Guide

Part 3 contains a dictionary of Macintosh Pascal including all of the Quickdraw and Toolbox routines used in the *Fear and Loathing Guide.* Most entries are illustrated by program examples.

The dictionary is presented as a single alphabetic listing. Since MacPascal takes its vocabulary from many sources, each entry is categorized.

The MacPascal vocabulary consists of:

- Pascal reserved words, procedures, functions, types, and constants
- UCSD Pascal procedure, function, type, and constant extensions
- MacPascal procedure and function extensions
- Quickdraw procedures and functions
- Toolbox procedures and functions

Also included at the front of Part 3 are five short references:

- a guide for programming in new frontiers
- an outline of Pascal's structures
- a table of the Macintosh character set
- a table of Imagewriter printer codes
- a listing by category of the MacPascal vocabulary and symbols

a New Frontiers, or Are We Having Fun Yet?

Don't you hate it when you work your butt off to learn something new, only to have some clod say, "You've only scratched the surface?"

The *Fear and Loathing Guide* deplores such condescension. An authority no less than Rollo goads you on for a second helping while recognizing your accomplishments:

> You have taken a chunk out of the postman's leg.

Mr. Moss recommends this path along the Macintosh frontier:

- Write programs. The more you learn about programming, the more astounded you become at how *little* nearly everyone else knows. Computer programmers write programs; computer talkers trade faulty information on subjects they do not understand.
- Prepare yourself for *pointers.* Variables of a pointer type are used for dynamic memory allocation. Pointers give data a fluidity that makes programs more efficient. Every other introductory Pascal book waits till the last chapter to explain pointers and fails in the attempt. The best way to understand

pointers is to first understand how a computer works. See the last item in this list for the best way to learn about how a computer works.

- Explore Quickdraw and the Toolbox. The programs in this book take up more computer space than the Macintosh ROM, but don't be misled. The Quickdraw and Toolbox routines built in the ROM can keep you fascinated for years, and represent an elegance of programming technique you will find nowhere else. The *MacPascal Reference Manual* will clue you to new vistas of Quickdraw and the Toolbox. Down the road, consider perusing the bible of the Macintosh ROM, *Inside Macintosh.* Written at Apple, it is long, dry, difficult, and excellent.

- Get your paws on *Macintosh 68000 Assembly Illustrated: The Fear and Loathing Guide.* People tend to be scared of assembly language because programs use fewer English words, but the advantages are considerable. The most important is: assembly language offers beauty and grace; it is the language closest to the heart of the computer, showing you how a computer works and allowing you to touch the flesh of the beast.

b Pascal Structures

Program structure

```
program title;
   uses      list;
   label     list;
   const     list;
   type      list;
   var       list;
   procedure or function title;
     begin
        statements;
     end;
begin
     statements;   {main body}
end.
```

Statement structures

```
case expression of
   label constant:
      statements;
   label constant:
      statements;
   otherwise      {optional}
      statements;
end;

for variable := expression to expression do   {downto
can replace to}
   statements;
```

```
if boolean expression then
  statements;
else     {optional}
  statements;

repeat
  statements;
until boolean expression;

while boolean expression do
  statements;

with record name
    do fieldName statements;
```

Definition structures

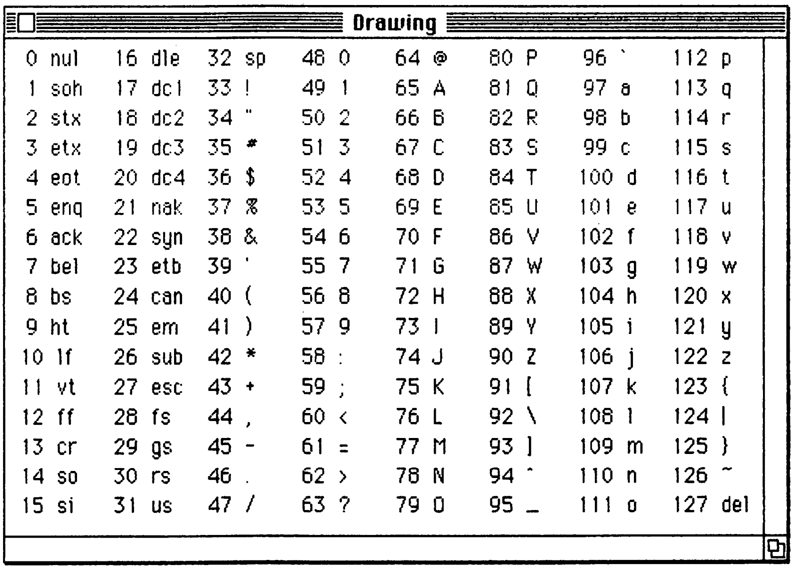
```
uses
  SANE;
  Quickdraw2;

const
  age = 31;
  experience = 'silly willy university';

type
  friend = string[50];
  friendList = array[1..8] of str1;
  play = record
          name : friend
          interests : string;
          phone : integer
        end;
  purpleBook = array[1..24] of play;
  purpleDisk = file of play;
  neatFriends = (michael, andy, susan, martha,
    karen);  {enumerated list}
  womenFriends = susan..karen;  {subrange}
  fairGame = 18..48;  {subrange}
  purpleFriends = set of neatFriends;
  purplePeople = array[subrange,'a'..'z'] of friend;
```

Declaration structures

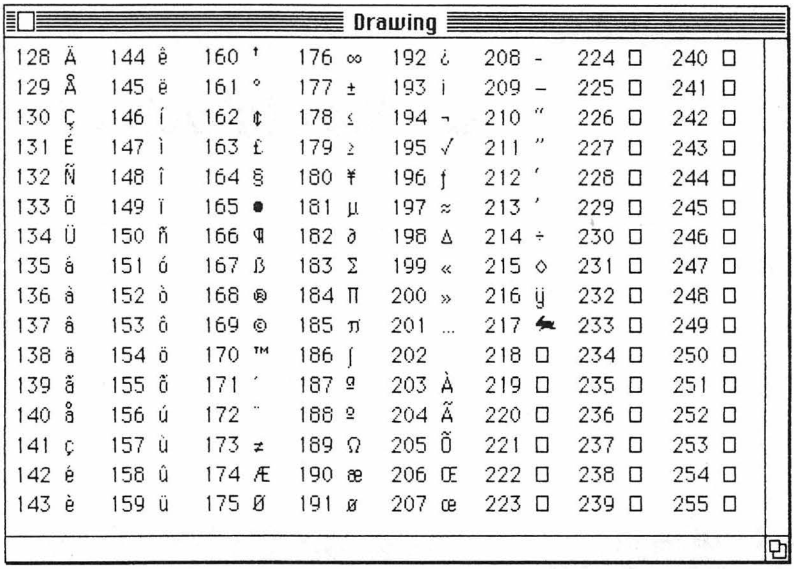
```
var
  musician: string[40];
  weight : integer;
  height : real;
  skier : friend;
  playVar : play;
  bookVar : purpleBook;
  diskVar : purpleDisk
  programmer : neatFriends;
  partner : womenFriends;

procedure doSomething;   {parameters optional}
  statements;

function returnSomething;   {parameters optional}
  statements;
```

c Macintosh Character Set

ord value in decimal—ASCII name



0	nul	16	dle	32	sp	48	0	64	@	80	P	96	`	112	p
1	soh	17	dc1	33	!	49	1	65	A	81	Q	97	a	113	q
2	stx	18	dc2	34	"	50	2	66	B	82	R	98	b	114	r
3	etx	19	dc3	35	#	51	3	67	C	83	S	99	c	115	s
4	eot	20	dc4	36	$	52	4	68	D	84	T	100	d	116	t
5	enq	21	nak	37	%	53	5	69	E	85	U	101	e	117	u
6	ack	22	syn	38	&	54	6	70	F	86	V	102	f	118	v
7	bel	23	etb	39	'	55	7	71	G	87	W	103	g	119	w
8	bs	24	can	40	(	56	8	72	H	88	X	104	h	120	x
9	ht	25	em	41	)	57	9	73	I	89	Y	105	i	121	y
10	lf	26	sub	42	*	58	:	74	J	90	Z	106	j	122	z
11	vt	27	esc	43	+	59	;	75	K	91	[	107	k	123	{
12	ff	28	fs	44	,	60	<	76	L	92	\	108	l	124	|
13	cr	29	gs	45	-	61	=	77	M	93	]	109	m	125	}
14	so	30	rs	46	.	62	>	78	N	94	^	110	n	126	~
15	si	31	us	47	/	63	?	79	O	95	_	111	o	127	del



Drawing

```
128 Ä   144 ê   160 †   176 ∞   192 ¿   208 –   224 □   240 □
129 Å   145 ë   161 °   177 ±   193 ¡   209 —   225 □   241 □
130 Ç   146 í   162 ¢   178 ≤   194 ¬   210 “   226 □   242 □
131 É   147 ì   163 £   179 ≥   195 √   211 ”   227 □   243 □
132 Ñ   148 î   164 §   180 ¥   196 ƒ   212 ‘   228 □   244 □
133 Ö   149 ï   165 ●   181 µ   197 ≈   213 ’   229 □   245 □
134 Ü   150 ñ   166 ¶   182 ∂   198 ∆   214 ÷   230 □   246 □
135 á   151 ó   167 ß   183 Σ   199 «   215 ◊   231 □   247 □
136 à   152 ò   168 ®   184 Π   200 »   216 ÿ   232 □   248 □
137 â   153 ô   169 ©   185 π   201 …   217 🐇   233 □   249 □
138 ä   154 ö   170 ™   186 ∫   202     218 □   234 □   250 □
139 ã   155 õ   171 ´   187 ª   203 À   219 □   235 □   251 □
140 å   156 ú   172 ¨   188 º   204 Ã   220 □   236 □   252 □
141 ç   157 ù   173 ≠   189 Ω   205 Õ   221 □   237 □   253 □
142 é   158 û   174 Æ   190 æ   206 Œ   222 □   238 □   254 □
143 è   159 ü   175 Ø   191 ø   207 œ   223 □   239 □   255 □
```

d Printer Control Codes

A complete list of Apple Imagewriter control codes can be found in the *Imagewriter User's Manual.*

(d) denotes the default setting.

Name	Decimal	Type/Spacing
ESCAPE p	27 112	Pica proportional
ESCAPE P	27 80	Elite proportional
ESCAPE n	27 110	Extended {9 characters per inch}
ESCAPE N	27 78	Pica {10 characters per inch}
ESCAPE E	27 69	Elite {12 characters per inch} (d)
ESCAPE e	27 101	Semicondensed {13.4 characters per inch}
ESCAPE q	27 113	Condensed {15 characters per inch}
ESCAPE Q	27 81	Ultracondensed {17 characters per inch}
ESCAPE !	27 33	Starts boldface printing
ESCAPE "	27 34	Ends boldface printing (d)
ESCAPE X	27 88	Starts underlining text
ESCAPE Y	27 89	Ends underlining text (d)
CONTROL-N	14	Starts headline mode
CONTROL-O	15	Ends headline mode (d)
ESCAPE A	27 65	6 lines per vertical inch (d)

ESCAPE B	27 66	8 lines per vertical inch
CONTROL-_n	31 n	Scrolls n lines of blank paper {n = 1,2,3,4,5,6,7,8,9,:,;,<,=,>,?}
CONTROL-L	12	Scrolls paper to next top of form
ESCAPE L nnn	27 76 nnn	Sets left margin at position nnn
ESCAPE c	27 99	Resets to default settings (d) {Software Reset}

e Vocabulary and Symbols

Pascal reserved words

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

Pascal files

input, output

Pascal constants

false, maxint, maxlongint, true

Pascal types

boolean, char, computational, double, extended, integer, longint, real, text

Pascal procedures

close, get, insert, open, pack, page, put, read, readln, reset, rewrite, seek, unpack, write, writeln

Pascal functions

abs, arctan, chr, concat, copy, cos, delete, eof, eoln, exp, filepos, include, length, ln, odd, omit, ord, ord4, pos, pred, round, sin, sqr, sqrt, stringOf, succ, trunc

Macintosh units

Quickdraw1, Quickdraw2, SANE

Quickdraw procedures

addPt, drawLine, drawString, eraseArc, eraseOval, eraseRect, eraseRoundRect, fillArc, fillOval, fillRect, fillRoundRect, frameArc, frameOval, frameRect, frameRoundRect, getPen, globalToLocal, insetRect, invertArc, invertCircle, invertOval, invertRect, invertRoundRect, line, lineTo, localToGlobal, move, moveTo, offsetRect, paintArc, paintCircle, paintOval, paintRect, paintRoundRect, penMode, penNormal, penPat, penSize, pt2Rect, setPt, setRect, subPt, textFace, textFont, textMode, textSize

Quickdraw functions

equalPt, ptlnRect, random

Toolbox procedures

getMouse, getTime, hideAll, note, saveDrawing, setDrawingRect, setTextRect, setTime, showDrawing, showText, sysBeep, writeDraw

Toolbox functions

button, newFileName, oldFileName, tickCount

Extensions

The UCSD extensions to Standard Pascal included above are:

concat, copy, delete, insert, length, longint, maxlongint, pos, seek, string, uses

The Macintosh Pascal extensions to Standard Pascal included above are:

computational, double, extended, filepos, include, omit, ord4, otherwise, Quickdraw1, Quickdraw2, SANE, stringOf, all Quickdraw and Toolbox functions and procedures.

The Macintosh Pascal extensions to Quickdraw (in ROM) included above are:

drawLine, paintCircle, invertCircle.

Symbols	**Purpose**
+	plus
−	minus
*	multiply
div	integer divide
/	real divide
mod	modulus
=	equal to
<>	not equal to
<	less than
>	greater than
<=	less than or equal to; subset of
>=	greater than or equal to; superset of
in	set membership
not	negation
or	disjunction
and	conjunction
@	create pointer
$	hex number specifier
:=	assignment
,	list delimiter
;	statement delimiter
:	variable name/type delimiter
‘	character and string literal delimiter
.	decimal point, program endpoint, and record notation
..	subrange notation

Symbols	Purpose
^	buffer variable notation
(	parameter list or nested expression startpoint
)	parameter list or nested expression endpoint
[	subscript list or set expression startpoint
]	subscript list or set expression endpoint
{	comment startpoint
}	comment endpoint
(.	alternative for [
.)	alternative for]
(*	alternative for {
*)	alternative for }

f The Whiz Kid's Dictionary

abs

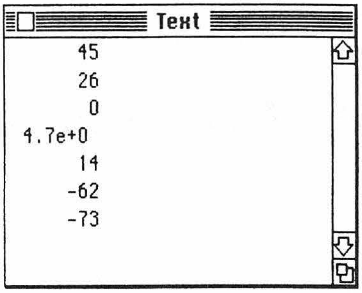
■ **Pascal function**

abs(number) evaluates a single numeric parameter, and returns a type *longint* value if the parameter is of an integer-type, or a type *extended* value if the parameter is a real-type.

The absolute value function has the effect of removing the minus sign from a negative value parameter. Zero or positive parameter values are unaffected. The returned value is always zero or greater.

Inserting a minus sign *before* the abs function forces a returned value of zero or less.

```
program AbsDemo;
 var
  temp : integer;
begin
 writeln(abs(-45));
 writeln(abs(26));
 writeln(abs(0));
 writeln(abs(-4.65));
 temp := -14;
 writeln(abs(temp));
 writeln(-abs(62));
 writeln(-abs(-73))
end.
```



```
      45
      26
       0
 4.7e+0
      14
     -62
     -73
```

addPt

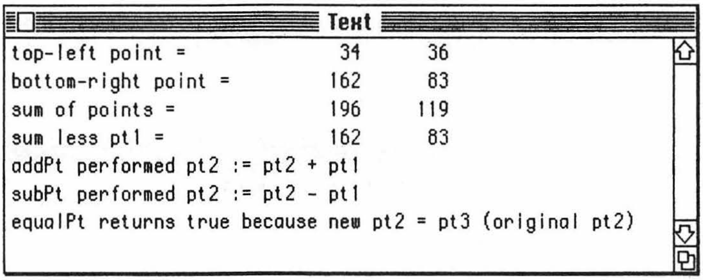
■ Quickdraw procedure

addPt(sourcePoint, destinationPoint) changes the coordinates of destinationPoint by adding the value of the coordinates of sourcePoint. The procedure returns with a new value of destinationPoint.

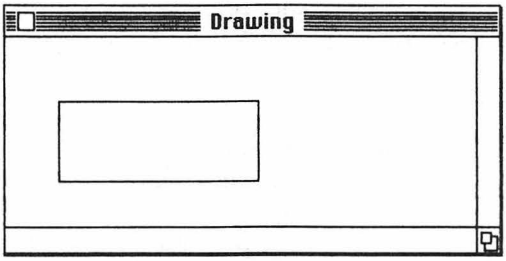
Both parameters of **addPt** are of the Quickdraw type *point.* Information on type *point* can be found under **setPt.**

```
program AddPtDemo;
 var
  x, y : integer;
  pt1, pt2, pt3 : point;
  r : rect;
begin
 repeat
 until button;
 getMouse(x, y);
 setPt(pt1, x, y);    {pt1 is a record variable of two integers such that}
 repeat               {pt1.h := x and pt1.v := y}
  getMouse(x, y);
  setPt(pt2, x, y);  {pt2 is a record variable of integers pt2.h and pt2.v}
  setRect(r, pt1.h, pt1.v, pt2.h, pt2.v);
  frameRect(r);
  eraseRect(r)
 until not button;
 frameRect(r);
 writeln('top-left point =        ', pt1.h, pt1.v);
 writeln('bottom-right point =    ', pt2.h, pt2.v);
 pt3 := pt2;
 addPt(pt1, pt2);
 writeln('sum of points =         ', pt2.h, pt2.v);
 subPt(pt1, pt2);
 writeln('sum less pt1 =          ', pt2.h, pt2.v);
```

```
  if equalPt(pt2, pt3) then
   begin
    writeln('addPt performed pt2 := pt2 + pt1');
    writeln('subPt performed pt2 := pt2 - pt1');
    writeln('equalPt returns true because new pt2 = pt3 (original pt2)');
   end
end.
```



```
top-left point =              34      36
bottom-right point =         162      83
sum of points =              196     119
sum less pt1 =               162      83
addPt performed pt2 := pt2 + pt1
subPt performed pt2 := pt2 - pt1
equalPt returns true because new pt2 = pt3 (original pt2)
```



and

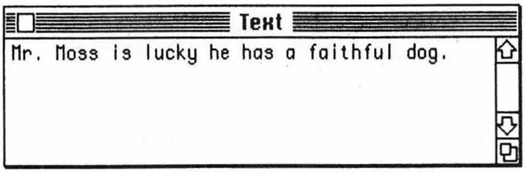
■ Pascal reserved word

and unites two type *boolean* expressions into a single boolean expression. The new expression is true only if both original expressions are true. Otherwise, the new expression is false.

Expressions using **and** require any equalities and inequalities (=, <, >, =<, =>) to be parenthesized.

and has greater precedence than the boolean operator **or**, and less than **not**.

```
program AndDemo;
 var
  baldMenAreMoreVirile, womenFlockToBaldMen : boolean;
begin
 baldMenAreMoreVirile := (1 + 1 = 2) and (2 + 2 = 4);
                                          {true and true = true}
 womenFlockToBaldMen := (1 + 1 = 3) and (2 + 2 = 5);
                                          {false and false = false}
 if (baldMenAreMoreVirile) and (womenFlockToBaldMen) then
                                          {true and false = false}
  writeln('Mr. Moss is a lucky man.')
 else
  writeln('Mr. Moss is lucky he has a faithful dog.')
end.
```



```
Mr. Moss is lucky he has a faithful dog.
```

arctan

■ **Pascal function**

arctan(expression) evaluates a single *real* or *integer* tangent value of an angle, and returns the value of the angle expressed in radians.

array

■ **Pascal reserved word**

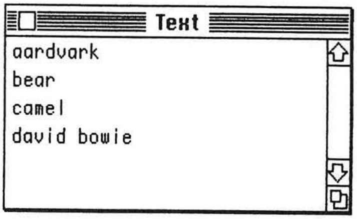
An **array** is a series of variables, called elements, identified by a common name, and whose elements are distinguished by a numeric subscript. The structure of an **array** resembles a series of mailboxes on a Post Office wall. Just as each mailbox is of the same type, so is each element of the array. Likewise, just as the contents of each mailbox belong to different people, the contents of each element are independent and individually accessible.

The array type can be one dimensional like a row of mailboxes, two dimensional like a wall of mailboxes, three dimensional like a room full of mailboxes, or have more than three dimensions. The additional dimensions are created by declaring its boundaries in brackets.

The two dots {..} separating the array bounds mean "through and including." A comma separates the bounds for arrays of more than one dimension.

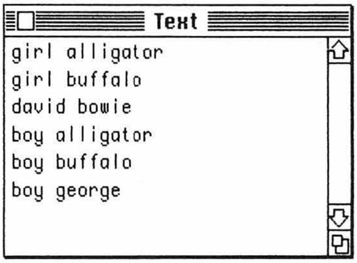
The array boundaries can be of type *integer*, *char*, or *boolean*. Road maps use types *char* and *integer*, like D-2 and K-8, in the same way that arrays use different types of boundaries.

```
program ArrayDemo1;
 var
  ark : array[1..4] of string;
  creatures : integer;
begin
 ark[1] := 'aardvark';
 ark[2] := 'bear';
 ark[3] := 'camel';
 ark[4] := 'david bowie';
 for creatures := 1 to 4 do
  writeln(ark[creatures])
end.
```



```
aardvark
bear
camel
david bowie
```

```
program ArrayDemo2;
 var
  ark : array[1..3, 1..2] of string[20];
  creatures : integer;
begin
 ark[1, 1] := 'girl alligator';
 ark[1, 2] := 'boy alligator';
 ark[2, 1] := 'girl buffalo';
 ark[2, 2] := 'boy buffalo';
 ark[3, 1] := 'david bowie';
 ark[3, 2] := 'boy george';
 for creatures := 1 to 3 do
  writeln(ark[creatures, 1]);
 for creatures := 1 to 3 do
  writeln(ark[creatures, 2])
end.
```



```
girl alligator
girl buffalo
david bowie
boy alligator
boy buffalo
boy george
```

begin

■ Pascal reserved word

begin, in conjunction with the reserved word **end**, serves as a bracket to hold together sections of a Pascal program. Procedures, functions, and the main body are individual blocks whose statements require a **begin** and **end**.

Also, **begin** and **end** hold together multiple statements so that preceding control structures such as **for..to..do, if..then..else,** and **while..do** will act upon two or more statements, instead of performing only the single statement following the control structure.

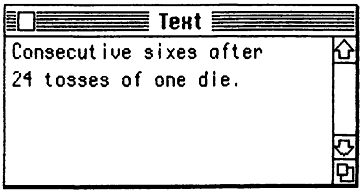
Every **begin** requires its own **end**.

Every **end** belongs to the immediately preceding un**end**ed **begin.**

begin and **end** are not statements, merely punctuation to hold together associated or block statements. Nested blocks can **begin** and **begin** and **begin** inside one another, but always the innermost block must **end** before any outer block.

Macintosh Pascal illustrates the nesting process through its formatting. Each **end** is indented to match the nearest prior un**end**ed **begin.**

```
program BeginDemo;
 var
  oneRoll, countRolls : integer;
  twoSixes : boolean;
begin
 countRolls := 0;
 twoSixes := false;
 repeat
  oneRoll := random mod 6 + 1;
  countRolls := countRolls + 1;
  if oneRoll = 6 then
   begin
    oneRoll := random mod 6 + 1;
    countRolls := countRolls + 1;
    if oneRoll = 6 then
     begin
      twoSixes := true;
      writeln('Consecutive sixes after ', countRolls, ' tosses of one die.');
     end               {end for second if statement}
   end                 {end for first if statement}
 until twoSixes
end.                   {end for program BeginDemo}
```



```
Consecutive sixes after
24 tosses of one die.
```

boolean

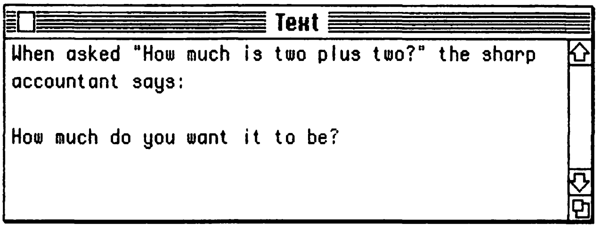
■ Pascal type

A **boolean** type of constant, variable, or function has one of two possible values: true or false. Whenever logic suggests a yes/no, on/off, true/false, or similar choice, use of the boolean type should be considered. Descriptively named booleans add clarity, and often efficiency, to program code.

Equalities and inequalities expressed by mathematical operators (=,<,>,<=,>=,<>) are boolean in nature because expressions using these symbols are either true or false.

Three reserved words, **and, or,** and **not,** are also boolean operators. They are used to construct boolean valued expressions. Expressions using **and, or,** and **not** require equalities and inequalities to be parenthesized.

```
program BooleanDemo;
 var
  goodAccountant : boolean;
begin
 write('When asked "How much is two plus two?"');
 writeln(' the sharp accountant says:');
 writeln;
 goodAccountant := (2 + 2 = 4);
 if goodAccountant then
  writeln('How much do you want it to be?')
 else
  writeln('Four.')
end.
```



```
When asked "How much is two plus two?" the sharp
accountant says:

How much do you want it to be?
```

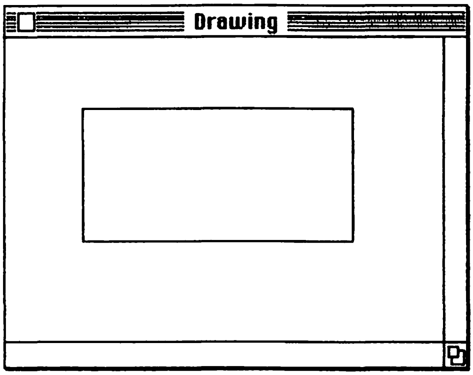
button

■ Toolbox function

button evaluates the status of the Macintosh's mouse button, and returns a boolean value of true if the mouse button is currently being held down. A value of false indicates the mouse button is up at the moment the **button** function is executed.

The **button** function has no parameters.

```
program ButtonDemo;
 var
  x1, y1, x2, y2 : integer;
begin
 repeat
 until button;
 getMouse(x1, y1);
 repeat
  getMouse(x2, y2);
  frameRect(y1, x1, y2, x2);
  eraseRect(y1, x1, y2, x2)
 until not button;
 frameRect(y1, x1, y2, x2);
end.
```



case..of

■ Pascal reserved word

Use of the **case..of** statement closely resembles use of the **if..then** statement.

The **if..then** statement reads a true/false expression, then selects between two choices of action. The **case..of** statement reads any ordered expression, then selects between two or more choices of action from a constant list of options.

The case statement uses the form:

```
case expression of
  constant1:
    statement;
  constant2:
    statement;
  otherwise
    statement
end;
```

The optional use of the reserved word **otherwise** after the last label-constant will perform the indicated action for all values of the expression not **otherwise** listed. If the constant list has no value matching the value of the case expression, and no **otherwise** action is included, an error will halt the program.

The case structure must be concluded with **end.**

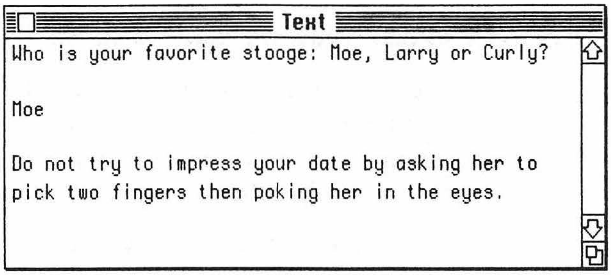
The expression between **case** and **of** acts as the constant selector and must be of the same type as the list of constants. The expression and constants can be of type *integer, char, boolean* or a subrange of a self-defined type.

More than one constant can be used to perform the same action by separating each constant in the constant list with a comma.

The action statements—equivilent to the statements following **then** in an **if..then** statement—can consist of one or, using **begin** and **end**, more than one statement. The choice of no action can be implemented by inserting the semicolon without any statement.

```
program CaseDemo;          {Three Stooges Horoscope}
 var
  firstLetter : char;
begin
 writeln('Who is your favorite stooge: Moe, Larry or Curly?');
 writeln;
 readln(firstLetter);
 writeln;
```

```
  case firstLetter of
   'M', 'm' :
    begin
      write('Do not try to impress your date by asking her to');
      writeln(' pick two fingers then poking her in the eyes.');
    end;      {end 'M' option}
   'L', 'l' :
     writeln('Beware of people who rip out chunks of your hair.');
   'C', 'c' :
    begin
      write('Curly was more than a comedic genius.');
      write('  He was a saint.');
      writeln('  Allow the child in you to flourish.')
    end;      {end 'C' option}
   'S', 's' :
     writeln('Shemp fans:  greatness sometimes stands shadowed.');
   otherwise
     writeln('Hey stooge, you didn't type Moe, Larry, Curly or even Shemp.');
  end         {end case statement}
end.          {end program CaseDemo}
```



```
Who is your favorite stooge: Moe, Larry or Curly?

Moe

Do not try to impress your date by asking her to
pick two fingers then poking her in the eyes.
```

char

■ Pascal type

char is a predefined type representing a single character. A **char** type belongs to an ordered set of characters including, though not limited to, the letters A to Z in upper and lower case, the numbers 0 to 9 and all keyboard punctuation marks.

In order to keep better track of characters, Pascal assigns each character an integer number. Thus, the **char** type becomes an ordered set of characters or *ordinal* type.

A function exists to yield the *ordinal* number of each character. The function **ord**(ch) returns an integer value assigned to the character *ch.*

A complementary function exists to yield the character associated with an ordinal number. The function **chr**(int) returns the character assigned to the integer *int.*

Macintosh Pascal supports both upper- and lower-case letters. Each case of a letter has its own ordinal number.

The integer characters 0 through 9 are also represented by ordinal numbers. The ordinal numbers of the integers differ in value from the integer itself. The ordinal value of the character 0 is 48, the ordinal value of 1 is 49, and so on through 57, the ordinal value of 9.

See **chr** for an example of the **char** type.

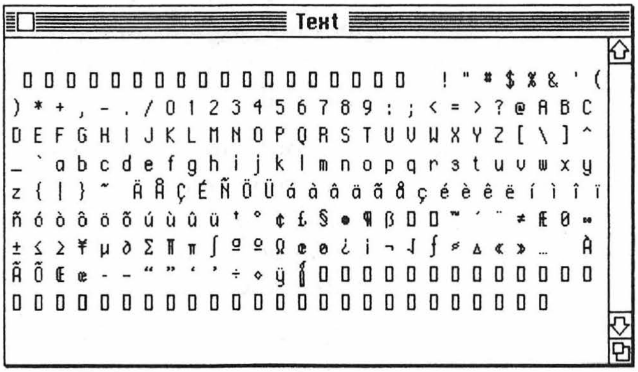
chr

■ Pascal function

The function **chr**(int) returns the character whose ordinal number is the integer *int.*

See **char** for more information on ordinal numbers.

```
program ChrDemo;
 var
  i : integer;
begin
 for i := 1 to 255 do
  begin
   write(' ');
   write(chr(i))
  end
end.
```



```
 □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □   ! " # $ % & ' (
) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C
D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^
_ ` a b c d e f g h i j k l m n o p q r s t u v w x y
z { | } ~  Ä Å Ç É Ñ Ö Ü á à â ä ã å ç é è ê ë í ì î ï
ñ ó ò ô ö õ ú ù û ü † ° ¢ £ § • ¶ ß □ □ ™ ´ ¨ ≠ Æ Ø ∞
± ≤ ≥ ¥ µ ∂ Σ ∏ π ∫ ª º Ω æ ø ¿ ¡ ¬ √ ƒ ≈ ∆ « » …   À
Ã Õ Œ œ - – “ ” ‘ ’ ÷ ◊ ÿ ∫ □ □ □ □ □ □ □ □ □ □ □ □ □
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
```

close

■ **Pascal procedure**

The statement **close**(fileName) terminates the connection between the file organizer *fileName* and the associated external device. No further input or output to the file can occur until the file is opened again.

The **close** procedure serves to protect a file's data from accidental intrusion. Even though MacPascal closes some files automatically, the connection between the file organizer and its external device should be terminated when the file is no longer needed.

If a program ends with any file still open, MacPascal will automatically close it. Likewise, if a procedure or function is exited with a local file organizer still open, the file is automatically closed.

See **rewrite, reset,** or **open** for more information and examples of the **close** procedure.

computational

■ **Macintosh type**

computational is one of four predefined real-types for representing numbers in floating-point notation. The others are *real, double,* and *extended.* The name *real* is used both as a Macintosh type and as a category of the four floating-point types.

Unlike integer-types, real-type numbers can express values with a fractional part, and allow for numbers to be displayed using decimal points.

The purpose of having more than one real-type as options is to provide the range and precision necessary for a particular program application without being wasteful of computer memory space. The higher the range and precision, the more memory space must be allocated.

The type **computational** is a special real-type in that only integral values may be represented. This provides for precise, fixed-point decimal notation as is required for financial applications. A decimal point can be implied to the left of the second to last digit in order to represent dollars and cents.

The procedures **write** and **writeln** allow formatting within their parameter lists to output a **computational** value with the insertion of a decimal point. The format is:

writeln(realValue : fieldWidthInteger : decimalPlacesInteger);

The range of type **computational** is -9.2×10^{18} to 9.2×10^{18}.

For arithmetic operations, all real-type values are converted to type *extended,* and the results are also type *extended.* When a **computational**-type value is required, the *extended* type can be used provided that the value, when rounded to an integral value, falls within the range allowed by **computational**.

See **real** for more information and an example.

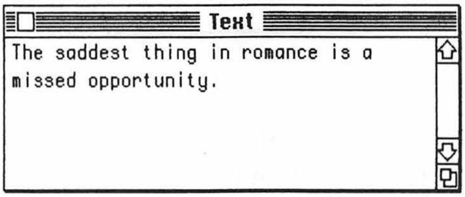
concat

■ **UCSD Pascal function**

concat(string1, string2) concatenates (links) its string parameters, and returns a single string composed in the order of the parameter strings.

More than two literals or variables of type *string* can be joined, though the result string cannot have more than 255 characters. A literal is a string-type set of characters enclosed by single quotation marks.

```
program ConcatDemo;
 var
   s1, s2, s3 : string[60];
begin
 s1 := 'is a missed opportunity.';
 s2 := 'The saddest thing in';
 s3 := concat(s2, ' romance ', s1);
 writeln(s3)
end.
```



```
The saddest thing in romance is a
missed opportunity.
```

const

■ **Pascal reserved word**

const (short for "constant") is the heading for a definition section within a program. The definitions in a **const** section identify data items whose values cannot be changed under program control.

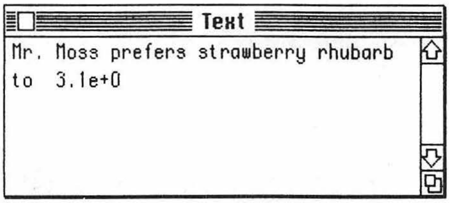
Constants give identifying names to individual data.

Constants can be of any simple type: integer-type, real-type, *char, string,* or *boolean.* The type of a constant does not need to be declared because the constant's value indicates its type.

Constants of an enumerated type cannot be defined at the same level as the type definition. Structured constants such as arrays and records are not allowed.

The **const** section belongs before **type** and **var** sections. **const** definitions can be included globally for use anywhere in a program, or locally within a procedure or function block.

```
program ConstDemo;
 const
  pi = 3.1416;
  pie = 'strawberry rhubarb';
  pieFaced = true;
begin
 if pieFaced then
  writeln('Mr. Moss prefers ', pie, ' to ', pi)
end.
```



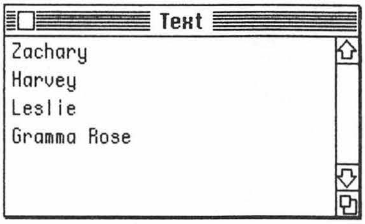
```
Mr. Moss prefers strawberry rhubarb
to  3.1e+0
```

copy

■ **UCSD Pascal function**

copy(string,positionInteger,lengthInteger) evaluates a string and two integers parameters, and returns a string that copies a subset of characters from the string parameter. The subset begins at the positionIntegerth character and continues for lengthInteger characters.

```
program CopyDemo;
 var
  s1, s2, s3, s4, s5, s6, s7, s8 : string[50];
begin
 s1 := 'Mildred, Peter, Zachary, Leslie, Steven, Neil';
 s2 := 'Annette, Jay, Arlene, Harvey';
 s3 := 'Leslie, Matthew, Benjamin';
 s4 := 'Rose, Louie, Dora, Charlie';
 s5 := copy(s1, 17, 7);
 s6 := copy(s2, pos('Harvey', s2), 6);
 s7 := copy(s3, 1, length('Leslie'));
 s8 := copy(s4, 1, 4);
 writeln(s5);
 writeln(s6);
 writeln(s7);
 writeln('Gramma ', s8)
end.
```



```
Zachary
Harvey
Leslie
Gramma Rose
```

cos

Pascal function

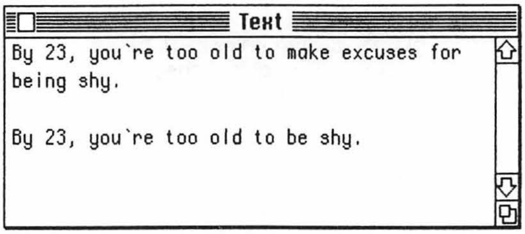
cos(expression) evaluates a single *real* or *integer* angle parameter expressed in radians, and returns an extended *real* value equal to the angle's trigonometric cosine.

delete

UCSD Pascal procedure

delete(string,positionInteger,lengthInteger) changes the string value of the first parameter by removing characters. The characters to be deleted begin at the positionIntegerth character and continue for lengthInteger characters.

```
program DeleteDemo;
 var
  s1, s2 : string[75];
begin
 s1 := 'By 23, you're too old to make excuses for being shy.';
 writeln(s1);
 writeln;
 s2 := 'make excuses for ';
 delete(s1, pos('make', s1), length(s2));
 delete(s1, pos('ing', s1), 3);
 writeln(s1)
end.
```



```
By 23, you're too old to make excuses for
being shy.

By 23, you're too old to be shy.
```

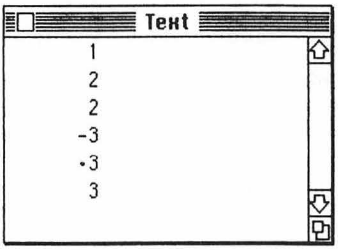
div

■ Pascal reserved word

div works as a mathematical symbol in the same family as +, −, *, /, and **mod. div** performs the operation of *division with an integer result.*

The value of p **div** q is the quotient of p/q rounded to the type *longint* value nearest zero. Both p and q must be an integer-type. If q = 0, an error will result.

```
program DivDemo;
begin
 writeln(9 div 5);
 writeln(10 div 5);
 writeln(11 div 5);
 writeln(-7 div 2);
 writeln(7 div (-2));
 writeln(-7 div (-2))
end.
```



```
      1
      2
      2
     -3
     •3
      3
```

do

■ **Pascal reserved word**

do precedes the action statement(s) in the **for..to..do** loop. See **for** for more information and an example.

double

■ **Macintosh type**

double is one of four predefined real-types for representing numbers in floating-point notation. The others are *real, extended,* and *computational.* The name *real* is used both as a Macintosh type and as a category of the four floating-point types.

Unlike integer-types, real-type numbers can express values with a fractional part, and allow for numbers to be displayed using decimal points.

The purpose of having more than one real-type as options is to provide the range and precision necessary for a particular application without being wasteful of computer memory space. The higher the range and precision, the more memory space must be allocated.

The range of type **double** is 5.0×10^{-324} to 1.7×10^{308}. The precision extends 15 to 16 decimal digits. Type *real* offers less range and precision than **double,** while type *extended* offers more.

For arithmetic operations, all real-type values are converted to type *extended,* and the results are also type *extended.* When a *real* or *double* type value is required, the *extended* type can be used provided that the value falls within the ranges allowed by *real* and *double* respectively.

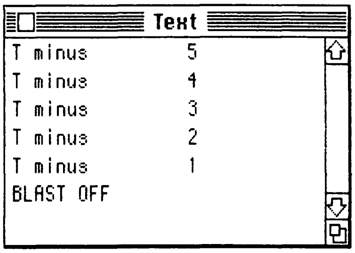
See **real** for more information and an example.

downto

■ **Pascal reserved word**

downto is an alternative to the reserved word **to** in a **for..to..do** loop. Whereas **to** indicates an incrementing loop count, **downto** indicates a decrementing loop count. In both cases, the loop count changes by one.

```
program DowntoDemo;
 var
  count : integer;
begin
 for count := 5 downto 1 do
  writeln('T minus ', count);
 writeln('BLAST OFF')
end.
```



```
T minus        5
T minus        4
T minus        3
T minus        2
T minus        1
BLAST OFF
```

drawLine

■ **MacPascal procedure**

drawLine(horiz1Integer, vert1Integer, horiz2Integer, vert2Integer) draws a line from the coordinate point (horiz1Integer, vert1Integer) to the coordinate point (horiz2Integer, vert2Integer). This MacPascal addition to Quickdraw offers an alternative method of drawing a line to the **lineTo** procedure.

drawString

■ **Quickdraw procedure**

drawString(stringName) places the parameter string into the Drawing window to the right of the Quickdraw pen location. The variable stringName is of the predefined Quickdraw type *str255*.

The type *str255* is defined as **string**[255], allowing from 0 to 255 characters to be assigned.

The current pen location moves to the right of each character as the string is drawn. **drawString** performs no carriage returns, line feeds, or text formatting.

See **writeDraw** for information on a similar procedure that offers limited text formatting.

else

■ **Pascal reserved word**

else is the optional third word of the **if..then** statement. In the format **if..then..else, else** precedes the action statement(s) to be performed when the **if** condition is false.

See **if** for more information and an example.

end

■ **Pascal reserved word**

end, in conjunction with the reserved word **begin**, serves as a bracket to hold together as a unit a program's main body, procedures, functions, multiple statements, record declarations, and case statements. **begin** and **end** show the boundaries of a section of Pascal code that is to be performed as a unit.

The **end;** of a procedure or function requires the statement-separating semicolon. The **end.** of the main body, the final **end** of a program, requires a period.

The **end** of a multiple statement sometimes requires a semicolon. If the instruction following the **end** is a statement, a semicolon is required. If the instruction following the **end** is another **end** or an **else**, the semicolon should be omitted.

The instruction preceding any **end** does not require a semicolon. **end** serves the purpose of the semicolon by separating statements.

Macintosh Pascal automatically formats code, and each **end** is indented to match the nearest prior un**ended** **begin**.

See **begin** for more information and an example.

eof

■ **Pascal function**

eof(fileName) is a boolean function that returns a value of false if the file position of the organizer (fileName) is *at* or *before* the last component of the file, and true if the organizer is *beyond* the last component of the file. **eof** is an abbreviation for *end-of-file.*

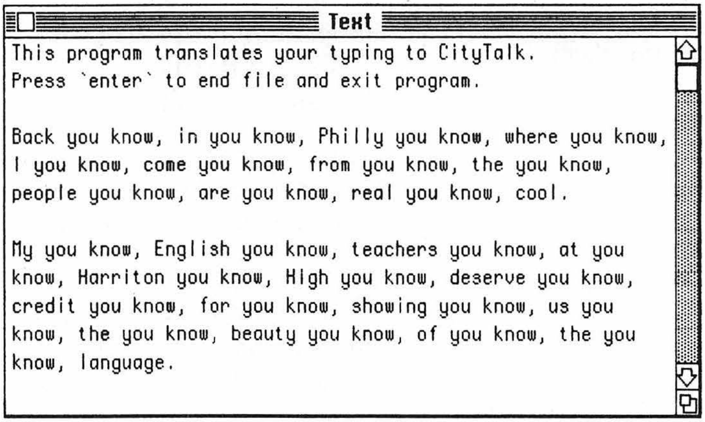
The file organizer parameter is a variable of a file type that directs instructions to a particular file component. Pascal allows only one component to a file to be accessed at a time.

Since Pascal regards all input from devices such as the disk, keyboard, and printer as *files,* the **eof** function signifies where data input has ended. An error will result if a program tries to access a data item beyond the end of a file.

If the parameter of **eof** is omitted, the standard file variable *input* is assumed, and the default input device is the keyboard. Pressing the keyboard's *enter* key causes **eof**(input) to return true.

Whenever **eof**(fileName) returns true, the value of the buffer variable *fileName^* is undefined.

```
program EofDemo;
 var
  oneChar : char;
begin
 writeln('This program translates your typing to CityTalk.');
 writeln('Press `enter` to end file and exit program.');
 writeln;
 while not eof do
  begin
   while not eoln do
    begin
     read(oneChar);
     if oneChar = ' ' then
      write('you know, ');
    end;    {end while not eoln loop}
   readln;    {discard rest of line}
   writeln
  end    {end while not eof loop}
end.    {end EofDemo}
```



```
This program translates your typing to CityTalk.
Press `enter` to end file and exit program.

Back you know, in you know, Philly you know, where you know,
I you know, come you know, from you know, the you know,
people you know, are you know, real you know, cool.

My you know, English you know, teachers you know, at you
know, Harriton you know, High you know, deserve you know,
credit you know, for you know, showing you know, us you
know, the you know, beauty you know, of you know, the you
know, language.
```

eoln

■ Pascal function

eoln(fileName) is a boolean function that returns a value of false if the character position of the organizer (fileName) is *before* an end-of-line character, and true if the organizer is *at* an end-of-line character. **eoln** is an abbreviation for *end-of-line.*

The file organizer *fileName* must be a variable of type *text.* If the parameter is omitted, the standard file *input* is assumed, and the default input device is the keyboard. Pressing the keyboard's *return* key issues an end-of-line character. Since only text-type files are organized with respect to end-of-line characters, it is an error if **eoln** is used with a nontext file.

See **eof** for more information and an example.

equalPt

■ **Quickdraw function**

equalPt(point1Name, point2Name) evaluates its two point parameters, and returns the boolean result true if they are equal, false if they are not. The variables point1Name and point2Name are of the predefined Quickdraw type *point.*

See *addPt* for an example.

eraseArc
eraseOval
eraseRect
eraseRoundRect

■ **Quickdraw procedures**

In all the following procedures, rectName is a variable of type *rect,* which can be created by using the **setRect** procedure. Alternatively, the rectName variable can be substituted with four integers representing the rectangle's boundary coordinates *top, left, bottom,* and *right,* respectively.

All procedures paint the specified shape with the Drawing window's background pattern. Drawing is done in the patCopy mode. The Quickdraw pen pattern and draw-over mode are ignored. The pen location is unaffected. See **penMode, penPat,** and **penSize** for more information on the Quickdraw pen.

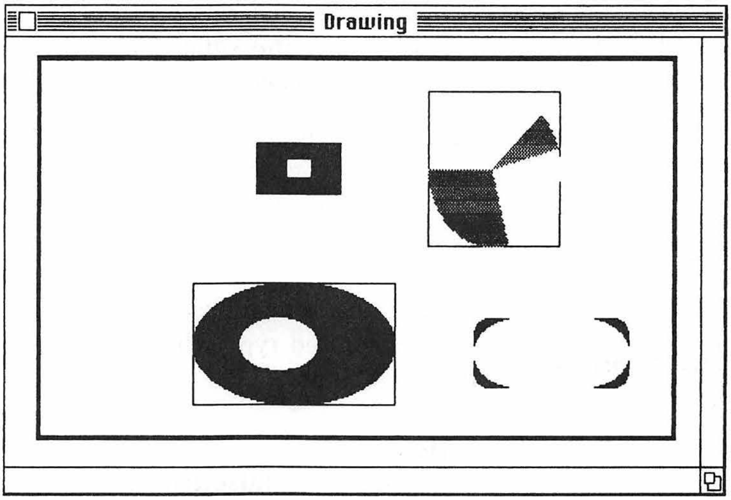
eraseArc(rectName, startAngleInteger, arcAngleInteger) paints in the background pattern a wedge of the oval that fits within the rectangular dimensions set by rectName. The parameter startAngleInteger is a degree value between 0 and 359 that works like the hand of an oval clock: 0 points to 12 o'clock, 90 points to 3 o'clock, 180 points to 6 o'clock, and so on. The parameter arcAngleInteger is a degree value between −359 and 359 that sets the extent of the arc, positive angles extending clockwise, negative angles counterclockwise.

eraseOval(rectName) paints in the background pattern an oval that fits within the rectanglar dimensions set by rectName.

eraseRect(rectName) paints in the background pattern a rectangle within the dimensions set by rectName.

eraseRoundRect(rectName, ovalWidthInteger, ovalHeightInteger) paints in the background pattern a rounded-corner rectangle within the dimensions set by rectName. The curvature of the rounded corners is set by two integers that specify the diameters of an oval shape suggested by the rounded corners.

```
program EraseDemo;
 procedure createDrawingWindow;
  var
   drWindow : rect;
 begin
  hideAll;
  setRect(drWindow, 20, 60, 480, 320);
  setDrawingRect(drWindow);
  showDrawing
 end;
 procedure paintArt;
  var
   i : integer;
   r1, r2, r3, r4, r5 : rect;
 begin
  paintRect(60, 160, 90, 215);
  eraseRect(70, 180, 80, 195);
  setRect(r1, 50, 30, 100, 120);
  setRect(r2, 120, 140, 250, 210);
  setRect(r3, 300, 160, 400, 200);
  setRect(r4, 270, 30, 355, 120);
  setRect(r5, 150, 160, 200, 190);
  paintRect(r1);
  frameRect(r2);
  paintOval(r2);
  eraseOval(r5);
  paintRoundRect(r3, 20, 20);
  eraseRoundRect(r3, 60, 60);
  frameRect(r4);
  penPat(gray);
  paintArc(r4, 45, 225);
  eraseArc(r4, 75, 90);
  penSize(3, 3);
  penPat(black);
  frameRect(10, 20, 230, 430);
  for i := 1 to 5000 do           {time delay}
   ;
  eraseRect(r1);
 end;
begin
 createDrawingWindow;
 paintArt
end.
```



exp

■ **Pascal function**

exp(expression) evaluates a single *real* or *integer* value, and returns an extended *real* value of its exponential. **exp**(x) computes the value of e^x where **e** is the base of the natural logarithms (2.718...).

extended

■ **Macintosh type**

extended is one of four predefined real-types for representing numbers in floating-point notation. The others are *real, double,* and *computational.* The name *real* is used both as a Macintosh type and as a category of the four floating-point types.

Unlike integer-types, real-type numbers can express values with a fractional part, and allow for numbers to be displayed using decimal points.

The purpose of having more than one real-type as options is to provide the range and precision necessary for a particular program application without being wasteful of computer memory space. The higher the range and precision, the more memory space must be allocated.

The range of type **extended** is 1.9×10^{-4951} to 1.1×10^{4932}. The precision extends 19 to 20 decimal digits. Types *real* and *double* economize space by offering less range and precision.

For arithmetic operations, all real-type values are converted to type *extended,* and the results are also type *extended.* When a *real* or *double* type value is required, the *extended* type can be used provided that the value falls within the ranges allowed by *real* and *double* respectively.

See **real** for more information and an example.

false

■ Pascal constant

false is one of two predefined values of type *boolean.* Boolean is an enumerated type whose members are the ordered constants false and **true.**

```
type
  boolean = (false,true);
```

The written output of a boolean expression will be the word **false** or **true.**

See the boolean function **odd** for an example.

file of

■ Pascal reserved words

file of denotes, in part, a Pascal type declaration. The format of this kind of declaration is:

```
fileName : file of componentType;
```

A primary purpose of declaring a file is to communicate information to an external device such as a disk drive or printer. When a file is opened, a connection can be made between the file variable *fileName* and the external device.

A file is composed of a linear sequence of data items or components. The sequence is numbered so that each component has its own numeric position. Only one component of a file can be accessed at a time. To access a particular component of a file, you open the file and reference the numbered position of the component.

This nature of a file demands that a file not only keep track of *what* components are contained in a file, but *where* among the components is the file currently active.

A variable of a file-type is used to organize the access to a file's components. This variable, called the *file organizer,* represents all of a file's components while keeping track of the current file position.

A *buffer* variable—created by appending a caret sign (^ on the 6 key) to the end of the file organizer name—represents the current component of the file organizer. Whereas the organizer is a variable of a file-type, the buffer variable is of the same type as the file's component.

A file must be opened in order to access any component. Three predefined procedures exist for this purpose. **rewrite** opens a file so that components can be *written* to the file sequentially. **reset** opens a file so that components can be *read* from the file sequentially. **open** opens a file so that components can be either *written or read randomly.*

The predefined procedure **close** closes a file, thus protecting its contents from inadvertent activity.

A file must belong to one of two classes: those declared with the **file of** notation, and those declared of type *text.* A text-type file is unique because its data is organized into lines. Data sent to the printer must be contained in a file variable of type *text.*

Files cannot contain other files.

See **eof, eoln, filepos, get, open, put, reset, rewrite, seek,** and **text** for information on other predefined procedures and functions affecting files.

filepos

■ **Macintosh function**

Use **filepos**(fileName) to return a *longint* value that indicates the current position of the file organizer parameter among a file's components.

Components are numbered consecutively, the first component numbered zero and the last component followed by an end-of-file character. A file's current component is accessible through the buffer variable, notated by the file organizer name with a caret sign appended (fileName^).

The **filepos** function returns the file position number of the component currently available through the buffer variable.

See **file** for more information. See **get** for an example.

fillArc fillOval fillRect fillRoundRect

■ **Quickdraw procedures**

In the following procedures, rectName is a variable of type *rect,* which can be created by using the **setRect** procedure. Alternatively, *rectName* can be substituted with four integers representing the rectangle's boundary coordinates *top, left, bottom,* and *right,* respectively.

All procedures fill a specified shape with patternName, a variable of type *pattern.* The five predefined patterns are: *white, black, gray, ltGray,* and *dkgray.* See **penPat** for more information on creating custom patterns.

All procedures fill using the patCopy mode. The current state of the Quickdraw pen's pattern and mode, as well as the background pattern, are ignored. The pen location is unaffected. See **penMode, penPat,** and **penSize** for more information on the Quickdraw pen.

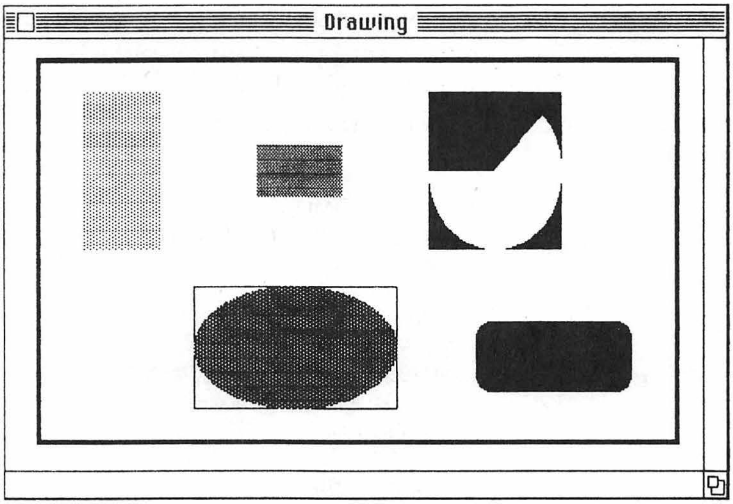
fillArc(rectName, startAngleInteger, arcAngleInteger, patternName) fills a wedge of the oval that fits within the rectangular dimensions set by rectName. The parameter startAngleInteger is a degree value between 0 and 359 that works like the hand of an oval clock: 0 points to 12 o'clock, 90 points to 3 o'clock, 180 points to 6 o'clock, and so on. The parameter arcAngleInteger is a degree value between −359 and 359 that sets the extent of the arc, positive angles extending clockwise, negative angles counterclockwise. The fill pattern is set by the parameter patternName.

fillOval(rectName, patternName) fills an oval that fits within the rectangular dimensions set by rectName. The fill pattern is set by the parameter patternName.

fillRect(rectName, patternName) fills a rectangle within the dimensions set by rectName. The fill pattern is set by the parameter patternName.

fillRoundRect(rectName, ovalWidthInteger, ovalHeightInteger, patternName) fills a rounded-corner rectangle within the dimensions set by rectName. The curvature of the rounded corners is set by two integers that specify the diameters of an oval shape suggested by the rounded corners. The fill pattern is set by the parameter patternName.

```
program FillDemo;
 procedure createDrawingWindow;
  var
   drWindow : rect;
 begin
  hideAll;
  setRect(drWindow, 20, 60, 480, 320);
  setDrawingRect(drWindow);
  showDrawing
 end;
 procedure fillArt;
  var
   r1, r2, r3, r4 : rect;
 begin
  fillRect(60, 160, 90, 215, gray);
  setRect(r1, 50, 30, 100, 120);
  setRect(r2, 120, 140, 250, 210);
  setRect(r3, 300, 160, 400, 200);
  setRect(r4, 270, 30, 355, 120);
  fillRect(r1, ltgray);
  frameRect(r2);
  fillOval(r2, dkgray);
  fillRoundRect(r3, 20, 20, black);
  paintRect(r4);
  fillArc(r4, 45, 225, white);
  penSize(3, 3);
  penPat(black);
  frameRect(10, 20, 230, 430)
 end;
begin
 createDrawingWindow;
 fillArt
end.
```



for..to..do

■ **Pascal reserved word**

for..to..do creates a loop that will perform an action for a set number of repetitions. The **for** loop requires the following format:

for variable := initialExpression **to** finalExpression **do**
  statement

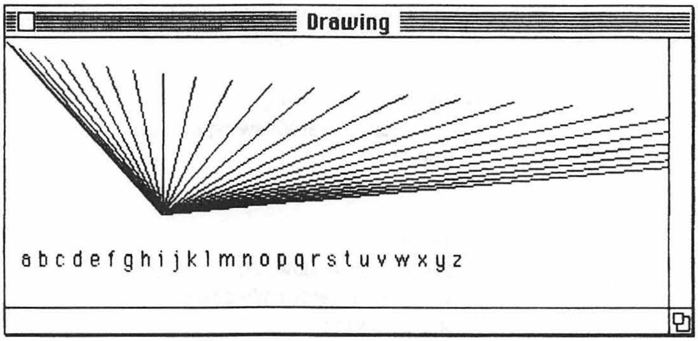
The type of the variable and two expressions must be identical to one another, and be ordered such that the loop can be counted. The **for** variable counts repetitions and cannot be assigned a new value within the loop.

The completion of each repetition causes the **for** variable to be incremented by one. When the value of the first expression has been incremented to the value of the second expression, the statement(s) will be performed for the final repetition.

If the action following **do** consists of more than one statement, the statements of the loop must be bracketed by **begin** and **end.**

The reserved word **to** can be replaced by the reserved word **downto.** Instead of incrementing the **for** variable, **downto** causes the value to decrement by one, and the loop performs its last repetition when the first expression drops to the value of the second.

```
program ForDemo;
 var
  i : integer;
  ch : char;
begin
 for i := 1 to 26 do
  begin
   moveTo(100, 100);
   lineTo(i * i, i + i)
  end;
 moveTo(10, 130);
 for ch := 'a' to 'z' do
  writeDraw(ch, ' ')
end.
```



```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

frameArc
frameOval
frameRect
frameRoundRect

■ **Quickdraw procedures**

In the following procedures, rectName is a variable of type *rect,* which can be created by using the **setRect** procedure. Alternatively, rectName can be substituted with four integers representing the rectangle's boundary coordinates *top, left, bottom,* and *right,* respectively.

The frame or outline drawn by each procedure uses Quickdraw's currently selected pen pattern, size, and mode. Pen location is unaffected. See **penMode, penPat,** and **penSize** for more information on the Quickdraw pen.

frameArc(rectName, startAngleInteger, arcAngleInteger) draws an arc of the oval that fits within the rectangular dimensions set by rectName. The parameter startAngleInteger is a degree value between 0 and 359 that works like the hand of an oval clock: 0 points to 12 o'clock, 90 points to 3 o'clock, 180 points to

6 o'clock, and so on. The parameter arcAngleInteger is a degree value between −359 and 359 that sets the extent of the arc, positive angles extending clockwise, negative angles counterclockwise.

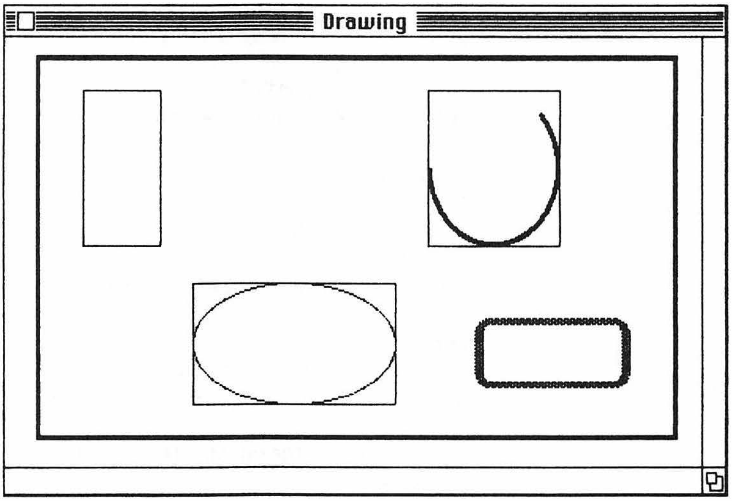
frameOval(rectName) draws an oval outline that fits within the rectanglar dimensions set by rectName.

frameRect(rectName) draws a rectangular outline within the dimensions set by rectName.

frameRoundRect(rectName, ovalWidthInteger, ovalHeightInteger) draws a rounded-corner rectangular outline within the dimensions set by rectName. The curvature of the rounded corners is set by two integers that specify the diameters of an oval shape suggested by the rounded corners.

```
program FrameDemo;
 procedure createDrawingWindow;
  var
   drWindow : rect;
 begin
  hideAll;
  setRect(drWindow, 20, 60, 480, 320);
  setDrawingRect(drWindow);
  showDrawing
 end;
 procedure frameArt;
  var
   r1, r2, r3, r4 : rect;
 begin
  setRect(r1, 50, 30, 100, 120);
  setRect(r2, 120, 140, 250, 210);
  setRect(r3, 300, 160, 400, 200);
  setRect(r4, 270, 30, 355, 120);
  frameRect(r1);
  frameRect(r2);
  frameOval(r2);
  penSize(6, 4);
  penPat(dkgray);
  frameRoundRect(r3, 20, 20);
  penNormal;
  frameRect(r4);
  penSize(3, 3);
  frameArc(r4, 45, 225);
  frameRect(10, 20, 230, 430)
 end;
```

```
begin
 createDrawingWindow;
 frameArt
end.
```



function

■ **Pascal reserved word**

A **function** takes a value, acts on that value, then returns a value. Functions are created by giving the function a name, declaring its type and, between **begin** and **end;**, inserting statements that will assign the function's name a returning value.

The format of a function is as follows:

```
function name (parameters—if any) : type;
  begin
    statements
  end;
```

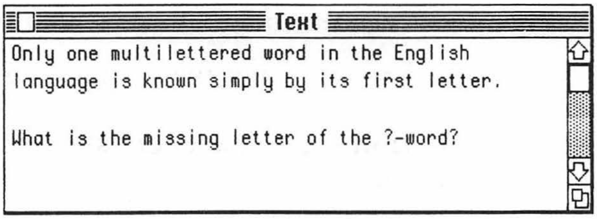
The function's type must be stated after the colon, and refers to the *result type,* the type of the value that the function returns to the program.

Functions always return a single value assigned by the function's statements. This differs from procedures, which execute statements without being required to return anything.

Like procedures, functions are executed when called by name. Functions can use parameters, as well as local **const** and **var** declarations, in a similar way as do procedures. See **procedure** for more information on these features.

Pascal's predefined functions require only that the function be called using an appropriate result type and parameters.

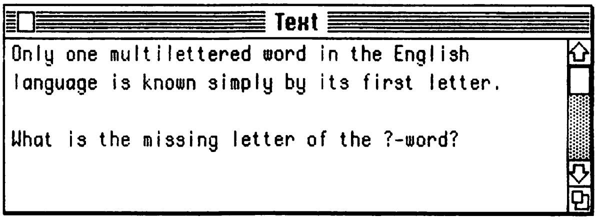
```
program FunctionDemo;
 function rightAnswer : boolean;
  var
   ch : char;
 begin
  rightAnswer := false;
  read(ch);
  rightAnswer := (ch = 'f') or (ch = 'F')
 end;
begin
 write('Only one multilettered word in the English language ');
 writeln('is known simply by its first letter.');
 writeln;
 writeln('What is the missing letter of the ?-word?');
 writeln;
 if rightAnswer then
  writeln('  You're right.  Shame on you. ')
 else
  writeln('  Sorry, you mucked-up, and no, it's not the M-word.')
end.
```



```
Only one multilettered word in the English
language is known simply by its first letter.

What is the missing letter of the ?-word?
```

```
program FunctionDemo2;
 var
  answer : char;
 function getAnswer : char;
  var
   ch : char;
 begin
  read(ch);
  getAnswer := ch
 end;
```

```
begin
  write('Only one multilettered word in the English language ');
  writeln('is known simply by its first letter.');
  writeln;
  writeln('What is the missing letter of the ?-word?');
  writeln;
  answer := getAnswer;
  if (answer = 'f') or (answer = 'F') then
    writeln('  You're right.  Shame on you. ')
  else
    writeln('  Sorry, you mucked-up, and no, it's not the M-word.')
end.
```



```
Only one multilettered word in the English
language is known simply by its first letter.

What is the missing letter of the ?-word?
```

get

■ **Pascal procedure**

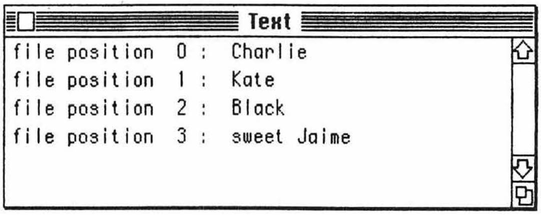
get(fileName) advances the file organizer by one position to the next file component. Then the buffer variable *fileName*^ is assigned the value of the current component. If the new position of the file organizer is beyond the last component of the file, then the value of *fileName*^ becomes undefined.

Subsequent calls to the **get** procedure will access the next sequential component of the file. The **get** procedure advances itself through a file, one component at a time.

To randomly access a component, that is, without first getting all preceding components, use the procedure **seek. seek** uses a numeric parameter to directly position the file organizer anywhere along the numbered file, whereas **get** advances the file organizer by a single component position.

get advances the file organizer for the purpose of accessing a file data. In order to insert a file component, use the file command **put.**

```
program GetDemo;     {GetDemo's file was created in PutDemo}
 var
  giveName, fileName : string[50];
  Jaime : string[20];
  friends : file of string[20];
begin
 giveName := 'type in  new file name';
 fileName := oldFileName(giveName);
 reset(friends, fileName);  {reset performs the first get}
 while not eof(friends) do
  begin
   write('file position', filepos(friends) : 3, ' : ');  {file window}
   writeln(friends^);    {write contents of current window}
   get(friends)          {advance window and get new contents}
  end;
 close(friends)
end.
```



```
file position  0 :  Charlie
file position  1 :  Kate
file position  2 :  Black
file position  3 :  sweet Jaime
```

getMouse

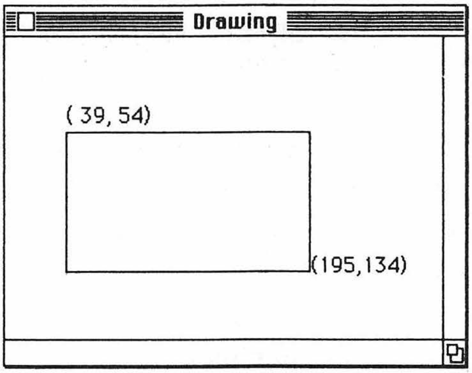
■ Toolbox procedure

getMouse(horizInteger,vertInteger) returns two integers corresponding the current coordinates of the mouse cursor. The coordinates (0,0) plot the upper-left corner of the Drawing window.

If the mouse is left of the Drawing window when **getMouse** is called, the horizontal integer—the first parameter—will return as negative. If the mouse is above the Drawing window, the vertical coordinate will return as negative.

The two **var** parameters horizInteger and vertInteger can be given the field names of the record type *point* (pt.h,pt.v). In this way, a variable of type *point* can be defined. Alternatively, the **setPt** procedure can be used to create a variable of type *point.*

```
program getMouseDemo;
 var
   x1, y1, x2, y2 : integer;
begin
 repeat
 until button;
 getMouse(x1, y1);
 moveTo(x1, y1 - 5);
 writeDraw('(', x1 : 3, ',', y1 : 3, ')');
 repeat
   getMouse(x2, y2);
   frameRect(y1, x1, y2, x2);
   eraseRect(y1, x1, y2, x2)
 until not button;
 frameRect(y1, x1, y2, x2);
 moveTo(x2, y2);
 writeDraw('(', x2 : 3, ',', y2 : 3, ')')
end.
```



```
( 39, 54)
(195,134)
```

getPen

■ **Quickdraw procedure**

getPen(varPointName) returns with the variable varPointName assigned to the current pen location expressed in the local coordinates of the Drawing window. The parameter varPointName is of type *point.*

See **line** for an example.

getTime

■ **Toolbox procedure**

getTime(recordName) returns the date and time information from the Macintosh system clock. The **var** parameter of **getTime** is of a *record* type. This *record* type is defined as:

```
DateTimeRec = record
     year,
     month,
     day,
     hour,
     minute,
     second,
     dayOfWeek : integer
   end;
```

The field *year* must be greater than or equal to 1904. *month* is a number from 1 to 12, corresponding to January through December, respectively. *day* is a number from 1 to 31, representing the day of the month.

The field *hour* is the number of hours since midnight. The first hour of the day is 0^{th} hour. The P.M. hours are represented by the numbers 12 through 23. *minute* and *second* are numbers from 0 to 59.

dayOfWeek is a number from 1 to 7, corresponding to Sunday through Saturday, respectively.

The Toolbox procedure **setTime** can be used to set the date and time of the Macintosh system clock.

See Chapters 27 and 28 in Part 2 for more information and examples of **getTime.**

globalToLocal

■ **Quickdraw procedure**

globalToLocal(pointName) converts a point expressed in global coordinates—such as the Macintosh screen—to the local coordinates of the Drawing window. The variable pointName of type *point* is created using the **setPt** procedure.

A complementary procedure, **localToGlobal** performs the opposite conversion.

See Chapter 20 in Part 2 for more information and an example.

goto

■ **Pascal reserved word**

The **goto**(labelnumber) statement searches for the place in the program code where labelnumber has been inserted, then continues program execution from point of the label.

Label numbers must be defined using the reserved word **label** followed by the label numbers to be used. ***label** 1,2,3,4;* defines four labels. The label definition part must immediately follow the program, procedure, or function heading.

To place a label in the program code, insert the label number with a colon attached. **goto**(3) will go to the label marked 3:.

The use of **goto** disrupts program readability. It can and should be avoided, hence no example.

hideAll

■ **Toolbox procedure**

Use **hideAll** to clear the Macintosh screen of all windows. Any window can be redrawn by selecting the window from the Window menu. From within a program, the Text and Drawing windows can be drawn by the **showText** and **showDrawing** procedures, respectively.

The **hideAll** procedure cannot selectively hide windows. All windows are hidden and, except for the menu bar, the screen displays the background pattern.

See **setTextRect** and **setDrawingRect** for drawing windows to specified dimensions from within a program, and an example of **hideAll**.

if..then

■ **Pascal reserved word**

Use an **if..then** statement when a decision between two possible courses of action is necessary. The decision must be based on whether a condition is either true or false.

The format of the **if..then** statement is as follows:

```
if booleanExpression then
   statement
```

The expression must be a *boolean* type, that is, true or false. An expression evaluated as true would cause the action after **then** to be performed. An expression evaluated as false would ignore the action after **then.**

An action of more than one statement needs to be bracketed by **begin** and **end.**

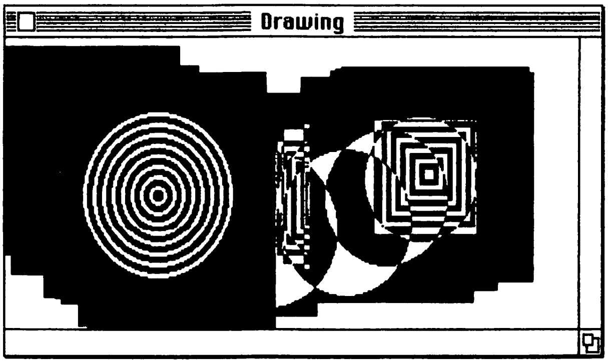
An optional format of the **if..then** statement includes the reserved word **else.**

```
if booleanExpression then
  statement
else
  statement
```

The **else** option permits a defined alternative course of action should the expression be evaluated as false.

When an **if..then..else** statement is used within the action part of another **if..then..else** statement, the word **else** will be associated with the nearest preceding **if** that is not already paired with an **else.**

```
program IfDemo;
  var
    i, x, y : integer;
begin
  i := 0;
  repeat
    getMouse(x, y);
    if odd(x) then
      invertRect(y - i, x - i, y + i, x + i)
    else
      invertOval(y - i, x - i, y + i, x + i);
    i := i + 3;
    if i = 60 then
      begin
        paintRect(y - i, x - i, y + i, x + i);
        i := 0
      end
  until button
end.
```



in

■ Pascal reserved word

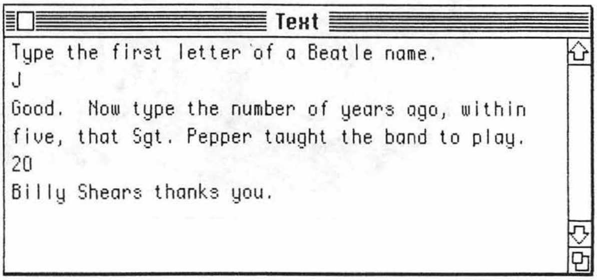
in works like the mathematical symbols (=, <, >), except that instead of testing whether an expression's value is *equal to, less than,* or *greater than* another value, **in** tests whether the value is contained *in* a listed set of ordered values. The format is as follows:

expression **in** [set of ordered values]

The result is a boolean expression; true if the value is a member of the specified set, false if the value is not.

A set of ordered values is created by enclosing the member values with square brackets. Each member must be separated by a comma, though an inclusive list of integer and char values can be notated using two periods (..) between the first and last members.

```
program InDemo;
 var
  mophead : char;
  yearsAgo : integer;
begin
 writeln('Type the first letter of a Beatle name.');
 repeat
  readln(mophead)
 until mophead in ['J', 'j', 'P', 'p', 'G', 'g', 'R', 'r'];
 write('Good. Now type the number of years ago, within five,');
 writeln(' that Sgt. Pepper taught the band to play.');
 readln(yearsAgo);
 if yearsAgo in [15..25] then
  writeln('Billy Shears thanks you.')
 else
  writeln('It was twenty years ago today.')
end.
```



```
Type the first letter of a Beatle name.
J
Good.  Now type the number of years ago, within
five, that Sgt. Pepper taught the band to play.
20
Billy Shears thanks you.
```

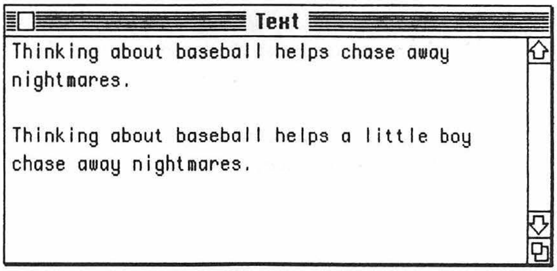
include

■ Macintosh function

include(addString, baseString, positonInteger) inserts the addString parameter into the baseString parameter beginning at the positionIntegerth character of the baseString, and returns the resulting string.

The **include** function differs from the **insert** procedure because **include** returns a result string without affecting baseString, whereas **insert** changes the value of, and returns as a **var** parameter, baseString.

```
program IncludeDemo;
 var
  s1, s2, s3 : string[80];
begin
 s1 := 'Thinking about baseball helps chase away nightmares.';
 s2 := 'a little boy ';
 s3 := include(s2, s1, pos('chase', s1));
 writeln(s1);
 writeln;
 writeln(s3)
end.
```



```
Thinking about baseball helps chase away
nightmares.

Thinking about baseball helps a little boy
chase away nightmares.
```

input

■ **Pascal file**

input is a predeclared file allowing read operations to the Macintosh keyboard. MacPascal does not require **input**, or the predeclared write file **output**, to be explicitly stated. Other versions of Pascal require **input** and **output** to be stated as program parameters.

Program parameters are never required in MacPascal. **input** and **output** are the only program parameters allowed.

The format is:

program programName(input, output);

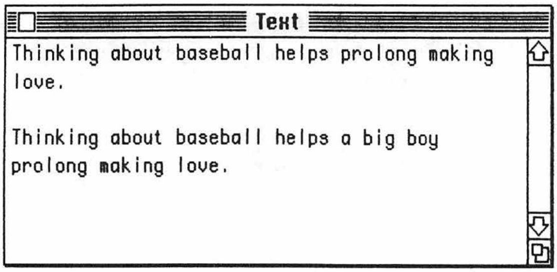
insert

■ **UCSD Pascal procedure**

insert(addString, baseString, positionInteger) inserts the addString parameter into the baseString parameter beginning at the positionInteger*th* character of the baseString. The **var** parameter baseString returns from the procedure with a new value.

The **insert** procedure differs from the **include** function because **insert** changes the value of, and returns as a **var** parameter, baseString, whereas **include** returns a result string without affecting baseString.

```
program InsertDemo;
 var
  s1, s2 : string[80];
begin
 s1 := 'Thinking about baseball helps prolong making love.';
 s2 := 'a big boy ';
 writeln(s1);
 writeln;
 insert(s2, s1, pos('prolong', s1));
 writeln(s1)
end.
```



```
Thinking about baseball helps prolong making
love.

Thinking about baseball helps a big boy
prolong making love.
```

insetRect

■ **Quickdraw procedure**

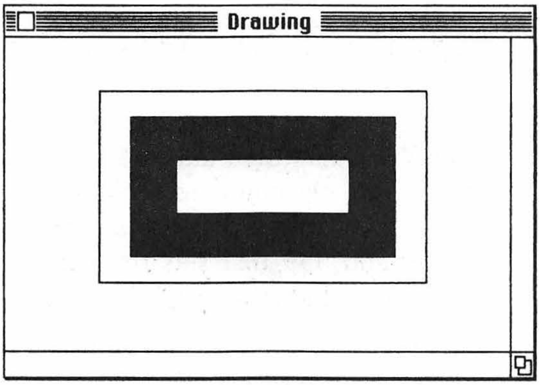
insetRect(rectName, horizInteger, vertInteger) shrinks or expands the variable rectName. The variable parameter rectName is of the predefined Quickdraw type *rect,* which can be created with the **setRect** procedure.

The left and right sides of rectName are shrunk toward one another by positive values of horizInteger, and expanded away from one another by negative values. The horizontal coordinates of both sides are changed by the amount of horizInteger.

The top and bottom sides of rectName are shrunk toward one another by positive values of vertInteger, and expanded away from one another by negative values. The vertical coordinates of both sides are changed by the amount of vertInteger.

The **insetRect** procedure does not affect the centering of the rectangle unless the rectangle is shrunk to where its width or height is less than one, whereupon it becomes an empty rectangle (0, 0, 0, 0).

```
program InsetRectDemo;
 var
  r : rect;
begin
 setRect(r, 80, 45, 250, 125);
 paintRect(r);
 insetRect(r, 30, 25);
 invertRect(r);
 insetRect(r, -50, -40);
 frameRect(r)
end.
```



integer

■ Pascal type

integer is a predefined type whose members include all positive and negative whole numbers between −32,767 and 32,767 inclusive. Zero is also an integer.

Variables declared to be of type **integer** can take any value in the range stated above. Whole numbers outside the range of **integer** and numbers requiring floating-point notation can be declared using types **longint** and **real**, respectively.

The following predefined Pascal functions will produce an integer result: **abs, sqr, trunc, round, succ,** and **pred.** See each classification for information and an example.

Examples of type **integer** can be found throughout Part 3.

invertArc
invertOval
invertRect
invertRoundRect

■ Quickdraw procedures

In the following procedures, rectName is a variable of type *rect,* which can be created by using the **setRect** procedure. Alternatively, rectName can be substituted with four integers representing the rectangle's boundary coordinates *top, left, bottom,* and *right,* respectively.

All procedures invert the dots within the specified shape. Every black dot becomes white and every white dot becomes black. The Quickdraw pen pattern and draw-over mode, as well as the background pattern, are ignored. The pen location is unaffected. See **penMode, penPat,** and **penSize** for more information on the Quickdraw pen.

invertArc(rectName, startAngleInteger, arcAngleInteger) inverts the dots enclosed in a wedge. The wedge is specified by the oval that fits within the rectangular dimensions set by rectName. The parameter startAngleInteger is a degree value between 0 and 359 that works like the hand of an oval clock: 0 points to 12 o'clock, 90 points to 3 o'clock, 180 points to 6 o'clock, and so on. The parameter arcAngleInteger is a degree value between −359 and 359 that sets the extent of the arc, positive angles extending clockwise, negative angles counterclockwise.

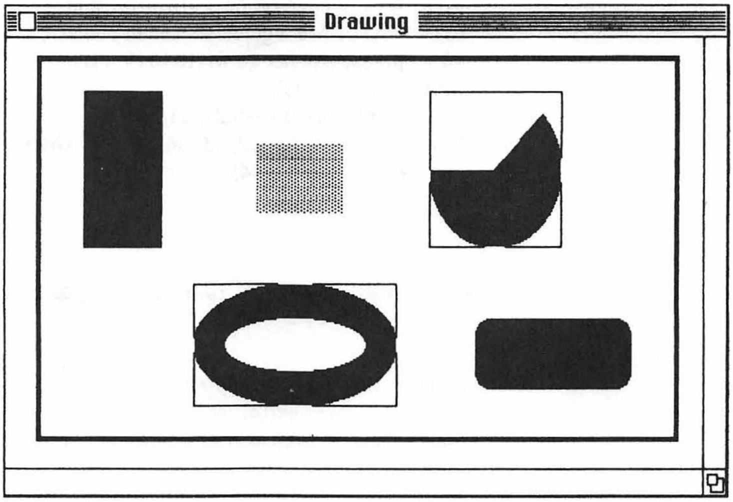
invertOval(rectName) inverts the dots enclosed in an oval that fits within the rectangular dimensions set by rectName.

invertRect(rectName) inverts the dots enclosed in a rectangle whose dimensions are set by rectName.

invertRoundRect(rectName, ovalWidthInteger, ovalHeightInteger) inverts the dots enclosed in a rounded-corner rectangle whose dimensions are set by rectName. The curvature of the rounded corners is set by two integers that specify the diameters of an oval shape suggested by the rounded corners.

```
program InvertDemo;
 var
  i : integer;
 procedure createDrawingWindow;
  var
   drWindow : rect;
 begin
  hideAll;
  setRect(drWindow, 20, 60, 480, 320);
  setDrawingRect(drWindow);
  showDrawing
 end;
 procedure invertArt;
  var
   r1, r2, r3, r4 : rect;
 begin
  invertRect(60, 160, 100, 215);
  setRect(r1, 50, 30, 100, 120);
  setRect(r2, 120, 140, 250, 210);
  setRect(r3, 300, 160, 400, 200);
  setRect(r4, 270, 30, 355, 120);
  invertRect(r1);
  penSize(1, 1);   {default pensize}
```

```
   frameRect(r2);
   invertOval(r2);
   invertOval(160, 140, 190, 230);
   invertRoundRect(r3, 20, 20);
   frameRect(r4);
   invertArc(r4, 45, 225);
   penSize(3, 3);
   frameRect(10, 20, 230, 430)
  end;
begin
 createDrawingWindow;
 penPat(dkGray);
 paintRect(60, 160, 100, 215);
 penPat(black);
 for i := 1 to 49 do
  invertArt
end.
```



invertCircle

■ **MacPascal procedure**

invertCircle(horizInteger, vertInteger, radiusInteger) inverts the dots enclosed in a circle that has its center at coordinate point (horizInteger, vertInteger) and a radius of radiusInteger. This addition to Quickdraw offers a limited alternative to the **invertOval** procedure.

label

■ **Pascal reserved word**

label is used in the definition part of a program to identify a 1- to 4-digit number as a **goto** location reference.

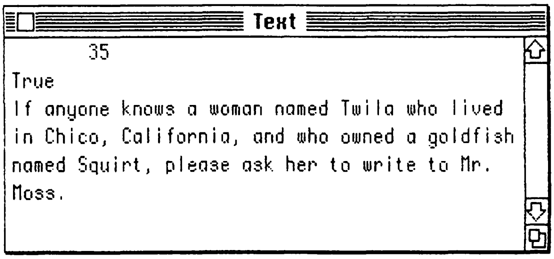
See **goto** for more information.

length

■ **UCSD Pascal function**

length(stringExpression) returns the number of characters in the string parameter. Each space within a string is considered a character. The **length** function returns an *integer* value.

```
program LengthDemo;
 var
  s1, s2, s3, s4 : string[50];
begin
 s1 := 'If anyone knows a woman named Twila';
 s2 := ' who lived in Chico, California, and';
 s3 := ' who owned a goldfish named Squirt,';
 s4 := ' please ask her to write to Mr. Moss.';
 writeln(length(s1));
 writeln(length(s2) > length(s3));
 if length(concat(s1, s2, s3, s4)) <= 160 then
  writeln(s1, s2, s3, s4)
end.
```



```
          35
True
If anyone knows a woman named Twila who lived
in Chico, California, and who owned a goldfish
named Squirt, please ask her to write to Mr.
Moss.
```

line

■ **Quickdraw procedure**

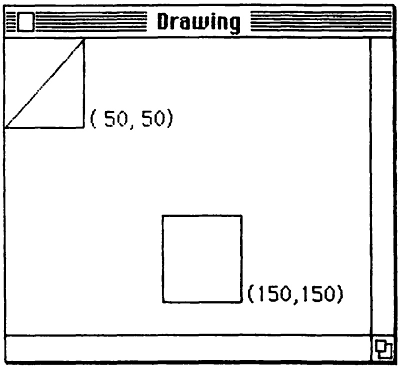
line(horizInteger, vertInteger) draws a line starting from the current Quickdraw pen location to a distance that is *horizInteger* dots to the right or left, and *vertInteger* dots up or down. The parameters of **line** measure a distance; they *are not* the coordinates of the line's endpoint.

Positive parameters draw a line to the right or down. This is consistent with the coordinate map of the Drawing window whose origin, point (0, 0), is the upper-left corner of the window.

After the **line** procedure is completed, the current pen location becomes the point at the end of the drawn line. If the starting point is coordinate (x, y), then the endpoint is (x + horizInteger, y + vertInteger).

The procedure **line** (x + horizInteger, y + vertInteger) is equivalent to the procedure **lineTo**(x, y).

```
program LineDemo;
 var
  pt : point;
 procedure triangle;    {moves and draws To coordinates}
 begin
  moveTo(50, 50);
  lineTo(50, 0);
  lineTo(0, 50);
  lineTo(50, 50);
  getPen(pt);
  writeDraw(' (', pt.h : 3, ',', pt.v : 3, ')');
  moveTo(50, 50)
 end;
 procedure square;    {moves and draws distances}
 begin
  move(50, 50);
  line(50, 0);
  line(0, 50);
  line(-50, 0);
  line(0, -50);
  move(50, 50);
  getPen(pt);
  writeDraw(' (', pt.h : 3, ',', pt.v : 3, ')')
 end;
begin
 triangle;
 square
end.
```



lineTo

■ **Quickdraw procedure**

lineTo(hCoordInteger, vCoordInteger) draws a line starting from the current Quickdraw pen location to the coordinate point location specified by the parameters. The parameters of **lineTo** are the coordinates of the line's endpoint, they *do not* measure a distance.

After the **lineTo** procedure is completed, the current pen location becomes the endpoint coordinate (hCoordInteger, vCoordInteger.)

See **line** for an example.

ln

■ **Pascal function**

ln(expression) evaluates a positive *real* or *integer* value, and returns the extended *real* value of its natural logarithm. **ln**(x) computes log to the base e($\log_e$) of x. **e** is approximately 2.718.

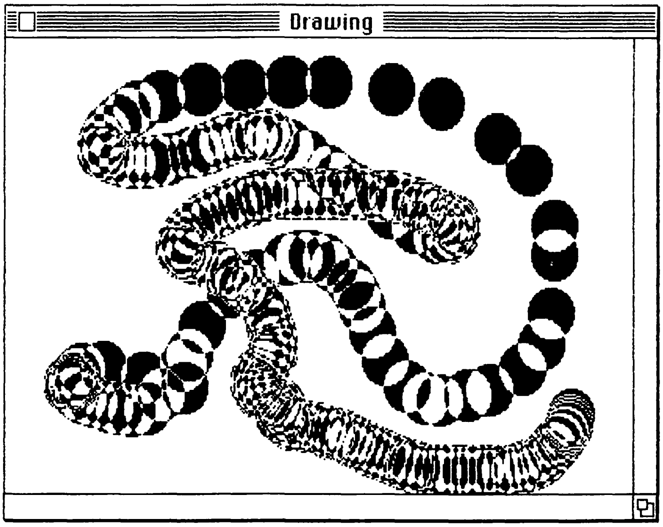
The exponential function, **exp**(x), performs the inverse of the natural log function, **ln**(x).

localToGlobal

■ **Quickdraw procedure**

localToGlobal(pointName) converts a point expressed in the local coordinates of the Drawing window to global coordinates—such as the Macintosh screen. The variable pointName of type *point* is created using the **setPt** procedure.

```
program LocalToGlobalDemo;
 var
  drWindow : rect;
 procedure bop;
  const
   offset = 15;
  var
   x, y : integer;
   pt : point;
 begin
  getMouse(x, y);
  setPt(pt, x, y);
  localToGlobal(pt);
  if ptInRect(pt, drWindow) then
   invertOval(y - offset, x - offset, y + offset, x + offset);
 end;
begin
 setRect(drWindow, 50, 50, 466, 326);
 setDrawingRect(drWindow);
 showDrawing;
 repeat
  bop
 until button
end.
```



longint

■ **UCSD Pascal type**

longint is a predefined integer-type whose members include all positive and negative whole numbers between −2,147,483,647 and 2,147,483,647 inclusive.

Variables declared to be of type **longint** can take any value in the range stated above. Whole numbers outside the range of **longint** and numbers requiring floating-point notation can be declared using a real-type.

The **longint** type provides for a much larger subset of whole numbers than does the **integer** type. Conversely, the use of **integer** types provides an economy of memory space when the **integer** range is sufficient for the particular application. Integers can take a value from −32,767 to 32,767.

See **integer** and **real** for more information.

maxint
maxlongint

■ **Pascal constant**
UCSD Pascal constant

maxint and **maxlongint** are predefined constants representing the largest allowable values of types *integer* and *longint* respectively.

The *integer* constant **maxint** is defined to be 32,767.

The *longint* constant **maxlongint** is defined to be 2,147,483,647.

mod

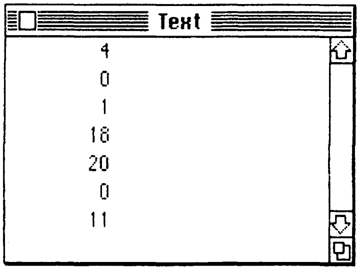
■ **Pascal reserved word**

mod works a mathematical symbol in the same family as +, −, *, /, and **div**. **mod** computes a *division returning a remainder result.*

The value of p **mod** q is the remainder part of the quotient p/q. The expressions p and q must be an integer-type.

Whereas **div** provides the largest whole-number quotient of a division ignoring any remainder, **mod** ignores the whole number and provides only the integer remainder.

```
program ModDemo;
 var
   i : integer;
begin
 writeln(9 mod 5);
 writeln(10 mod 5);
 writeln(11 mod 5);
 for i := 1 to 4 do
   writeln(random mod 21)
end.
```



```
 4
 0
 1
18
20
 0
11
```

move

■ **Quickdraw procedure**

move(horizInteger, vertInteger) moves the Quickdraw pen starting from the current pen location to a distance that is *horizInteger* dots to the right or left, and *vertInteger* dots up or down. The parameters of **move** measure a distance; they *are not* the coordinates of the new pen location.

move and **moveTo**, unlike **line** and **lineTo**, do not perform drawing. Like lifting a pencil to begin a drawing elsewhere, they only move the current Quickdraw pen location.

Positive numbers move a line to the right or down. This is consistent with the coordinate map of the Drawing window whose origin, point (0,0), is the upper-left corner of the window.

After the **move** procedure is completed, the current pen location becomes the endpoint. If the starting point is coordinate (x,y), then the endpoint is (x + horizInteger, y + vertInteger).

The procedure **move** (x + horizInteger, y + vertInteger) is equivalent to the procedure **moveTo**(x,y).

See **line** for an example.

moveTo

■ **Quickdraw procedure**

moveTo(hCoordInteger, vCoordInteger) moves a line starting from the current Quickdraw pen location to the coordinate point location specified by the parameters. The parameters of **moveTo** are the coordinates of the new pen location, they *do not* measure a distance.

moveTo and **move**, unlike **lineTo** and **line**, do not perform drawing. Like lifting a pencil to begin a drawing elsewhere, they only move the current Quickdraw pen location.

After the **moveTo** procedure is completed, the current pen location becomes the coordinate (hCoordInteger, vCoordInteger).

See **line** for an example.

newFileName

■ **Toolbox function**

newFileName(promptString) first produces a dialog box on the Macintosh screen with the parameter promptString displayed inside. The promptString gives a message instructing the user to type in a name for a new datafile. After a name has been typed and the Save button clicked (or the *return* key pressed), a datafile with the new name is created on the active disk and the function returns the user-selected file name.

The dialog box allows for different disks to be inserted and either drive to become active. The function returns the selected file name, which in turn can be opened by the program using **reset, rewrite,** or **open.**

Both the prompt and the function's result value are string types. An error will result if no file name is typed into the dialog box.

See **rewrite,** or Chapter 15 in Part 2 for an example.

nil

■ **Pascal reserved word**

The use of **nil** is an advanced Pascal topic relating to pointers. Because of its status as a reserved word, **nil** cannot be used in any other context. MacPascal will format any occurrence of **nil** in bold lettering. As with any reserved word, a program cannot redefine the name.

not

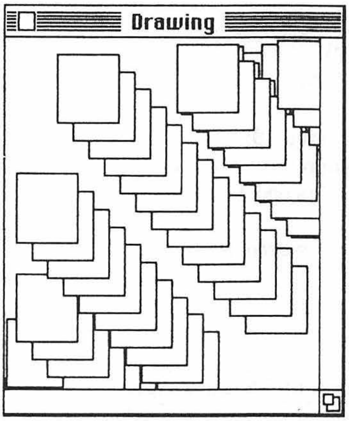
Pascal reserved word

not works to reverse the boolean value of the expression it immediately precedes. **not** before a true value creates a false value. **not** before a false value creates a true value.

Expressions using **not** require any equalities and inequalities (=, <, >, <=, >=) to be parenthesized.

not has the highest precedence of the boolean operators **not, and,** and **or.**

```
program NotDemo;
 var
  x, y : integer;
begin
 repeat
  getMouse(x, y);
  while not (x < 0) and not (y < 0) do
   begin
    eraseRect(y, x, y + 40, x + 40);
    frameRect(y, x, y + 40, x + 40);
    x := x - 10;
    y := y - 10
   end
 until button        {press Button to end}
end.
```



note

Toolbox procedure

note(frequency, amplitude, duration) evaluates three parameters to produce a single square-wave tone. The value of frequency must be a *longint* type in the range 12..783360. The values of amplitude and duration must be integers in the range 0..255.

See **sysBeep** for additional sound capability.

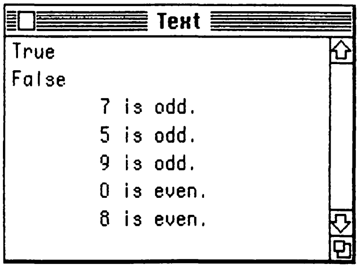
```
program NoteDemo;
 const
  time = 4;    {time range [0..255]}
 var
  freq : longint;
  amp : integer;
begin
 freq := 1;     {freq range [-32767..32767], cannot be 0}
 repeat
  amp := 0;   {amp range [0..255]}
  repeat
   note(freq, amp, time);
   amp := amp + 15
  until amp = 255;
  freq := freq + 1000
 until freq > 31767
end.
```

odd

■ Pascal function

odd(expression) returns a boolean value of true if the parameter is an odd number, or a value of false if the parameter is an even number. The parameter must be an integer-type.

```
program OddDemo;
 var
  i, x : integer;
begin
 writeln(odd(21));
 writeln(odd(4));
 for i := 1 to 5 do
  begin
   x := random mod 10;
   if odd(x) then
    writeln(x, ' is odd.')
   else
    writeln(x, ' is even.')
  end
end.
```



```
True
False
       7 is odd.
       5 is odd.
       9 is odd.
       0 is even.
       8 is even.
```

of

■ **Pascal reserved word**

of is used in conjunction with the reserved words **array, file,** and **set** in the declaration part of a program. **of** is also used as part of the **case** statement to separate the selector expression from the list of constants.

See each of the above reserved words for formats and examples.

offsetRect

■ **Quickdraw procedure**

offsetRect(varRectName, horizInteger, vertInteger) changes the coordinate position of the rectangle rectName. The variable parameter rectName is of the predefined Quickdraw type *rect,* created with the **setRect** procedure.

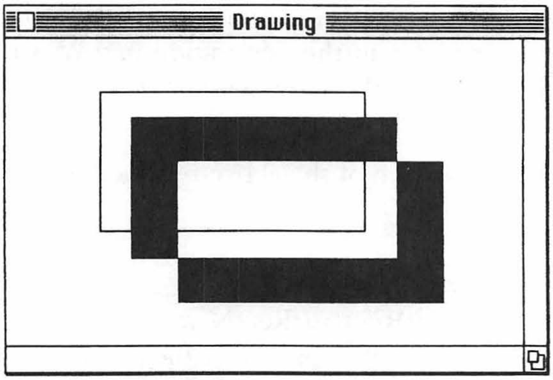
The left and right sides of rectName are moved to the right by positive values of horizInteger, and moved to the left by negative values. The horizontal coordinates of both sides are changed by adding the amount of horizInteger.

The top and bottom sides of rectName are moved downward by positive values of vertInteger, and moved upward by negative values. The vertical coordinates of both sides are changed by adding the amount of vertInteger.

The **offsetRect** procedure does not affect the shape or size of the rectangle. The procedure simply moves the rectangle to different coordinates.

The new position of the rectangle is not drawn onto the Macintosh screen by the **offsetRect** procedure. Drawing within the new rectangle must be done with a shape-drawing routine.

```
program offsetRectDemo;
 var
  r : rect;
begin
 setRect(r, 80, 45, 250, 125);
 paintRect(r);
 offsetRect(r, 30, 25);
 invertRect(r);
 offsetRect(r, -50, -40);
 frameRect(r)
end.
```



oldFileName

■ **Toolbox function**

oldFileName(promptString) first produces a dialog box on the Macintosh screen with the parameter promptString displayed inside. The promptString gives a message instructing the user to select a file name from among those displayed in the dialog box. After a file has been selected and the Open button clicked or the *return* key pressed), the function returns the user-selected file name.

The dialog box will display all the files on the active disk. When more than seven files are on the disk, the dialog box's mini-finder uses a scroll bar to allow the user to view all the files on the disk.

The dialog box also allows for different disks to be inserted and either disk drive (on a two-drive system) to become active. The user can mouse-select any file, which in turn can be opened by the program using **reset, rewrite,** or **open.**

Both the prompt and the function's result are string-types.

See **reset,** or Chapter 16 in Part 2 for an exmaple.

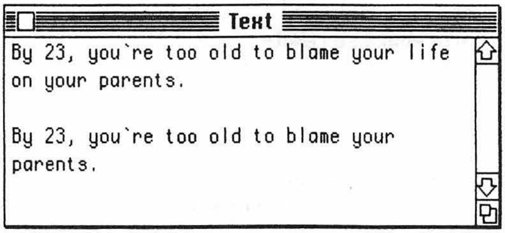
omit

■ **Macintosh function**

omit(string,positionInteger,lengthInteger) removes characters from the string parameter beginning at the positionIntegerth character and continuing for lengthInteger characters. The resulting string is returned.

The **omit** function differs from the **delete** procedure in that **omit** returns a result string without affecting the parameter string, whereas **delete** changes the value of the returning variable parameter string.

```
program OmitDemo;
 var
  s1, s2 : string[75];
begin
 s1 := 'By 23, you're too old to blame your life on your parents.';
 s2 := omit(s1, pos('your', s1), length('your life on '));
 writeln(s1);
 writeln;
 writeln(s2)
end.
```



```
By 23, you're too old to blame your life
on your parents.

By 23, you're too old to blame your
parents.
```

open

Macintosh procedure

Use **open** to open a new or existing file for random access. **open** allows a component to be read from or written to the file. The format for **open** is:

open(fileName, deviceName);

The file organizer *fileName* represents all the components contained in the datafile on the external device *deviceName.*

After a file is opened with **open**, the file organizer is positioned at the first component, number 0. The file organizer's current component can be read or written to through the buffer variable, *fileName^.*

Any component can be randomly accessed by positioning the file organizer with the **seek** procedure. The **seek** procedure uses an integer parameter to directly position the file organizer at a specified component file position. Unlike **reset** and **rewrite,** files opened with **open** do not sequentially advance the buffer following each read or write operation.

If the opened file is empty, the **eof** function returns true and the buffer variable *fileName^* is undefined.

```
program OpenDemo;    {see SeekDemo to read the file created here}
 var
  mossNote : array[0..20] of string;
 procedure makeNote;
 begin
  mossNote[0] := 'From Mr. Moss's purple notebook:';
  mossNote[13] := 'Wheat Hearts cereal sticks to your ribs; broccoli is magic.';
  mossNote[8] := 'Don't skimp on tires or shoes--they connect you to earth.';
  mossNote[5] := 'Ask only that your mate smells good and doesn't get too fat.';
  mossNote[3] := 'To make romance last, take lots of walks and watch sunsets.';
  mossNote[16] := 'When sick and in bed, television is a miracle drug.';
  mossNote[11] := 'Ken Kesey and J. D. Salinger ought to give us more to read.';
 end;
 procedure storeNote;
  const
   empty = '';
  var
   i : integer;
   fileName : string[50];
   mossNoteFile : file of string;
 begin
  fileName := newFileName('type a datafile name');
  open(mossNoteFile, fileName);
  for i := 0 to 20 do
   begin
    seek(mossNoteFile, i);
    if mossNote[i] <> empty then    {empty is the const ''}
     mossNoteFile^ := mossNote[i]
    else
     mossNoteFile^ := '';
    put(mossNoteFile)
   end;
  close(mossNoteFile)
 end;
begin
 makeNote;
 storeNote
end.
```

or

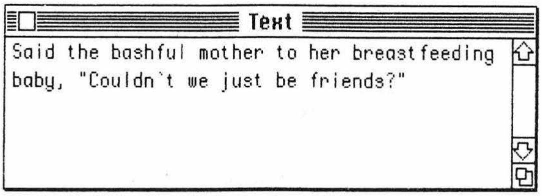
■ Pascal reserved word

or unites two boolean (true/false) expressions into a new single boolean expression. The new expression is true if either or both original expressions are true. The new expression is false only if both original expressions are false.

Expressions using **or** require any equalities and inequalities (=, <, >, =<, =>) to be parenthesized.

or has the least precedence of the boolean operators **not, and,** and **or.**

```
program OrDemo;
 var
  justFriends, intimacy : boolean;
begin
 justFriends := (1 + 1 = 5) or (2 + 2 = 9);
                  {false or false = false}
 intimacy := (1 + 1 = 2) or (2 + 2 = 4);
                  {true or true = true}
 if justFriends or intimacy then
                  {false or true = true}
  begin
   write('Said the bashful mother to her breastfeeding baby,');
   writeln(' "Couldn't we just be friends?"')
  end
end.
```



```
Said the bashful mother to her breastfeeding
baby, "Couldn't we just be friends?"
```

ord

Pascal function

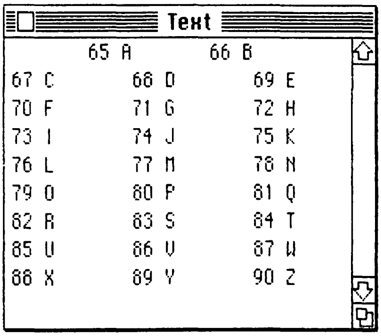
The function **ord**(orderedExpression) returns the ordinal number assigned to the parameter. The ordinal number is an integer-type.

Ordinal numbers are assigned according to the ASCII character set. The sample program below shows how to create such a table of characters and their ordinal numbers.

If the parameter is a pointer type, the function result is the address of the dynamic variable pointed to by expression.

See *char* and *chr* for more information.

```
program OrdDemo;
 var
  ch : char;
begin
 for ch := 'A' to 'Z' do
  begin
   write(ord(ch));
   write(' ', ch)
  end
end.
```



```
        65 A        66 B
67 C        68 D        69 E
70 F        71 G        72 H
73 I        74 J        75 K
76 L        77 M        78 N
79 O        80 P        81 Q
82 R        83 S        84 T
85 U        86 V        87 W
88 X        89 Y        90 Z
```

ord4

■ **Macintosh function**

ord4(ch) returns a *longint* value by converting the ordered or pointer type value of its parameter *ch*. The value of **ord4**(ch) is the same as **ord**(ch).

otherwise

■ **Pascal reserved word**

otherwise can be used in conjunction with the **case** statement after the list of case constants. If the **case** expression does not match any of the constants, the statement(s) following **otherwise** will be performed.

See **case** for more information and an example.

output

■ **Pascal file**

output is a predeclared file allowing write operations to the Macintosh screen. MacPascal does not require **output**, or the predeclared read file **input**, to be explicitly stated. Other versions of Pascal require **output** and **input** to be stated as program parameters.

Program parameters are never required in MacPascal. **output** and **input** are the only program parameters allowed.

The format is:

program programName(input, output);

pack

■ **Pascal procedure**

pack(arrayName, index, packedName) transfers the contents of the ordinary array arrayName to the packed array packedName, starting at the indexth position of arrayName. index must be of a type that is compatible with the index-type of arrayName.

See **packed** and **unpack** for more information.

packed

■ **Pascal reserved word**

The internal storage of structured types can be modified by preceding the type's declaration with the word **packed.**

packed types will be compressed to economize storage space, possibly at the expense of increased access time to components of variables of these types. Unpacked, or ordinary types, are stored in such a way as to make access time efficient by spacing components uniformly.

Record, file, and set types cannot be declared as **packed.** Any arrays can be declared as **packed,** but MacPascal will actually pack only the component types *char, 0..255 (unsigned),* and *−128..127 (signed).*

page

■ **Pascal procedure**

Use **page**(textfileName) to advance the printer to the top of the next page or to clear the Macintosh screen. Only files of type *text* can be referenced. If the textfileName parameter is omitted, the Macintosh screen is assumed as the standard output device.

paintArc
paintOval
paintRect
paintRoundRect

■ **Quickdraw procedures**

In the following procedures, rectName is a variable of type *rect,* which can be created by using the **setRect** procedure. Alternatively, rectName can be substituted with four integers representing the rectangle's boundary coordinates *top, left, bottom,* and *right,* respectively.

All procedures paint the specified shape with the Quickdraw pen's currently selected pattern and draw-over mode. The pen location is unaffected. See **penMode, penPat,** and **penSize** for more information on the Quickdraw pen.

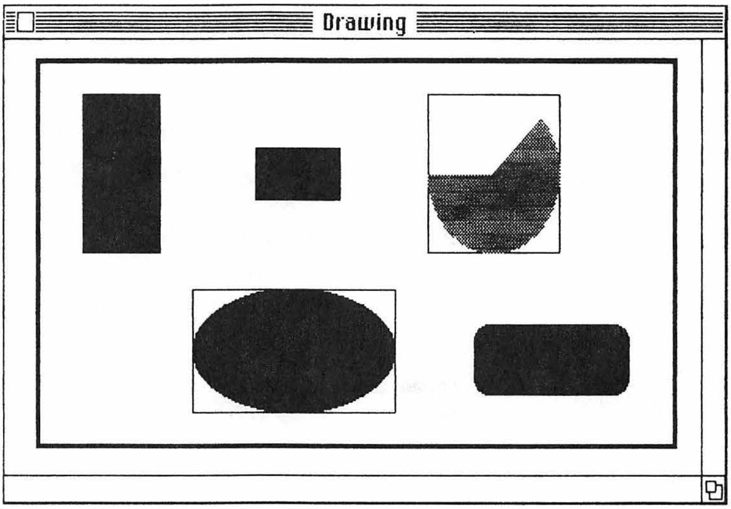
paintArc(rectName, startAngleInteger, arcAngleInteger) paints a wedge of the oval that fits within the rectangular dimensions set by rectName. The parameter startAngleInteger is a degree value between 0 and 359 that works like the hand of an oval clock: 0 points to 12 o'clock, 90 points to 3 o'clock, 180 points to 6 o'clock, and so on. The parameter arcAngleInteger is a degree value between −359 and 359 that sets the extent of the arc, positive angles extending clockwise, negative angles counterclockwise.

paintOval(rectName) paints an oval that fits within the rectangular dimensions set by rectName.

paintRect(rectName) paints a rectangle within the dimensions set by rectName.

paintRoundRect(rectName, ovalWidthInteger, ovalHeightInteger) paints a rounded-corner rectangle within the dimensions set by rectName. The curvature of the rounded corners is set by two integers that specify the diameters of an oval shape suggested by the rounded corners.

```
program PaintDemo;
 procedure createDrawingWindow;
  var
   drWindow : rect;
 begin
  hideAll;
  setRect(drWindow, 20, 60, 480, 320);
  setDrawingRect(drWindow);
  showDrawing
 end;
 procedure paintArt;
  var
   r1, r2, r3, r4 : rect;
 begin
  paintRect(60, 160, 90, 215);
  setRect(r1, 50, 30, 100, 120);
  setRect(r2, 120, 140, 250, 210);
  setRect(r3, 300, 160, 400, 200);
  setRect(r4, 270, 30, 355, 120);
  paintRect(r1);
  frameRect(r2);
  paintOval(r2);
  paintRoundRect(r3, 20, 20);
  frameRect(r4);
  penPat(gray);
  paintArc(r4, 45, 225);
  penSize(3, 3);
  penPat(black);
  frameRect(10, 20, 230, 430)
 end;
begin
 createDrawingWindow;
 paintArt
end.
```



paintCircle

■ **MacPascal procedure**

paintCircle(horizInteger, vertInteger, radiusInteger) paints a circle with the center point given by the coordinate point (horizInteger, vertInteger) with a radius of radiusInteger. This MacPascal addition to Quickdraw offers an alternative, and more limited, circle-drawing procedure to **paintOval**

penMode

■ **Quickdraw procedure**

penMode(modeName) determines how the Quickdraw pen will *draw over* the existing dot at a particular location on the Macintosh screen. The eight available modes listed in the table below cause the pen's *inkdots* to draw differently depending on the selected pen *pattern* and whether the pen is drawing over a black dot or a white dot.

Ordinarily, the pen draws in black dots, but the Quickdraw pen can also draw in white dots or in a *thick line pattern* made up of both black and white inkdots. The following table shows the color dot each of the eight modes will produce according to the pen's inkdot and the dot already on the screen.

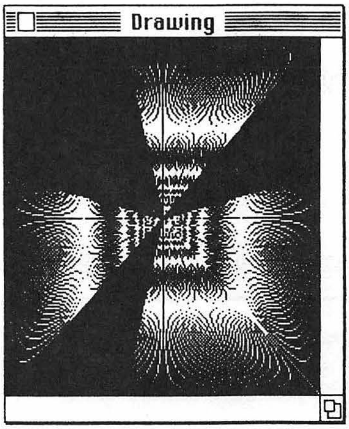
modeName	black inkdot	white inkdot
patCopy	always black	always white
patOr	always black	unchanged
patXor	invert	unchanged
patBic	always white	unchanged

modeName	black inkdot	white inkdot
notPatCopy	always white	always black
notPatOr	unchanged	always black
notPatXor	unchanged	invert
notPatBic	unchanged	always white

The initial setting of penMode is to patCopy. In this mode, black ink will always draw a black dot, no matter which dot it is drawing over, and white ink will always draw a white dot.

```
program penModeDemo;
 var
  mode : integer;
 procedure setMode;
 begin
  case mode of
   1 :
    penMode(patXor);
   2 :
    penMode(patOr);
   3 :
    penMode(patBic);
   4 :
    penMode(patCopy);
  end
 end;
 procedure lines;
  const
   core = 100;
   edge = 200;
  var
   i : integer;
 begin
  for i := 0 to edge do
   begin
    moveTo(core, core);
    lineTo(i, 0);
    moveTo(core, core);
    lineTo(edge, i);
    moveTo(core, core);
    lineTo(0, i);
    moveTo(core, core);
    lineTo(i, edge)
   end
 end;
```

```
begin
 for mode := 1 to 4 do
  begin
   setMode;
   lines;
  end
end.
```



penNormal

■ **Quickdraw procedure**

penNormal resets the characteristics of the Quickdraw pen to the initial settings. **penSize** becomes (1,1), **penMode** becomes patCopy, **penPattern** becomes black. The location of the pen does not change.

See **penPat** or **penSize** for an example.

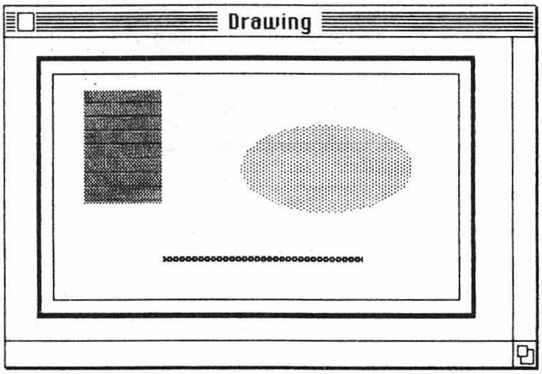
penPat

■ **Quickdraw procedure**

penPat(patternName) sets the ink pattern of the Quickdraw pen. Five patterns are predefined: *black, white, gray, ltGray,* and *dkGray.* The initial pen pattern is black.

Custom patterns can be designed by declaring and assigning a variable of type *pattern,* a predefined Quickdraw type. The type *pattern* is a packed array [0..7] of [0..255].

```
program PenPatDemo;
begin
 penPat(gray);
 paintRect(30, 50, 95, 100);
 penPat(ltGray);
 paintOval(50, 150, 100, 260);
 penPat(dkGray);
 penSize(3, 3);
 moveTo(100, 125);
 lineTo(225, 125);
 penPat(black);
 frameRect(10, 20, 160, 300);
 penNormal;
 frameRect(20, 30, 150, 290)
end.
```



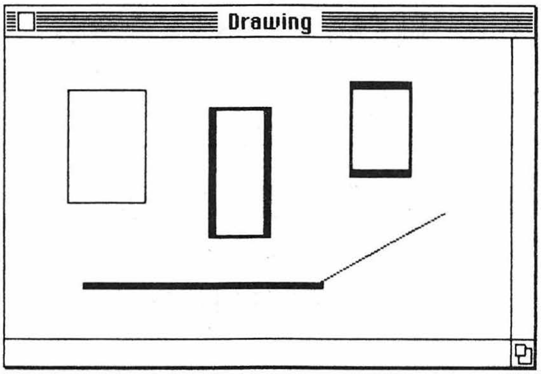
penSize

■ **Quickdraw procedure**

penSize(widthInteger, heightInteger) sets the thickness dimensions of the Quickdraw pen. All line drawings and framed shapes are drawn with a pen thickness as set by **penSize.**

The initial setting of **penSize** is (1,1), its thinnest dimensions. If either parameter is set to zero or a negative value, the pen will not draw anything.

```
program PenSizeDemo;
begin
 frameRect(30, 40, 95, 90);
 penSize(5, 2);
 frameRect(40, 130, 115, 170);
 penSize(2, 5);
 frameRect(25, 220, 80, 260);
 penSize(4, 4);
 moveTo(50, 140);
 lineTo(200, 140);
 penNormal;
 lineTo(280, 100)
end.
```



pos

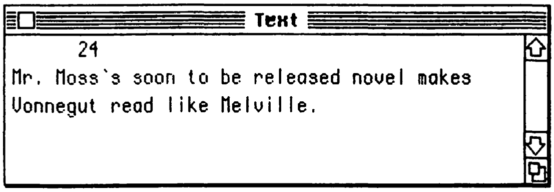
■ UCSD Pascal function

pos(findString, sourceString) returns an integer corresponding to the position of findString within sourceString. The integer corresponds to the n^{th} character of sourceString where findString begins.

The n^{th} character of sourceString is determined by the number of characters preceding the occurrence of the first character of findString. If findString does not exactly match characters in sourceString, the **pos** function returns a value of zero.

The parameters of the **pos** function must be a string type.

```
program PosDemo;
 var
  s1, s2 : string[75];
begin
 s1 := 'Mr. Moss`s novel makes Vonnegut read like Melville.';
 s2 := 'Vonnegut';
 writeln(pos(s2, s1));
 insert('soon to be released ', s1, pos('novel', s1));
 writeln(s1)
end.
```



```
      24
Mr. Moss`s soon to be released novel makes
Vonnegut read like Melville.
```

pred

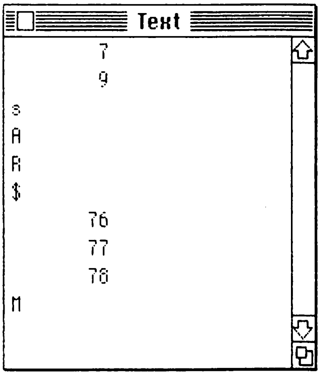
■ Pascal function

pred(orderedExpression) returns a value that precedes the value of the parameter. The parameter must belong to an ordered type.

If the parameter is the first value of an ordered type, the **pred** function remains undefined.

See **succ** for information on the complementary successor function.

```
program PredSuccDemo;
begin
 writeln(pred(8));
 writeln(succ(8));
 writeln(pred('t'));
 writeln(pred('B'));
 writeln(succ('Q'));
 writeln(pred('%'));
 writeln(pred(ord('M')));
 writeln(ord('M'));
 writeln(succ(ord('M')));
 writeln(pred(succ('M')))
end.
```



```
         7
         9
8
A
A
$
        76
        77
        78
M
```

procedure

■ **Pascal reserved word**

A **procedure** works like a small Pascal program inserted into a larger Pascal program. Using procedures to accomplish specific tasks, makes writing and reading Pascal clearer and easier.

Two important differences distinguish a procedure from a self-contained program.

First, a procedure cannot run by itself. Statement execution cannot begin inside a procedure until the **procedure** name is first called from outside. A program, on the other hand, will begin executing its main body on a run command without reference to the **program** name.

Second, a procedure can keep its data private from the rest of a program, exchanging only data that is globally declared or transmitted through its *parameters.*

A procedure heading has the following format:

procedure name(formal parameters—if any);

Any definitions or declarations should follow the heading in the same manner as the **program** heading. These definitions and declarations are local, and for use only within the procedure. Following this are the procedure's statements bracketed by **begin** and **end.**

To execute a procedure, the calling statement has this format:

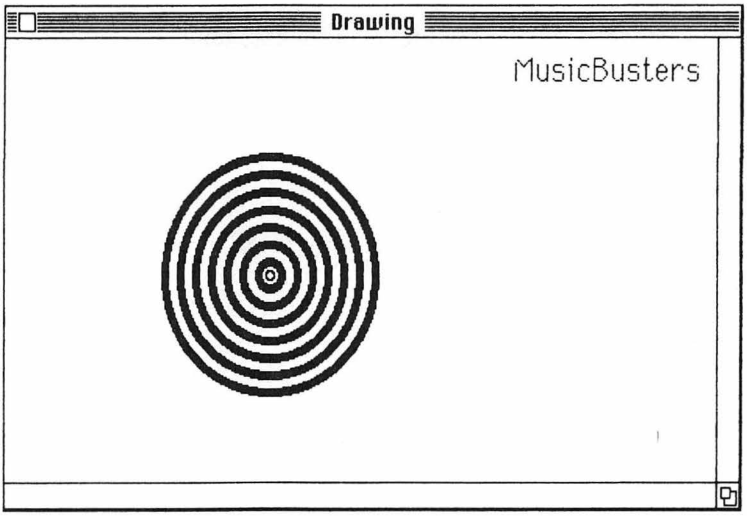
procedureName(values/variables—if any);

The name only, without the reserved word **procedure,** calls the procedure. The values/variables in parentheses, sometimes called *actual parameters,* must correspond in number, type, and position to the parameters listed in the procedure heading.

Parameters work to exchange data between procedures and the part of the program from which they are called. The two primary kinds of parameters are *value* and *variable.* Value parameters send values from the procedure call to the procedure. Variable or **var** parameters send variable values (only) back from the procedure to the procedure call.

Pascal's predefined procedures require only that the procedure be called using appropriate parameters.

```
program ProcedureDemo;
 var
  horiz, vert : integer;
 procedure windows;
  var
   r : rect;
 begin
  hideAll;
  setRect(r, 20, 50, 490, 320);
  setDrawingRect(r);
  showDrawing
 end;    {end windows}
 procedure title (s : string);
 begin
  moveTo(325, 25);
  textSize(18);
  writeDraw(s)
 end;
 procedure readMouse (var x, y : integer);
 begin      {to demonstate var parameters only}
  getMouse(x, y);
 end;    {end readMouse}
 procedure buster (x, y : integer);
  const
   amp = 225;
   time = 7;
   eyePop = 5;
   core = 3;
   var
    i : integer;
   procedure sound (a, t : integer);   {a procedure inside another procedure}
    var
     freq : longint;
   begin
    freq := random mod 30 + 1;
    freq := freq * 1000;
    note(freq, a, t)
   end;      {end procedure sound}
  begin
   i := eyePop;
   repeat
    frameOval(y - core, x - core, y + core, x + core);
    sound(amp, time);
    invertOval(y - i, x - i, y + i, x + i);
    i := i + eyePop
   until i = 75;
   eraseOval(y - i, x - i, y + i, x + i)
  end;    {end buster}
 begin
  windows;
  title('MusicBusters');
  repeat
   readMouse(horiz, vert);
   buster(horiz, vert);
  until button
 end.    {end ProcedureDemo}
```



program

■ **Pascal reserved word**

program must be the first word of every Pascal program. The format for the first line of every Pascal program is:

program programName;

The parameter programName following **program** cannot be accessed or reused by any instruction within the program. The name is required and used only for identification.

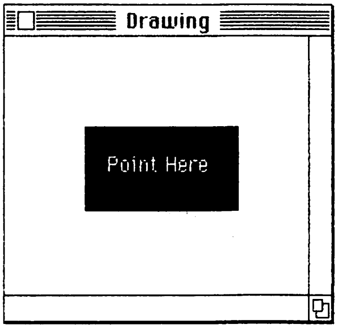
ptInRect

■ **Quickdraw function**

ptInRect(pointName, rectName) evaluates a *point* type and a *rect* type parameter, and returns the boolean result of true if the dot below and to the right of the coordinate point is enclosed in the given rectangle. Otherwise the function returns a value of false.

The Quickdraw predefined types *point* and *rect* can be assigned variables using the **setPt** and **setRect** procedures, respectively.

```
program PtInRectDemo;
 var
  x, y : integer;
  r : rect;
  pt : point;
begin
 SetRect(r, 50, 50, 150, 100);
 FrameRect(r);
 MoveTo(65, 77);
 WriteDraw('Point Here');
 repeat
  GetMouse(x, y);
  SetPt(pt, x, y);
  if PtInRect(pt, r) then
   InvertRect(r)
 until button
end.
```



pt2Rect

Quickdraw procedure

pt2Rect(point1Name, point2Name, varRectname) evaluates two parameters of type *point,* and returns the smallest rectangle that encloses the two points in the variable parameter varRectName. The result varRectName is a variable of type *rect.*

put

Pascal procedure

put(fileName) inserts the value of the buffer variable *fileName^* at the current file position of the file organizer, then advances the file organizer by one position. If the new position of the file organizer is beyond the last component of the file, then the value of *fileName^* becomes undefined.

Subsequent calls to the **put** procedure will insert a component into the next sequential position of the file. The **put** procedure advances itself through a file one component at a time.

To randomly insert a component, that is, without putting values into all preceding components of the file, first use the procedure **seek**. **seek** uses an integer parameter to directly position the file organizer anywhere along the numbered file. Then **put**(fileName) will insert the value of the buffer variable at the current position of the file organizer.

put inserts a new file component, then advances the file organizer. In order to read an existing file component, use the file command **get**.

```
program PutDemo;     {see GetDemo to read this file}
 var
  giveName, fileName : string[50];
  Jaime : string[20];
  friends : file of string[20];
```

```
begin
 giveName := 'type in new file name';
 fileName := newFileName(giveName);
 rewrite(friends, fileName);
 friends^ := 'Charlie';
 put(friends);
 friends^ := 'Kate';
 put(friends);
 friends^ := 'Black';
 put(friends);
 Jaime := 'sweet Jaime';
 friends^ := Jaime;
 put(friends);
 close(friends)
end.
```

Quickdraw1
Quickdraw2

■ Macintosh units

Quickdraw is a predefined set of procedures, functions, and definitions built into the Macintosh circuitry that has been integrated into the Macintosh Pascal language. For space economy, MacPascal has divided the Quickdraw library into the two units, **Quickdraw1** and **Quickdraw2**.

Quickdraw1 contains all of Quickdraw except those declarations involving grafPorts, regions, pictures, polygons, bit transfer operations, and customizing Quickdraw operations. Those are included in **Quickdraw2**.

The unit name **Quickdraw1** may optionally be included in the **uses** clause of a program for LisaPascal compatibility, but is unnecessary in MacPascal because the unit is included automatically.

The unit name **Quickdraw2** *must* be included in the **uses** clause immediately following the program heading in order to make use of its features. The format for this is:

```
uses Quickdraw2;
```

The additional memory requirements of **Quickdraw2** makes its use limited on 128K Macintoshes.

random

Quickdraw function

random returns a single integer from −32768 through 32767. The function generates a uniformly distributed pseudorandom integer.

See **mod** for an example and information on producing **random** numbers between specific boundaries.

read

Pascal procedure

The **read** procedure works similar to **readln** except that the end-of-line character will not halt the procedure and advance to the file's next component.

See **readln** for more information and an example.

readln

Pascal procedure

Use **readln** as a one-step process to access a single component of a file and assign that component to a variable. The format for this procedure is:

```
readln(fileName, componentVar);
```

If the file organizer *fileName* is omitted, the file device is assumed to be the *keyboard.* In this instance, the program will wait until input is received from the keyboard.

The **readln** procedure will access a single component from the file named in the first parameter, and assign its value to the variable named as the second parameter.

The same process could be accomplished by the following statements:

```
componentVar := fileName^;
get(fileName);
```

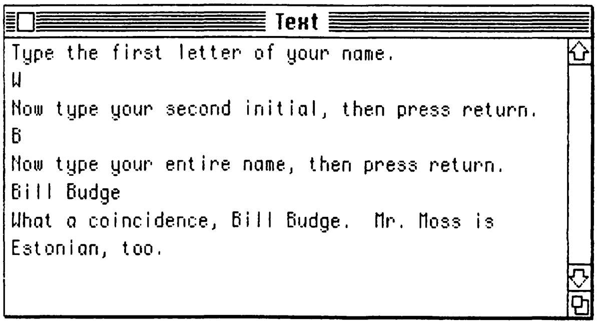
The **readln** procedure should only be used for inputting *char* or *string* variables. Numeric input should be read as a *char* type, then converted to a numeric equivilent using the **ord** function. Reading numeric input directly makes a program too susceptible to error.

The file-type of the file organizer affects the way **readln** (and **writeln**) operate. For nontextfiles, read and write values must be of type *char.* For textfiles, read and write values can be of other types, and their values will be translated to and from their character representations.

read will not recognize an end-of-line mark as will **readln.** For this reason, **readln** is the recommended procedure for reading strings.

See **text** for an example of **readln** with file parameters.

```
program ReadlnDemo;
 var
  ch : char;
  s : string[30];
begin
 writeln('Type the first letter of your name.');
 read(ch);
 writeln;
 writeln('Now type your second initial, then press return.');
 readln(ch);
 writeln('Now type your entire name, then press return.');
 readln(s);
 writeln('What a coincidence, ', s, '.  Mr. Moss is Estonian, too.')
end.
```



```
Type the first letter of your name.
W
Now type your second initial, then press return.
B
Now type your entire name, then press return.
Bill Budge
What a coincidence, Bill Budge.  Mr. Moss is
Estonian, too.
```

real

■ Pascal type

The name **real** is used both as a Pascal type and as a category that includes all four floating-point types.

The type **real** is one of four predefined real-types for representing numbers in floating-point notation. The others are *double, extended,* and *computational.*

The purpose of having more than one real-type as options is to provide the range and precision necessary for a particular program application without being wasteful of computer memory space. The higher the range and precision, the more memory space must be allocated.

Any number requiring a decimal point must be expressed as a real-type. All real-types represent numbers in floating-point notation. In floating-point notation, numbers are expressed as a power of ten.

The type **real** can accommodate values in a range from 1.5×10^{-45} to 3.4×10^{38} with precision of 7 to 8 digits.

The real-types *double* and *extended* extend the range and precision of the type **real.** The type *computational* is specifically designed for precise, fixed-decimal place applications such as financial programs. More information can be found under each real-type classification.

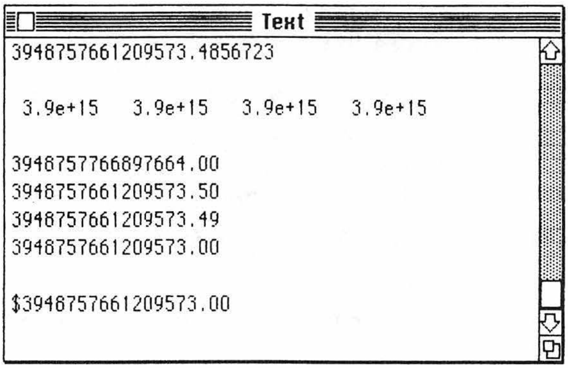
For arithmetic operations, all real-type values are converted to type *extended,* and the results are also type *extended.* When a **real** type value is required, the *extended* type can be used provided that the value, when rounded to an integral value, falls within the range allowed by **real.**

The procedures **write** and **writeln** allow formatting within their parameter lists to output a real-type value.

writeln(realValue: fieldWidthInteger : decimalPlacesInteger);

```
program RealDemo;
 var
  c : real;
  d : double;
  e : extended;
  f : computational;
 procedure makeDollar;
  var
   st : string;
 begin
  st := stringOf(f : 18 : 2);
  insert('$', st, 1);
  writeln;
  writeln(st)
 end;
```

```
begin
 writeln('Enter any large number including decimal point.');
 readln(e);
 c := e;
 d := e;
 f := e;
 writeln;
 writeln(c, d, e, f);
 writeln;
 writeln(c : 18 : 2);   {: minimum field width, : decimal places}
 writeln(d : 18 : 2);
 writeln(e : 18 : 2);   {note: MacPascal 1.0 has a decimal place bug when}
 writeln(f : 18 : 2);   {field width parameter is smaller than actual width}
 makeDollar
end.
```



```
3948757661209573.4856723

 3.9e+15   3.9e+15   3.9e+15   3.9e+15

3948757766897664.00
3948757661209573.50
3948757661209573.49
3948757661209573.00

$3948757661209573.00
```

record

■ Pascal reserved word

A **record** is a user-defined type composed of two or more variable *fields* each with their own type. A record has the following format:

```
type recordName = record
      fieldName1 : fieldType1;
      fieldName2 : fieldType2;
      fieldName3, fieldName4 :
      fieldType3
  end;
```

To use a record, its structure must first be defined in the **type** section of a program. Once defined, variables can be declared of the **record** type. Defined types can also be used to define subsequently listed types.

Use the **record** type to group together logically associated variables. These variables, or *fields,* can be accessed individually or as a whole.

Individual fields are accessed by referencing a variable of type recordName, then the name of the record field. The variable and field must be separated by a single period.

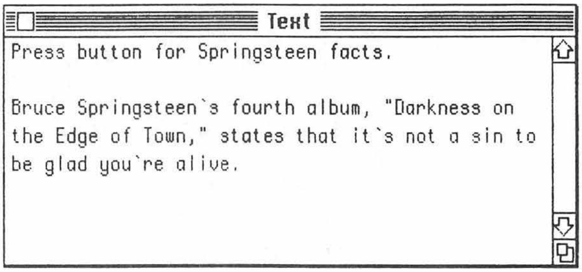
A field of a record can be another record. To reference fields of the imbedded record, append its fieldName onto the reference of the outer field, once again separated by a single period.

Record fields may also be referenced using the reserved word **with.** See **with** for more information.

The purpose of giving variables a **record** structure is to provide programming clarity. The **record** type works like a suitcase, holding together chosen belongings and offering a handle—the recordName—to make traveling easier.

```
program RecordDemo;
 type
  album = record
     artist : string[50];
     title : string[50];
     year : integer;
    end;
 var
  Bruce4 : album;
 procedure init;
 begin
  Bruce4.artist := 'Bruce Springsteen';
  Bruce4.title := 'Darkness on the Edge of Town';
  Bruce4.year := 1978;
 end;
 procedure display;
 begin
  writeln('Press button for Springsteen facts.');
  repeat    {wait until button is pressed}
  until button;
  writeln;
  write(Bruce4.artist, ''s fourth album, "', Bruce4.title);
  writeln(',"' states that it's not a sin to be glad you're alive.');
 end;
```

```
begin
 init;
 display
end.
```



```
Press button for Springsteen facts.

Bruce Springsteen's fourth album, "Darkness on
the Edge of Town," states that it's not a sin to
be glad you're alive.
```

repeat..until

■ **Pascal reserved word**

repeat..until creates a loop that will perform an action one or more times, stopping only when the boolean expression following the word **until** is true. The **repeat** loop takes the following format:

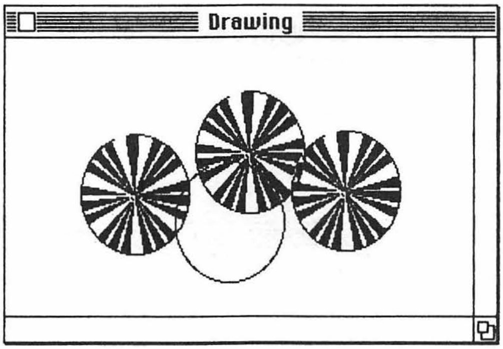
```
repeat
  statement
until booleanExpression
```

Since the loop's stopping mechanism—the boolean expression—comes after the statement action, the **repeat** loop will always perform the specified action *at least one time.*

This contrasts with the **while** loop whose stopping mechanism comes *before* the statement action. The **while** loop *may never* perform its action if its boolean expression never has a true value.

Because the word **repeat** indicates the beginning of the loop and **until** indicates the end of the loop, the reserved words **begin** and **end** are not needed to bracket an action of more than one statement.

```
program RepeatDemo;
 procedure arcAround;
  const
   arcLen = 15;
   arcJump = 55;
   spins = 8;
  var
   x, y, arcA, revolve : integer;
 begin
  getMouse(x, y);
  arcA := 0;
  revolve := 0;
  repeat
   frameOval(y, x, y + 70, x + 70);
   invertArc(y, x, y + 70, x + 70, arcA, arcLen);
   arcA := (arcA + arcJump);
   if arcA >= 360 then
    begin
     arcA := arcA - 360;
     revolve := revolve + 1
    end
  until revolve = spins
 end;
begin
 repeat
  arcAround
 until button
end.
```



reset

■ **Pascal procedure**

Use **reset** to open an existing file before any reading or writing operation on a sequential file. The format for **reset** is:

reset(fileName, deviceName);

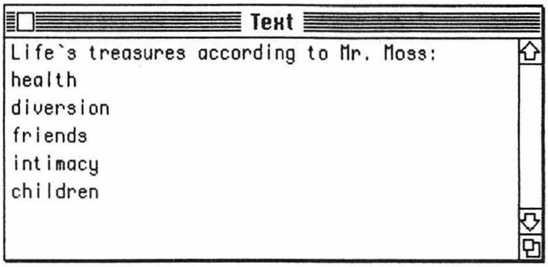
The file organizer *fileName* represents all of the components contained in the datafile on the external device *deviceName.*

After a file is opened with **reset**, the file organizer is always positioned at the first component, number 0. The file organizer's current component can be read or written to through the buffer variable *fileName^.*

All subsequent access to a file opened with **reset** must be done sequentially, one component at a time. For random access to a file's components, the **open** procedure must be used to create and open the file.

Opening a file with **reset** will not affect the file's contents. Opening a file with **rewrite** will erase its contents. Use **rewrite** only to open a new file.

```
program ResetDemo;    {ResetDemo`s file was created in RewriteDemo}
 var
  i : integer;
  fileName : string[50];
  treasures : array[0..5] of string[50];
  treasureFile : text;
begin
 fileName := oldFileName('select file name');
 reset(treasureFile, fileName);
 for i := 0 to 5 do
  begin
   readln(treasureFile, treasures[i]);
   writeln(treasures[i])
  end;
 close(treasureFile)
end.
```



```
Life`s treasures according to Mr. Moss:
health
diversion
friends
intimacy
children
```

rewrite

■ Pascal procedure

Use **rewrite** to open a new file before any reading or writing operation on a sequential file. If the parameter for **rewrite** specifies a file that already exists, the file's contents will be erased. The format for **rewrite** is:

rewrite(fileName, deviceName);

The file organizer *fileName* represents all of the components contained in the datafile on the external device *deviceName.*

After a file is opened with **rewrite**, the file organizer is always positioned at the first component, number 0. The file organizer's current component can be written to through the buffer variable *fileName^*.

All subsequent input to a file opened with **rewrite** must be done sequentially, one component at a time. For random access to a file's component positions, the **open** procedure must be used to create and open the file.

Use **rewrite** only to open a new file. All files opened with **rewrite** are empty regardless of their prior contents. To open an existing file without destroying its contents, use **reset**.

```
program RewriteDemo;    {See ResetDemo to read this file}
 var
  i : integer;
  fileName : string[50];
  treasures : array[0..5] of string[50];
  treasureFile : text;
begin
 fileName := newFileName('type in  new file name');
 rewrite(treasureFile, fileName);
 treasures[0] := 'Life's treasures according to Mr. Moss:';
 treasures[1] := 'health';
 treasures[2] := 'diversion';
 treasures[3] := 'friends';
 treasures[4] := 'intimacy';
 treasures[5] := 'children';
 for i := 0 to 5 do
  writeln(treasureFile, treasures[i]);
 close(treasureFile)
end.
```

round

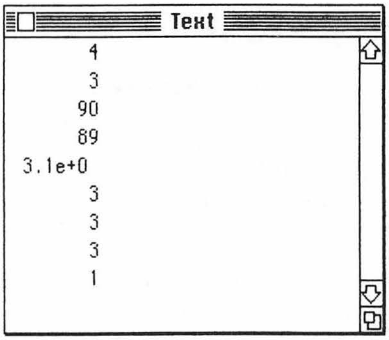
■ Pascal function

Use the function **round**(realNumber) to return the integer closest in value to the real number parameter.

If the real numbeı is exactly between two integers—the fractional part of the real number equal to 0.5—then positive reals are rounded to the higher integer and negative reals are rounded to the lower integer.

The **round** function returns an integer-type.

```
program RoundDemo;
begin
 writeln(round(3.7));
 writeln(trunc(3.7));
 writeln(round(89.5));
 writeln(trunc(89.5));
 writeln(22 / 7);
 writeln(round(22 / 7));
 writeln(trunc(22 / 7));
 writeln(22 div 7);
 writeln(22 mod 7)
end.
```



```
       4
       3
      90
      89
 3.1e+0
       3
       3
       3
       1
```

SANE

■ Macintosh unit

SANE is the name of a predefined unit offering extended mathematical procedures and functions. It is an acronym for Standard Apple Numeric Environment.

The SANE unit can be included in a program by including this declaration below the program heading: **uses** SANE;. The features of SANE are described in the *Macintosh Pascal Reference Manual.*

SANE also refers to the arithmetic methodology used throughou MacPascal. This intrinsic use of SANE requires no **uses** declaration.

See **uses** for more information and an example.

saveDrawing

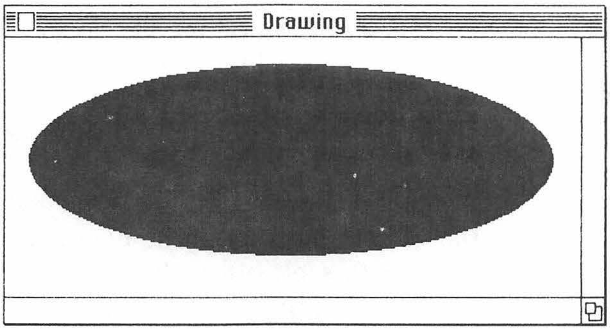
■ **Toolbox procedure**

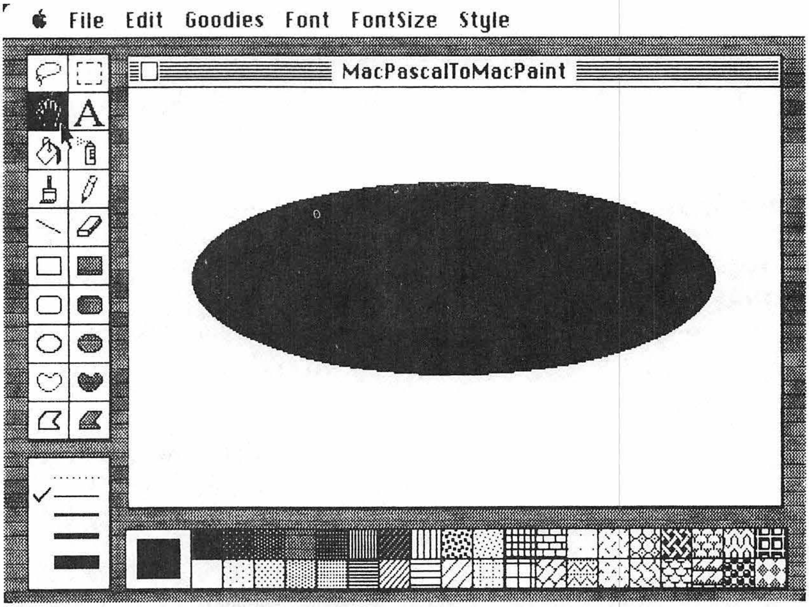
Use **saveDrawing**(fileName) to save the contents of the Drawing window as a picture file. The picture file then can be accessed by MacPaint.

The parameter fileName is a string containing the name of the picture file to be created. The file will replace any existing file with the same name.

If the current Quickdraw grafPort has been changed from the default Drawing window grafPort, **saveDraw** will save the current grafPort instead of the Drawing window.

```
program SaveDrawingDemo;
begin
 paintOval(15, 15, 125, 350);
 saveDrawing('MacPascalToMacPaint')
end.
```





seek

UCSD Pascal procedure

Use **seek** to position the file organizer at a specified component number of a random access file. The format for **seek** is:

seek(fileName, componentNumber);

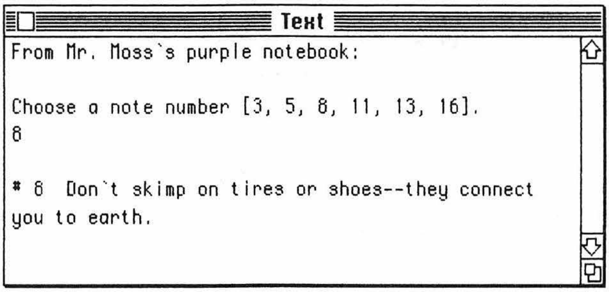
When a random access file is opened with the **open** procedure, the current position of the file organizer is the first component, number 0. **seek** offers direct access to a particular component by advancing the file organizer *componentNumber* positions from the beginning of the file.

Once the file organizer has been positioned, the buffer variable *fileName^* can be used to insert a component into the file or read a component from the file.

If the file organizer is positioned at or before the last component of the file, the value of *fileName^* becomes the value of the current component. If the file organizer is positioned beyond the last component of the file, the value of *fileName^* is undefined, and *eof(f)* becomes true.

seek requires two parameters. The component number must be a positive integer.

```
program SeekDemo;    {run OpenDemo to create the file read here}
 const
  empty = '';
 var
  note : integer;
  fileName : string[50];
  mossNote : array[0..20] of string;
  mossNoteFile : file of string;
begin
 fileName := oldFileName('select the file to read');
 open(mossNoteFile, fileName);
 writeln(mossNoteFile^);
 writeln;
 writeln('Choose a note number [3, 5, 8, 11, 13, 16].');
 readln(note);
 writeln;
 if note in [3, 5, 8, 11, 13, 16] then
  begin
   seek(mossNoteFile, note);
   write('*', filepos(mossNoteFile) : 2, ' ');  {write out the file number}
   writeln(mossNoteFile^)
  end
 else
  writeln('Sorry,' but Twila says you didn't type 3, 5, 8, 11, 13, or 16.');
 close(mossNoteFile)
end.
```



```
From Mr. Moss's purple notebook:

Choose a note number [3, 5, 8, 11, 13, 16].
8

* 8  Don't skimp on tires or shoes--they connect
you to earth.
```

set of

■ Pascal reserved word

A set is a group of values belonging to the same type. **set of** establishes a user-defined Pascal type composed of a group of values. The individual members of a set must be of an ordered type. The format for **set of** is:

```
type
  typeName = set of orderedType;
```

In MacPascal, the values of orderedType must be in the range −8192 to 8191.

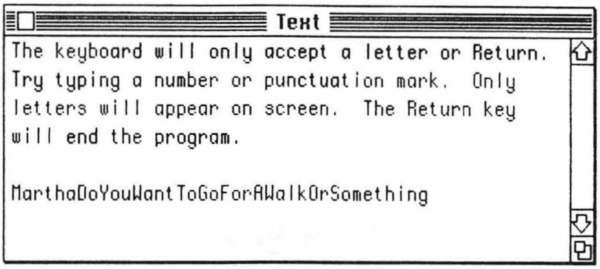
After a set has been defined, variables can be declared of the set type. Within a program, a variable of a set type can be assigned any value that is a member of the set.

Notation for individual members of a set requires *square brackets.* An inclusive list of set members can be shortened by using two dots [..] between the boundary members.

Sets may be evaluated using the relational expressions =, <=, >=, <>, and **in.**

Sets can also be used arithmetically. The *union* of two sets, containing all members of both sets, is obtained using the plus sign (+). The *difference* of two sets, containing all members of the first set that are not members of the second set, is obtained using the minus sign (−). The *intersection* of two sets, containing only members of the first set that are also members of the second set, is obtained using the multiplication sign (*).

```
program SetDemo;
 type
  keys = set of char;
 var
  fineLetter : keys;
  ch1 : char;
 function checkChar (fineKey : keys) : char;
  var
   fine : boolean;
   ch : char;
  begin
   repeat
    ch := input^;
    get(input);
    fine := ch in fineKey;
    if not fine then
     sysBeep(5)
    else
     write(ch)
   until fine;
   checkChar := ch
  end;
begin
  fineLetter := ['a'..'z', 'A'..'Z'];
  write('The keyboard will only accept a letter or Return.');
  write(' Try typing a number or punctuation mark.');
  write(' Only letters will appear on screen.');
  writeln(' The Return key will end the program.');
  writeln;
  repeat
   ch1 := checkChar(fineLetter)
  until eoln
end.
```



```
The keyboard will only accept a letter or Return.
Try typing a number or punctuation mark.  Only
letters will appear on screen.  The Return key
will end the program.

MarthaDoYouWantToGoForAWalkOrSomething
```

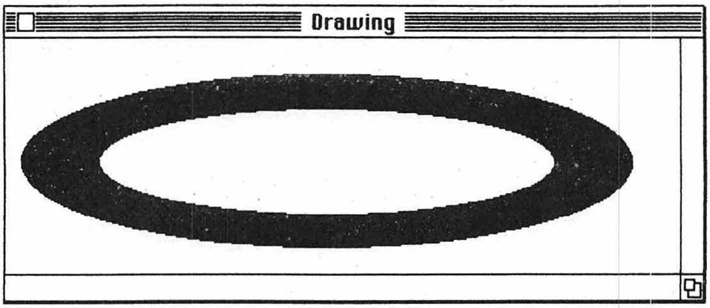
setDrawingRect

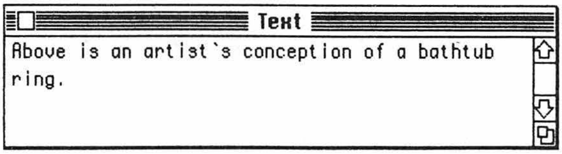
■ Toolbox procedure

Use **setDrawingRect**(windowRect) to establish the position and size of the Drawing window on the Macintosh screen. Note that **setDrawingRect** does not actually display the window. Use the **showDrawing** procedure to display the Drawing window.

The value of windowRect is of type *rect*. See **setRect** for information on creating a variable of type *rect*.

```
program SetDrawingRectDemo;
 procedure doDrawing;
  var
   drWindow : rect;
 begin
  setRect(drWindow, 30, 50, 475, 200);
  setDrawingRect(drWindow);
  showDrawing;
  paintOval(20, 10, 120, 400);
  invertOval(40, 60, 100, 350)
 end;
 procedure doText;
  var
   txWindow : rect;
 begin
  setRect(txWindow, 150, 240, 500, 320);
  setTextRect(txWindow);
  showText;
  writeln('Above is an artist`s conception of a bathtub ring.')
 end;
begin
 hideAll;
 doDrawing;
 doText
end.
```





```
Above is an artist`s conception of a bathtub
ring.
```

setPt

■ **Quickdraw procedure**

setPt(pointName, horizInteger, vertInteger) assigns the horizontal and vertical integer coordinates to the variable pointName. pointName is of the Quickdraw type *point.*

Most Quickdraw procedures involving points require coordinates to be expressed as type *point* rather than two integers. The **setPt** procedure performs this task.

See **addPt** or **ptInRect** for an example.

setRect

■ **Quickdraw procedure**

setRect(rectName, leftInteger, topInteger, rightInteger, bottomInteger) assigns the four side or boundary coordinates to the variable rectName. rectName is of the Quickdraw type *rect.*

Rectangles, rounded-corner rectangles, ovals, arcs, and wedges are drawn within a rectangle's coordinates. The coordinates can be assigned to a single variable of type *rect* using **setRect.** The variable rectName can then be used as the single parameter necessary to define a rectangle.

The alternative to using **setRect** is to insert the four boundary integers of a rectangle directly into the particular procedure's parameter list.

The use of **setRect** and the *rect* variable to define a rectangle makes no significant difference in code size or execution speed than its four integer counterpart, however, the *rect* variable is easier to write and manipulate.

See **frameRect** for an example.

setTextRect

■ **Toolbox procedure**

Use **setTextRect**(windowRect) to establish the position and size of the Text window on the Macintosh screen. Note that **setTextRect** does not actually *display* the window. Use the **showText** procedure to display the Text window.

The value of windowRect is of type *rect.* See **setRect** for information on creating a variable of type *rect.*

See **setDrawingRect** for an example.

setTime

■ **Toolbox procedure**

setTime(recordName) sends date and time information to the Macintosh system clock. The parameter of **setTime** is of *record* type.

The **setTime** procedure sends data to the system clock through its value parameter. The **getTime** procedure receives data from the system clock through its **var** parameter.

See **getTime** for the type definition of *dateTimeRec,* the record used by both **setTime** and **getTime.**

showDrawing

■ **Toolbox procedure**

Use **showDrawing** to display the Drawing window on the Macintosh screen. **showDrawing** makes the Drawing window the active window. The procedure has no parameters.

The position and size of the Drawing window are unaffected by **showDrawing.** Use the **setDrawingWindow** procedure to establish the window's position and size.

See **setDrawingRect** for an example.

showText

■ **Toolbox procedure**

Use **showText** to display the Text window on the Macintosh screen. **showText** makes the Text window the active window. The procedure has no parameters.

The position and size of the Text window are unaffected by **showText.** Use the **setTextWindow** procedure to establish the window's position and size.

See **setTextRect** for an example.

sin

■ **Pascal function**

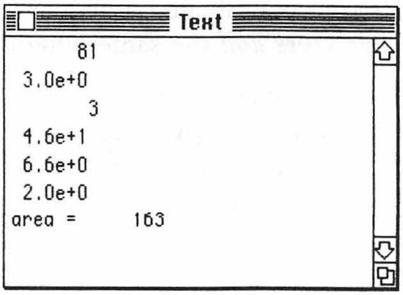
sin(expression) evaluates a single *real* or *integer* angle parameter expressed in radians, and returns a *real* value corresponding to the angle's sine.

sqr

■ **Pascal function**

sqr(expression) evaluates a single *real* or *integer* parameter, and returns the square of the parameter, expressed in the same type.

```
program SqrDemo;
  const
    pi = 3.1416;
begin
  writeln(sqr(9));
  writeln(sqrt(9));
  writeln(round(sqrt(9)));
  writeln(sqr(6.81));
  writeln(sqrt(43.26));
  writeln(sqrt((15 + 5) / 5));
  writeln('area =', round(sqr(7.2) * pi))
end.
```



```
      81
 3.0e+0
       3
 4.6e+1
 6.6e+0
 2.0e+0
area =      163
```

sqrt

■ **Pascal function**

sqrt(expression) evaluates a single *real* or *integer* parameter, and returns the square root of the parameter, expressed as a *real* number. The parameter cannot be less than zero.

See **sqr** for an example.

string

■ **Pasal reserved word**

string is a predefined type whose members are a sequence of characters. Each **string** type has a declared length, stated in brackets, equal to the maximum number of characters a variable of that type can contain. The format for the **string** type is:

string [integer];

The specification of a **string** length can range from 1 to 255. If the length specification is omitted, a default length of 255 is assigned.

Variables of a declared **string** type can be any number of characters up to and including the length specification. The exact length of a variable will be returned by the integer function **length**(stringVariable).

A *null* string has no characters and has a length of 0. A null string is notated by two consecutive single quotation marks [“].

The size of a **string,** different from the length, can be used to compare strings. Size comparisons are evaluated by the ordinal value of the first pair of nonidentical characters. If a shorter string has identical characters to the beginning of a longer string, the longer string is evaluated as larger. Two strings must have the same characters *and* the same length to be considered equal.

Strings can be manipulated using the following procedures and functions: **concat, copy, delete, include, insert, length, omit, pos,** and **stringOf.** See each classification for more information and **string** examples.

stringOf

■ **MacPascal function**

stringOf(writeVariable) returns its parameter as a string-type value. The function works like the *write* procedure except that instead of displaying its parameter on the screen, **stringOf** returns the characters of the parameter as a string value.

Like the parameter of a *write* procedure, the parameter of **stringOf** can include *colon modifiers.* Colon modifiers specify, in order, minimum field width and number of decimal places. The format for **stringOf** with colon modifiers is:

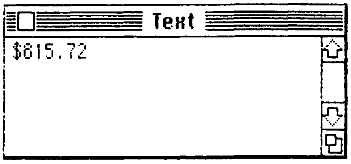
stringOf(writeVariable : fieldWidthInteger : decPlaceInteger);

Colon modifiers are explained in more detail in Chapter 25 of Part 2.

Unlike the *write* procedure, **stringOf** does not allow for a file parameter. The function returns a string-type value only to the program. The function's parameter can be an integer-type, real-type, char-type, string-type, packed-string-type or enumerated-type.

The primary reason for converting other types into a string value is to take advantage of numerous *string* functions and procedures that test and manipulate the characters of a string.

```
program StringOfDemo;
  var
    dollar : longint;
    stDollar : string[10];
begin
  dollar := 81572;
  stDollar := stringOf(dollar : 5);
  insert('.', stDollar, length(stDollar) - 1);
  insert('$', stDollar, 1);
  writeln(stDollar)
end.
```



```
$815.72
```

subPt

■ Quickdraw procedure

subPt(sourcePoint,destinationPoint) changes the coordinates of destinationPoint by subtracting their value by the coordinates of sourcePoint. The procedure returns with a new value of destinationPoint.

Both parameters of **subPt** are of the Quickdraw type *point.* See **setPt** for more information on type *point.*

See **addPt** for an example.

succ

■ Pascal function

succ(ordered Expression) returns a value that succeeds—that is, follows—the value of the parameter. The parameter must belong to an ordered type.

If the parameter is the last value of an ordered type, the **succ** function returns undefined.

See **pred** for information of the complementary predecessor function and an example of both.

sysBeep

■ **ToolBox procedure**

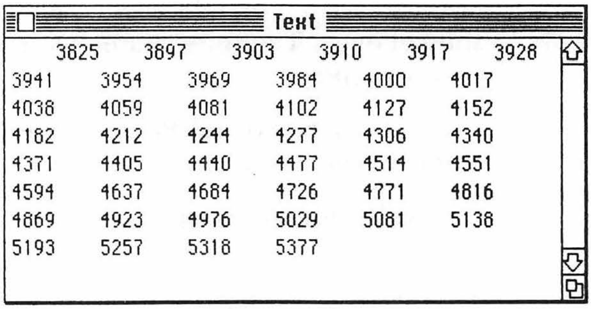
sysBeep(durationInteger) produces a simple square-wave tone. The integer parameter determines the amount of time the tone lasts.

A durationInteger value of 45 lasts approximately 1 second as does each increment of 45. A value of 90 lasts about 2 seconds, and so on. The sound produced by a single call to **sysBeep** fades within 5 to 6 seconds, so parameter values greater than 360 leave a silent gap.

The square wave produced by **sysBeep** is the same one produced when the Macintosh is turned on.

See **note** for additional sound capability.

```
program SysBeepDemo;
 var
  i : integer;
begin
 for i := 0 to 45 do
  begin
   sysBeep(i);
   write(tickCount)
  end
end.
```



```
     3825      3897      3903      3910      3917      3928
3941      3954      3969      3984      4000      4017
4038      4059      4081      4102      4127      4152
4182      4212      4244      4277      4306      4340
4371      4405      4440      4477      4514      4551
4594      4637      4684      4726      4771      4816
4869      4923      4976      5029      5081      5138
5193      5257      5318      5377
```

text

■ **Pascal type**

text is a predefined file type whose components are characters organized into lines. Files of this kind are called textfiles. The organization into lines makes textfiles unique; however, in most instances, a file of type **text** resembles and responds like a **file** of char.

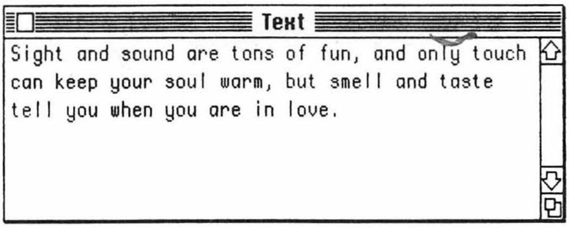
The **eoln** function can be used in conjunction with textfiles to test for the end-of-line character. The *return* key issues an end-of-line character.

Non-textfiles can be created using the declaration **file of** componentType. See **file of** for more information on these file-types.

Files to be sent to a printer must be declared to be of type **text.** The standard file *input* associated with the Macintosh keyboard and the standard file *output* associated with the Macintosh screen are textfiles.

Specialized input and output tasks using a **text** file parameter are discussed in the *Macintosh Pascal Reference Manual.*

```
program TextDemo;
 var
  s1, s2 : string;
  sense : text;
 procedure toPrinter;
 begin
  rewrite(sense, 'printer:');   {open printer to receive text file}
  writeln(sense);   {send a line feed to straighten paper}
  write(sense, chr(27), 'c', chr(27), 'p');   {set printer to standard style}
  writeln(sense, s1);   {write text on printer paper}
  writeln(sense, s2);
  close(sense);
 end;
 procedure toDisk;   {disk write and read}
 begin
  rewrite(sense, 'MacPascal 1.04: senseData');   {open disk to receive text}
  writeln(sense, s1, s2);   {write text on disk}
  close(sense);
  reset(sense, 'MacPascal 1.04: senseData');   {open disk to read text}
  readln(sense, s2, s1);   {read text from disk}
  writeln(s2, s1);   {display text in text window}
  close(sense)
 end;
begin
 s1 := 'Sight and sound are tons of fun, and only touch can keep your ';
 s2 := 'soul warm, but smell and taste tell you when you are in love.';
 toPrinter;
 toDisk
end.
```



Sight and sound are tons of fun, and only touch can keep your soul warm, but smell and taste tell you when you are in love.

textFace

Quickdraw procedure

textFace([styleName]) sets the style for text written in the Drawing window. The seven predefined styles are: *bold, italic, underline, outline, shadow, condense,* and *extend.*

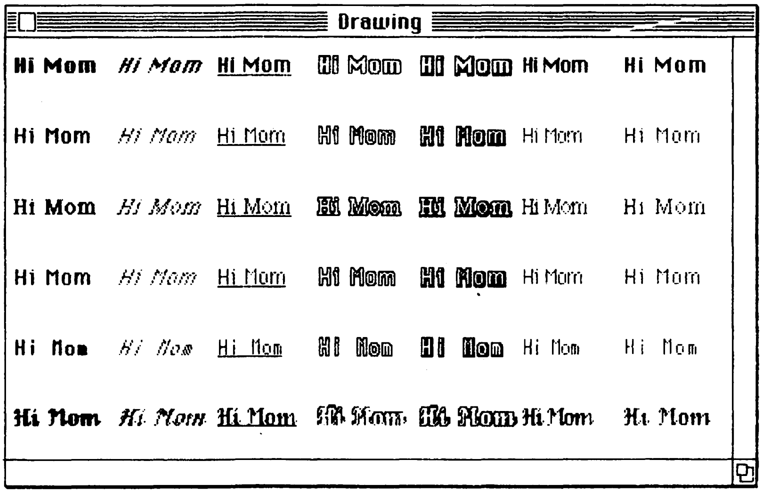
The style name parameter must be enclosed in brackets.

More than one style can be implemented at the same time. The format for setting more than one style is:

textFace([styleName 1, styleName2]);

The initial or normal style setting is obtained using empty brackets: **textFace**([]).

```
program TextFace;
 type
  face = (bold, italic, underline, outline, shadow, condense, extend);
 var
  i, x, y : integer;
  style : face;
begin
 y := 20;
 for i := 0 to 5 do
  begin
   x := 5;
   textFont(i);        {default textSize is 12}
   for style := bold to extend do
    begin
     textFace([style]);
     moveTo(x, y);
     writeDraw('Hi Mom');
     x := x + 65
    end;
   y := y + 40
  end
end.
```



```
Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom
Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom
Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom
Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom
Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom
Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom  Hi Mom
```

textFont

■ **Quickdraw procedure**

textFont(integer) sets the font for text written in the Drawing window. The system font is the initial setting represented by 0.

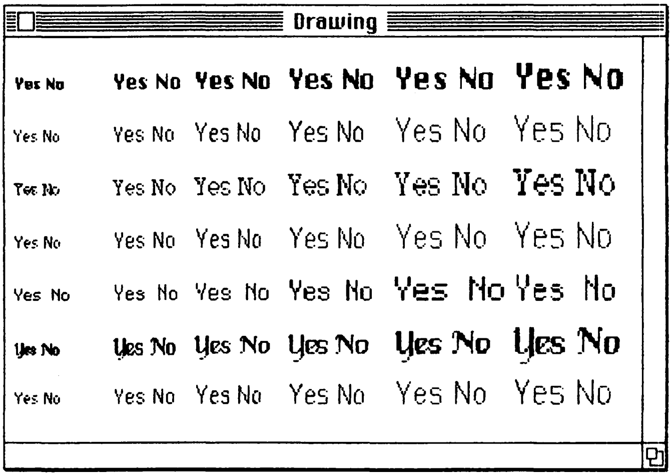
Five different fonts are available, numbered 0 through 5. **textFont**(1) and **textFont**(3) are identical. Paramaters over 5 will default to setting 1.

The names and point sizes of the fonts on the Pascal disk are as follows:

#0 Chicago (system font)—12 pt.
#1 Geneva (program window)—9, 12, 18 pts.
#2 Monaco (text window)—9, 12 pts.
#3 (same as #1)
#4 New York—12 pt.
#5 Venice—14 pt.
#6... (default to #1)

The New York and Venice fonts can be removed, and other fonts added, with the Font Mover utility program supplied with Macintosh.

```
program TextFontDemo;
 var
  i, p : integer;
begin
 for i := 0 to 6 do
  begin
   textFont(i);
   p := 9;
   repeat
    textSize(p);
    moveTo(p * p - 75, i * 30 + 30);
    if p = 9 then
     p := p + 3
    else
     p := p + 2;
    writeDraw('Yes No');
   until p = 22
  end
end.
```



```
Yes No  Yes No  Yes No  Yes No  Yes No  Yes No
Yes No  Yes No  Yes No  Yes No  Yes No  Yes No
Yes No  Yes No  Yes No  Yes No  Yes No  Yes No
Yes No  Yes No  Yes No  Yes No  Yes No  Yes No
Yes No  Yes No  Yes No  Yes No  Yes No  Yes No
Yes No  Yes No  Yes No  Yes No  Yes No  Yes No
Yes No  Yes No  Yes No  Yes No  Yes No  Yes No
```

textMode

■ **Quickdraw procedure**

textMode(modeName) determines how text will *write over* the current contents of the Drawing window. The names for text modes are: *srcOr, srcXor,* and *srcBic.*

The initial setting for **textMode** is srcOr.

See **penMode** more information on transfer modes.

textSize

■ **Quickdraw procedure**

textSize(integer) sets the size of text in the Drawing window. The integer parameter corresponds to the font's point size with one exception: a parameter of 0 will select the initial system font size of 12 point.

Any size can be selected. However, if Quickdraw does not have the font in the selected size, the nearest size will be scaled. This could result in funny looking letters. An even multiple of an available size for the font produces the best approximation.

See **textFont** for a listing of available fonts and point sizes, and an example.

then

■ **Pascal reserved word**

then precedes the action statement(s) as part of the **if..then** statement. See **if..then** for more information and an example.

tickCount

■ **Toolbox function**

tickCount returns a *longint* value representing the elapsed time, in sixtieths of a second, from the moment the Macintosh system starts to the moment it encounters the **tickCount** function.

See **sysBeep** for an example.

to

■ **Pascal reserved word**

to separates the two boundary expressions that determine the number of repetitions to be performed in the **for..to..do** loop. See **for..to..do** for more information.

true

■ **Pascal constant**

true is one of two predefined values of type *boolean.* Boolean is an enumerated type whose members are the ordered constants **false** and **true.**

```
type
  boolean = (false,true);
```

The written output of a boolean expression will be the word **false** or **true.** See the boolean function **odd** for an example.

trunc

■ **Pascal function**

trunc(expression) evaluates a single *real* parameter, truncates the fractional part of the expression, and returns the integer part. By removing the portion of the expression less than one, **trunc** rounds a real number to the nearest *integer* toward 0.

trunc will also return an *integer* type when the given parameter is a *longint* type.

See **round** for an example.

type

■ **Pascal reserved word**

type works like a generic classification. Pascal requires that data be identified by **type.** A **type** can be predefined or user-defined.

The reserved word **type** begins a definition section of a program. A user-defined **type** takes the following format:

```
type
   typeName = typeValue;
```

Below are the types in their categories. Predefined types are in *italics.*

1. simple type
   a. ordinal or ordered type
      1) *integer*
      2) *long integer*
      3) *boolean*
      4) *char*
      5) enumerated type
      6) subrange type
   b. *real* type

2. structured type
   a. array type
   b. set type
   c. record type
   d. file type
      1) fileName type
      2) *text*
   e. string type

3. pointer type

The user-defined types—enumerated, subrange, and structured—can be defined either in a **type** declaration or directly in the **var** declaration.

An enumerated type is an ordered list between parentheses with each component separated by a comma. The components of an enumerated type can take any name and their order is sequential, beginning at 0, according to the position each occupies in the list.

A subrange type consists of two constants separated by two dots (..). The components of the subrange include both constants and all ordered values between those constants.

A pointer type is used in advanced programming. Appendix A presents information on pursuing advanced topics.

Simple and structured types are explained separately under their own classifications.

See **set, textFace,** and **with** for examples of programs using **type** declarations.

unpack

■ Pascal procedure

unpack(packedName, arrayName, index) transfers the contents of the packed array packedName to the ordinary array arrayName, starting at the indexth position of arrayName.

See **packed** and **pack** for more information.

until

■ **Pascal reserved word**

until precedes the boolean expression of the **repeat..until** loop.

When the expression is false, the program execution loops back to **repeat** the action statement(s). When the expression is true, the loop stops repeating, and program execution continues at the statement following the **until** expression.

See **repeat..until** for more information and an example.

uses

■ **Pascal reserved word**

uses is the heading of a definition part of a program which allows separate and distinct sections of Pascal program code, called *units,* to be accessed. The host program does not contain the unit's code. The **uses** declaration automatically inserts the unit into memory.

In MacPascal, the only available units are predefined. Other versions of Pascal allow the user to create units.

Units work like libraries of code. They contain Pascal subprograms that are too bulky to have around all the time, but are available when you need them.

The **uses** clause must be placed immediately following the **program** declaration. The format to use MacPascal's predefined units is:

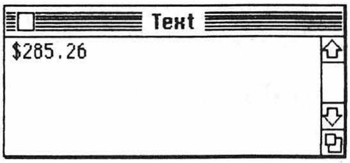
```
uses
  unitName;
```

MacPascal's units are *Quickdraw2* and *SANE.* Quickdraw2 offers extended graphic capabilities. SANE offers an extended mathematical environment.

Technically, *Quickdraw 1* is also a unit; however, programs do not need to declare Quickdraw 1 in a **uses** clause. Because the basic graphic instructions are used so often in Macintosh programming, the Quickdraw 1 unit has been automatically inserted into all MacPascal programs and works as if it is part of the Pascal language.

See the *Macintosh Pascal Reference Manual* for more information on using Quickdraw2 and SANE.

```
program UsesDemo;
 uses
  SANE;    {example of the financial compound function}
 var
  i, rate, periods : extended;
begin
 rate := 0.14;    {compounded interest:  $100 for 8 years at 14%}
 periods := 8;
 i := 100 * compound(rate, periods);    {(1 + 0.14) to the power of 8}
 writeln('$', i : 1 : 2)
end.
```



```
$285.26
```

var

■ **Pascal reserved word**

var (short for variable) is the heading of a declaration section of a program or program block. A variable is a name given to a particular type of data. The name can represent a variety of changing values, as long as each value belongs to the declared generic type.

The format for **var** is:

var
  variableName : type;

The **var** declaration can be used *globally,* below the **program** heading, to make the variable recognized throughout the program, or *locally,* within a procedure or function, to make the variable recognized only within that block.

The reserved word **var** also identifies *variable parameters.* A **var** parameter takes a value from within a procedure, and returns the value to a variable outside the block. The use of **var** parameters instead of global variables can enhance a program's efficiency in terms of memory space and modular clarity.

The format for a **var** parameter is:

  procedure procedureName(var variableName : type);

More than one **var** parameter requires only that the additional variableName(s)—and type if different—be included, separated by a comma. **var** parameters must be separated from *value* (untitled) parameters by a semicolon.

Examples of **var** can be found in almost all program examples.

while..do

■ **Pascal reserved word**

while..do creates a loop that will perform an action only if, and as long as, the boolean expression following the word **while** is true. The format for the **while** loop is:

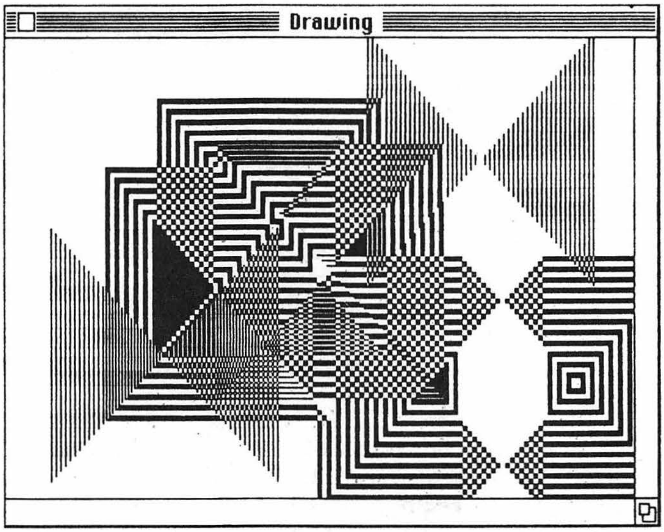
while booleanExpression **do**
  statement

If the action of the loop contains multiple statements, the statements must be bracketed by **begin** and **end.**

Since the loop's stopping mechanism—the boolean expression—comes before the statement-action, the **while** loop *might never* perform its action if its boolean expression never has a true value.

This contrasts with the **repeat** loop, whose stopping mechanism comes *after* the statement-action. The **repeat** loop will *always* perform the specified action *at least one time* before it encounters the stopping mechanism.

```
program WhileDemo;
 var
  drWindow : rect;
 procedure drawScreen;
 begin
  hideAll;
  setRect(drWindow, 50, 50, 466, 326);
  setDrawingRect(drWindow);
  showDrawing
 end;
 procedure bop;
  var
   x, y, offset : integer;
   pt : point;
   r : rect;
  begin
   offset := 0;
   getMouse(x, y);
   setPt(pt, x, y);
   localToGlobal(pt);
   while ptInRect(pt, drWindow) and (offset < 75) do
    begin
     setRect(r, x - offset, y - offset, x + offset, y + offset);
     invertRect(r);
     offset := offset + 3
    end
  end;
begin
  drawScreen;
  while not button do
   bop
end.
```



with

■ Pascal reserved word

with works as a shortcut method of accessing the fields of a *record.* The format for the **with** statement is:

```
with recordName do
  begin
    fieldName1 statement;
    fieldName2 statement
  end;
```

The statements between **begin** and **end** can reference the fields of the record without needing to specify the recordName. The period notation (recordName.fieldName) is not necessary because the **with** statement automatically attaches the recordName before the fieldName.

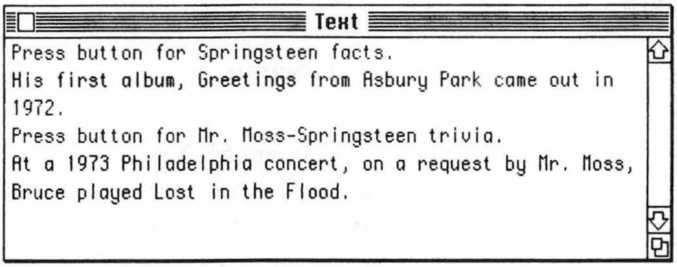
More than one recordName can be in the **with..do** statement by separating each with a comma: **with** recordName1, recordName2 **do**. This replaces the need for period notation (recordName1.recordName2.fieldName) in situations where records are nested within other records.

When more than one recordName is included in the **with** statement, the innermost, or latter recordName has precedence and is the first name attached to a field reference.

See **record** for information on records and their fields.

```
program WithDemo;
 type
  song = record
    name : string[50];
    time : integer
   end;
  album = record
    artist : string[50];
    title : string[50];
    year : integer;
    track : array[1..12] of song
   end;
 var
  Bruce1 : album;
 procedure init;
 begin
  with Bruce1 do
   begin
    artist := 'Bruce Springsteen';
    title := 'Greetings from Asbury Park';
    year := 1972;
    track[1].name := 'Spirit in the Night';
    track[1].time := 225;
    track[2].name := 'Lost in the Flood';
    track[2].time := 410
   end
 end;
 procedure display;
```

```
  begin
    writeln('Press button for Springsteen facts.');
    repeat    {wait until button is pressed}
    until button;
    writeln('His first album, "', Bruce1.title, '," came out in ', Bruce1.year, '.');
    writeln('Press button for Mr. Moss-Springsteen trivia.');
    while not button do    {alternative wait for button}
     ;
    write('At a 1973 Philadelphia concert, on a request by Mr. Moss, ');
    writeln('Bruce played "', Bruce1.track[2].name, '."')
  end;
begin
  init;
  display
end.
```



```
Press button for Springsteen facts.
His first album, Greetings from Asbury Park came out in
1972.
Press button for Mr. Moss-Springsteen trivia.
At a 1973 Philadelphia concert, on a request by Mr. Moss,
Bruce played Lost in the Flood.
```

write

Pascal procedure

The **write** procedure works similar to **writeln** except that the end-of-line character will not halt the procedure and advance to the file's next component.

See **writeln** for more information and an example.

writeDraw

Toolbox procedure

Use **writeDraw** to insert text into the Drawing window. **writeDraw** works like **write** except that no file parameter is allowed. The format for **writeDraw** is:

writeDraw(writeVariable);

If there is more than one parameter, they must be separated by commas. Variables must be one of the following types: *integer, real, char, string,* or *enumerated.* At least one parameter must be included.

The parameter of **writeDraw** can include *colon modifiers.* Colon modifiers specify, in order, minimum field width and number of decimal places. The format for using colon modifiers is the same as with the *writeln* procedure. See **writeln** for more information.

writeDraw inserts text at the current pen position, drawn in the pen's current size, mode, and pattern.

See **textFace** or **textFont** for an example.

writeln

■ Pascal procedure

Use **writeln** as a one-step process to insert a single component to a file, or in its simplest form, to display an output expression on the Macintosh screen. The format for **writeln** is:

```
writeln(fileName, componentVar);
```

If the file organizer fileName is omitted, the file device is assumed to be Macintosh screen. In this instance, the **writeln** output will be displayed on screen rather than inserted into a datafile.

The **writeln** procedure will insert the single component named in the second parameter into the file named in the first parameter.

The same process could be accomplished by the following statements:

```
fileName^ := componentVar;
put(fileName);
```

The file-type of the file organizer affects how **writeln** (and **readln**) operate. For non-textfiles, read and write values must be of type *char.* For textfiles, read and write values can be of other types, and their values will be translated to and from their character representations.

The output parameter (componentVar) of **writeln** can include *colon modifiers.* Colon modifiers are useful in formatting output such that it occupies a specific number of character spaces, and that real-type values are displayed in decimal point form rather than floating-point notation.

Colon modifiers specify, in order, minimum field width and number of decimal places. The format for using colon modifiers is:

writeln(outputExpression : fieldWidth : decimalPlaces)

Both fieldWidth and decimalPlaces must be integer values. A decimal place parameter cannot be used without a field width parameter, though fieldWidth can be used without decimalPlaces.

See **text** for an example of **writeln** with file parameters. See Chapter 25 in Part 2 for more information on colon modifiers.

```
program WritelnDemo;        {to create the dedication page}
 var
  i : integer;
  s : array[1..6] of string[80];
 procedure thanks;
 begin
  s[1] := 'Michael Cermak, who knows lots about friendship.';
  s[2] := 'Carol, Gary, Ruth, and Mason, who know lots about family.';
  s[3] := 'Amanda Hixson and Alan Goldstein, who know lots about books.';
  s[4] := 'Mr. Moss's girlfriend, who knows lots about Mr. Moss.';
  s[5] := 'And the person to whom this book is dedicated:  ';
  s[6] := 'Andy Hertzfeld, who knows lots about lots.'
 end;
begin
 thanks;
 writeln;
 write('Without the help of the following people, the Fear and ');
 write('Loathing Guide would have been entirely possible, but ');
 write('their generosity and talent allowed Mr. Moss ');
 write('to goof off, hang out, play with his girlfriend, and feel ');
 writeln('damn good about being alive.');
 writeln;
 for i := 1 to 6 do
  begin
   writeln(s[i]);
   if i = 4 then
    writeln
  end
end.                    {See dedication page for output}
```

INDEX

> $16.95 FPT USA

Macintosh Pascal Illustrated

"If coloring books had instructions, kids would just chew the pages." Thus begins the tirade of Mr. Moss, philosopher, programmer, and excitable boyfriend, in his crusade to strip the *boring evil* that plagues computer books.

You don't need any programming experience to use this book. By the time you have finished reading it, you'll know more than Pascal, the standard development language of the Apple Macintosh; you'll know how to enjoy programming. No flowcharts, no grammar lectures, no slobbering end-of-chapter exercises you would never do anyway. THE FEAR AND LOATHING GUIDE presents MacPascal as sharp, active, and ripe for improvisation.

MACINTOSH PASCAL ILLUSTRATED will carefully escort you through the graphic toolbook of MacPascal. From Mr. Moss you will discover why programming and being in love are the ideal complements, how to prevent such programming tragedies as premature compilation, and when a well-placed variable can produce satisfaction beyond your imagination. In addition, you'll:

- Build programs filled with the friendly Macintosh spirit
- Observe strange goings-on and perform amazing feats instantly in Macintosh windows
- Explore 100 complete, ready-to-run programs that demonstrate Quickdraw and the Toolbox in action
- Discover an illustrated dictionary that gives immediate access to the MacPascal facts you want to know

Learn to create files, play music to soothe your soul, make your printer perform like never before, and keep tabs with the calendar/clock. When your computer is on, your hands will always be on the mouse (and when your computer is off, Mr. Moss tells you where your hands ought to be).

> "It's funny. It's well-written. It will encourage people to play with their Macintoshes."
> *Andy Hertzfeld, Macintosh Software Creator*

> "A good telling of the facts of existence as we know them. And I like the programs too."
> *Bill Budge, Premier Software Artist*

ISBN 0-201-11675-8

Addison-Wesley Publishing Company, Inc.