Learn to Program
in
Pascal

by Mike Westerfield

Copyright 1990
Byte Works, Inc.

Table of Contents

Lesson One
Getting Started

Before We Get Started...

When I went to grade school, my teachers tried to beat some basic skills into my thick head. Back then, the basic skills included reading, writing, and arithmetic. When it came to spelling, my mind was already warped, because my teachers had also explained that these were the three R's.

Lately, in our rapidly changing world, we have added a new basic skill. It just isn't good enough to be able to read and write, plus do some math. In 1965, it wasn't easy to get from New York to Chicago without reading signs, writing instructions, counting some change and reading a clock. Today, you will use a computer to make the same trip. The travel agent will log your reservations in a computer. You may get spending money from a computer-based automatic teller. A digital watch counts bits to tell you what time it is. Computers control the flow of trains and the displays used by air traffic controllers. Your check book may even have a calculator. It's become a computerized world, and people who can't or won't deal with computers are rapidly being left as far behind as an illiterate person in the sixties.

Of course, you know all of that. That's why you have decided to learn to program. The purpose of this course is to teach you to program. By the end of the course, you will know one of today's most popular programming languages, Pascal. You will know it well enough to write programs of your own. Whether you want to plot an engineering equation, keep track of your Christmas mailing list, or write a computer game, this course will get you ready.

If you have been around computers for a long time, you may know that there are many languages you can use to write programs for your computer. It's fair to ask why this course uses Pascal.

One of the things you must look for in a computer language is that it must be fairly common. If a language is common, that tells you two important things: a lot of people think the language is a good one, and no matter what computer you decide to write a program for, you are likely to find the language you know. Today, there are four languages that fulfill this first requirement. They are C, Pascal, assembly language and BASIC.

If you decide to make your living programming a computer, you will eventually learn all of these languages. If you are learning to program, though, you have to pick just one of them to learn first. We can immediately rule out assembly language. In assembly language, you have to deal with the machine's internal structure. It takes many individual instructions to do the simplest thing. You will spend more time dealing with bits and bytes than learning how to write a well-organized program. We can also rule out BASIC. BASIC is a fine language in many respects, but it has become outdated by the rapid progress of programming practice. Modern programming often deals with pointers, linked lists, and dynamically allocated memory. The BASIC language can't deal with these ideas effectively.

That leaves C and Pascal. While either language is a good choice, Pascal is the better choice for a first language. The reason has to do with the way each language is structured. One of the design goals of the Pascal language was to create a language to teach programming to university level computer science students. It has all of the facilities needed to implement modern programs. It also has many built-in checks to help prevent programming errors. C was designed for professional programmers who implement programs that might need to do some very tricky things. Because of its built-in safety checks, the Pascal language often hinders their efforts. C, on the other hand, does not have these checks. That's good for the professional programmer, but bad for a beginner, who really needs those checks. Once you know more about programming, we can return to this topic and talk more about which language is best for a specific type of program. Rest assured that all of the programs in this book can easily be implemented in either language. In Pascal, though, your programs will not fail quite as often.

Before getting too much further, I also want to point out what this course is not. This is not a course about writing Apple IIGS desktop programs. I don't want to discourage you from writing desktop programs; quite the contrary. On the other hand, as you will find out, there is a lot to learn about programming before you are really ready to tackle something like a desktop program. By the time you finish this course, you will be ready to start to learn about desktop programming. If you tried to learn desktop programming right away,

though, you would probably fail. There's just too much to learn to try and do it all at once.

How to Learn to Program

Learning to program has a lot in common with learning to fly an airplane. When you learn to fly, most people start with an introductory flight with an instructor. Those that don't often make a bad first landing, and never get a second chance. (An old adage around flight schools is that any landing you walk away from was a good landing.) Before, during and after the flight, the instructor will tell you about some of the basics of flight: how the control surfaces work, what the controls do, and so forth. There will be a lot you don't know, and a lot of things you are told may not make sense right away. As you progress, you will spend time reading books and sitting in lectures, but you will also spend a lot of time actually flying the airplane. You wouldn't expect to spend all of your time reading books and sitting in lectures, then walk out to the plane and go off for a cross-country flight with no instructor; you gradually work up to that point. Eventually, though, you solo. You start to fly long distances, first with an instructor and then alone. Finally the day comes when you get your license.

It's the same way with programming. In a moment, we'll get started. We'll start off with a few simple programs. It is absolutely essential that you type them in and run them. There will be many problems that you can work on your own. The more problems you work, the better programmer you will become. Sure, we will spend some time talking about the ideas behind programming, and there will be some problems that you need to work through with a pencil and paper. For the most part, though, you will be programming; either typing in and analyzing programs with the help of this material, or writing and running your own programs. Gradually, the programs will get longer, and before long you will be able to write your own programs.

Just in case you missed the point, let me spell it out in very simple terms. If you read this material, but don't type in the sample programs or work the problems, you will know as much about programming as you would know about flying from reading a book. In short, very little. Programming is a skill. If you don't practice the skill, you will never learn it.

What You Need

Now is the time to sit down in front of your computer. Before starting, let's make sure you have everything you will need. First, you need an Apple IIGS computer. It must have a monitor; it really doesn't matter if it is black and white or color. The computer must have at least 1.125M of memory. For the older Apple IIGS that came with 256K on the mother board, this means that the memory card in the special memory slot must be populated with 1M of memory. In the most common case of an Apple memory card, this means that there should be a memory chip in each socket on the card. You can check this by taking the top off of your computer and looking. With the newer Apple IIGS, which comes with 1.125M of memory on the mother board, you don't need a memory card at all.

You must have at least one 3.5" disk drive. We will also assume that you have at least one other disk drive; it really doesn't matter if it is a 3.5" disk drive or a 5.25" disk drive.

You will need a copy of ORCA/Pascal. If you decide to use a different Pascal, there will be some things in this book that will not work. You would have to figure out why and make appropriate adjustments. By the time you finish this course, you will know enough to do that. At first, though, you may not. For that reason, I would suggest that you stick with ORCA/Pascal.

You will need at least four blank disks. This course is written with the assumption that you have two floppy disk drives. One of them must be a 3.5" floppy disk drive; you need that to run ORCA/Pascal. The other can be either a 3.5" floppy disk drive or a 5.25" floppy disk drive. Three of the floppy disks should be 3.5" disk drives; you will use these to make a copy of ORCA/Pascal. The fourth disk should be a 3.5" disk or a 5.25" disk, depending on what you are using for a second disk drive.

There are some other things that would be nice, but not essential. Most people like to print their programs and look at the paper copy. I highly recommend a printer if you intend to try this. With some of the longer programs we will write, more memory would be nice. With more memory, the process of translating your program from a text file into an executable program will go much faster. A hard disk is also very nice. Hard disks can hold much more than a floppy disk, so you will not have to switch disks as often. Hard disks are also faster than floppy disks, which again speeds up the programming process. Finally, an Applied Engineering TransWarp Accelerator card will roughly double the speed of your computer. As I said, all of these are nice. If you end up spending a lot of time programming, I would encourage developing a close relationship with St. Nicholas in an attempt to collect these items. You can, however, do everything in this course without them.

If you already have a hard disk, feel free to install ORCA/Pascal on your hard disk and work from there.

All of the things we will do in this course will work fine from a hard disk. You can find instructions on installing Pascal on a hard disk in the documentation that comes with the compiler. We will not cover it here.

Getting Everything Ready

When I bought my first FORTRAN compiler for the Apple II, I had a frightening experience. I wrote a program that crashed the compiler. The program actually erased some of the information on the compiler disk, so I could not use that disk anymore. In those days, many vendors still took the absurd position that compilers had to be copy protected. My local dealer either could not or would not help me restore the disk. I had one other copy (the program came with two copies), but I was afraid to use it.

Fortunately, times have changed. Most languages are no longer copy protected. The very first thing you should do when you open your copy of ORCA/Pascal is to make copies of each of the three floppy disks that come with the package. You can use the Finder to do this. If you know how to use some other copy program, and you like it better, go ahead and use it. Any copy program will work. Label each of the three disks you have copied, and put the originals in a safe place.

You will also need one other formatted disk to put your programs on. Go ahead and format that disk now. You can give the disk any name you like. Your name is one good choice. Keep in mind that this disk must be available at the same time as the ORCA/Pascal program disk. If you have one 3.5" disk drive and one 5.25" disk drive, your program disk must be a 5.25" disk. There will be plenty of room on the disk for a few dozen of the size programs you will write in this course.

One Disk Drive?

If you truly have only one disk drive, you will have to put your programs on the ORCA/Pascal program disk. As it is shipped, the ORCA/Pascal Desktop System disk, which is the one you will have in the disk drive while writing programs, has about 300K of free space. This is actually quite a lot of space for the size programs we will write in this course. At some point, the disk will be filled with your programs. When that happens, you can use the Finder to copy the files to another disk, or just make a second copy of the ORCA/Pascal Desktop System disk for your new programs.

Your First Flight... er, Program

It's time to take that first test flight. Strap yourself in. After all, as you have no doubt heard, computers can crash, so always wear your seat belt. Fortunately, though, a computer crash never hurts anything. As we go through this program, there will be a lot you don't understand yet. Be patient; in time, you will. The one thing you should keep in mind, though, is that you can't write a program that will damage the computer. Even if you do something wrong, the absolute worst thing that will happen is you will erase a disk – and even that is so unlikely that it isn't worth worrying about very much. It is, however, worth worrying about enough to make a copy of the ORCA/Pascal disks, which is why you should never run from the original disks.

We will use ORCA/Pascal exactly the way it comes out of the box. You will need two of the three disks for this program. Start by putting the disk labeled "ORCA/Pascal Boot Disk" into your boot drive, and starting your computer. After the program starts, you will be asked for the program disk. Eject the boot disk and replace it with the disk labeled "ORCA/Pascal Desktop System." After some more whirring, you will see a menu bar with four menus.

All of your programs will be written to a separate program disk. There is really nothing to stop you from putting your programs on the ORCA/Pascal program disk, and you may occasionally do that by accident, but you will run out of room on the disk if you do this on a regular basis. In short, it is time to put your blank disk in the second disk drive.

When you write a program, you type the program pretty much the same way you would type a letter in a word processor. Our first step, then, is to open a new Pascal program document. Pull down the File menu and select New. A window will appear, filling the desktop. You probably expected that, but what you may not have expected is the five new menus which appear on the menu bar. One of them is very important. ORCA/Pascal is one member of a large family of programming languages that all share the same programming environment. You must tell the system what language you want to use to write your program. To do this, pull down the Languages menu and select Pascal. If you pull it down again, you will see that Pascal has a check mark beside it.

Type in the following program. The format isn't terribly critical, but since you don't know what is and what is not important yet, it is best to type it exactly as shown. This program writes the characters "Hello, world." to the screen. It's a simple program, but we must start somewhere.

```
program Hello(output);

begin
writeln('Hello, world.');
end.
```

Setting the Language

If you forget to set the language, the message, "A compiler is not available for this language." If that happens, pull down the language menu and select the language. Make some change in the program - inserting and removing a space will do - and then save the program to disk again.

Once the program is typed in, it is time to save it to disk. This step isn't very important yet; it is very unlikely that these first few programs will crash the computer. If you form the habit of always saving the program, though, you will be very thankful the first time the computer crashes after you have spent several minutes typing. Once the computer crashes, it is easier to redo the typing than to recover the lost information – even assuming the information can be recovered. Sometimes it cannot.

To save the program, select Save As... from the File menu. A dialog will appear. Click on the disk button until the name of your blank disk shows up at the top of the dialog. If you are not sure what your program disk is named, watch the lights on the disk drive. When your disk is selected, the light on the disk will light up, and the disk will whir for a moment. Now type the name of the program file. In this case, you should type HELLO.PAS. Finally, click on the save button. Note that the name of the source window changed.

When ORCA/Pascal runs your program, it will

Desktop Editing

While you may be quite good at entering text with an editor, it is possible that this is your first look at a desktop editor. The new stuff that appeared on the screen is called a window. Along the top, you will see a black bar called the title bar. The box on the left hand side is called the close box. Clicking on the close box closes the window, removing it from the desktop.

The right-hand box on the title bar, called the zoom box, is used to change the size of the window back to a previous value. If you make the window smaller, then click on that box, the window will return to its original size.

In the middle of the title bar is the name of the disk file where the last permanent copy of the text was stored. If the window has never been saved, the name will be Untitled with a number.

At the bottom right-hand corner of the window, you will see another box; this one has two overlapping boxes inside. This is the grow box. The grow box lets you change the size of the window. To do this, you move the mouse to place the arrow cursor on the box, then press and hold down the mouse button. As you move the mouse, an outline showing where the new window will be moves across the screen. When the outline is in the correct spot, let up on the mouse button.

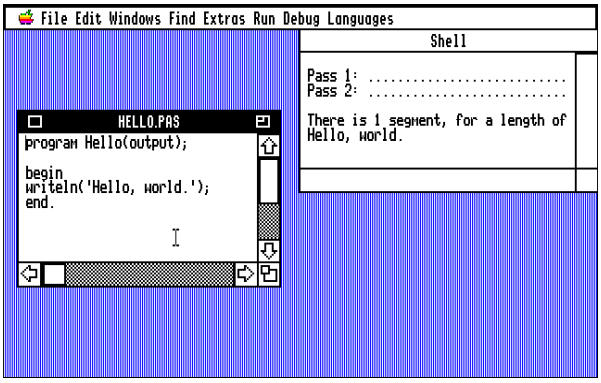
To move the window, you drag the title bar. Start by positioning the mouse cursor in the middle of the window's title bar, over the name of the window. Next, press and hold the mouse button, dragging the outline to the new location, and finally, let up on the mouse button.

As with any editor, you will often have more text in the file than can be displayed on the screen. The scroll bars on the right and bottom edge of the window let you move through the file. Clicking on one of the arrows "moves" the window over an imaginary, full size page that is 255 characters wide and as long as your program. (The window does not move; instead the text moves as if you moved the window over this larger, imaginary text page.) Clicking in the grey area beside the arrow moves the distance of a full window, rather than just one line or character. Finally, you can drag the white area, called the thumb, to move over very large distances. The relative position and size of the thumb in the scroll bar is proportional to the size and location of the window on the imaginary document that contains all of the text.

You can enter text just as you would with any editor. To correct a mistake, use the mouse to move the arrow mouse cursor over the window. The arrow will change to a vertical bar with a curly top and bottom. Position this bar where you want the flashing insertion-point to be and click on the mouse. You can now use the backspace key to delete old text, then type in the new text.

There are many short-cuts to using the desktop editor. We won't cover them in this course. At some point, you should spend some time reading the chapter in the ORCA/Pascal manual that describes the desktop editor to learn some of these shortcuts.

need someplace to write characters. The program will automatically open a special window called the shell window, and put the characters there. It will not, however, cover up your program with the shell window. To see what is happening, you need to shrink you program window to make room for the shell window. The shell window always appears on the top, right side of the desktop, so shrinking your window to half of its current width is a good idea. Do that now, putting the right hand side of your program window under the b in the Debug menu.



Well, the time has come to actually run your first Pascal program. Pull down the Debug menu and select Go. This tells the compiler to compile, link, and execute your program. The first thing you will see is the shell window. After some whirring of the ORCA/Pascal disk, you will see a message printed by the Pascal compiler as it compiles your program. When it finishes, it will tell you that no errors were found, and the linker will load and run. This prints a few messages, too. Finally, your program runs, and the characters "Hello, world." appear in the window.

It may seem like all of this took a long time, and it did. The first time you compile a program after booting the computer, several very large files are loaded from disk. The process is much faster the second time, since these files will stay in memory.

Go ahead – give yourself a cheap thrill. You've earned it. Select Go again, and run the program a second time. Maybe even a third!

The Program Window

If you forget to set the language, the message, "A compiler is not available for this language." If that happens, pull down the language menu and select the language. Make some change in the program - inserting and removing a space will do - and then save the program to disk again.

A Bug in Your Program?

If something else happened, check each step to see what you did wrong. One of the most common programmer mistakes is to assume that any mistake is the computer's fault. Sorry, it just ain't so. If things didn't work it's because, in order of likelihood:

1. You didn't do exactly what you were told.
2. You have set up a large RAM disk, tying up so much memory that ORCA/Pascal could not work. Get rid of the RAM disk. Be sure and power down and back up after setting the size of the RAM disk to zero.
3. You don't have the correct hardware.
4. You have a bad disk.
5. You have bad hardware.

Just for the record, you should know that absolutely every program we will show you in this course has been mechanically moved from the word processor to ORCA/Pascal and executed. They have also been typed in and executed by one or more guinea pigs we call beta testers, who we use to find errors before the errors can confuse you. In other words, every single program has been tested at least twice. If you encounter a problem, the chances are very, very small that the problem is in the program in the course material or in ORCA/Pascal.

To correct a problem, go back over each step in the text. If the program cannot be compiled, the compiler will give you an error message telling why. It also puts the flashing insertion-point near the place in the program where the error occurred. Check your typing in the area very carefully; a typing error is the most common cause of errors at this stage.

A Close Look at Hello World

Now that you have actually run a program, let's stop and spend some time talking about what happened. We'll start by examining the program in detail. The first step is to take a look at the words that make up the program

Like sentences in a book, programs are made up of a series of words and punctuation marks. Some of the words have special meaning, while some are words we pick to name parts of the program.

We'll dissect our first program to look at some of these rules. The program starts with a line that describes the program itself.

```
program Hello(output);

begin
writeln('Hello, world.');
end.
```

All Pascal programs start with the word `program`. This is followed by the name of the program; our program is called `Hello`.

The word `program` is called a reserved word. This just means that you can only use the word `program` in special ways in a Pascal program. For example, if you tried to name your program `Program`, the Pascal compiler would get confused, and complain. It would do this by bringing up an error message that says "identifier expected." The insertion-point would be placed on the word `Program`, showing just where the error occurred. If you like, you can try it to see what errors are like. It won't hurt anything. If you don't try it, don't worry – you will no doubt find out what errors are like soon enough.

Right after the name of the program is the word output, enclosed in parenthesis. This tells the Pascal compiler that you will be writing text to the text screen. It gives the compiler a warning, so it can set up things with the computer's operating system so text output will work. You don't have to put this in a program that does not write text to the screen. Many of the programs we write won't; instead, they will draw pictures on the graphics screen.

The program line ends with a semicolon. The semicolon character is used to separate lines in the Pascal language. Later, after you know more about Pascal, we will take a detailed look at how semicolons are used.

Right below the program line is a blank line. The blank line is for our convenience; the Pascal compiler really doesn't care if there is a blank line there or not. You can remove it, or put in several more, and the program will do exactly the same thing. We use the blank line to make the program easier to read. It's sort of like putting the chapter in books at the top of a new page. The book says the same thing if we don't, but it is easier to find the start of the chapter if we know it will be on a page by itself. You will find a lot of conventions like this described in the course.

The last three lines,

```
begin
writeln('Hello, world.');
end.
```

are collectively called the body of the program. The words `begin` and `end` mark the start and end of the body. Like the word `program`, these are reserved words, and can only be used in special ways in a Pascal program. All Pascal programs end with a period; it is placed right after the word `end`.

Between these words is the only line in the program that actually does anything, the writeln statement. Writeln is a built-in procedure in Pascal. That's a very fancy way of saying that all Pascal compilers know what writeln means. The characters that we want the program to write are placed after the word writeln. Pascal uses the quote character to mark the start and end of a string constant. This lets you write things like parenthesis, reserved words, and so forth, without confusing the compiler. As long as you keep the string on one line, you can put absolutely any characters you want in the string, except for the quote mark itself. You can still write a quote mark, but the compiler needs a special rule so it knows that the quote mark is supposed to be written, and doesn't signal the end of the string. The special rule is to put two quote marks together, like this:

```
writeln('Here''s how we write a quote mark.');
```

If you aren't positive what this will do, try replacing the writeln statement in your first program with this one, and run the program.

The string itself is surrounded by parenthesis. The entire line ends with a semicolon. Later, you will find out that this semicolon isn't really necessary in this particular case, but you will probably end up with fewer errors if you get in the habit of ending all of your lines with a semicolon. The program will work the same either way.

More About Reserved Words

In the last section, I pointed out that our first program had three reserved words: `program`, `begin` and `end`. These reserved words are three of the thirty-five reserved words in standard Pascal. Most Pascal compilers reserve a few more. ORCA/Pascal is no exception; it reserves seven more words. Here's a complete list of the reserved words:

Reserved Words in Standard Pascal

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

Additional Reserved Words in ORCA/Pascal

```
implementation          interface
otherwise   string      unit
univ        uses
```

Don't worry; you don't need to memorize the list. The important thing to remember is that there are some words you can only use in specific ways. If you get strange errors from the compiler, you can refer back to this table to see if the reason is misusing a reserved word.

Case Sensitivity

Pascal is case insensitive. That means that you can type the reserved words using lowercase characters, like I always do, uppercase letters, or any mix of case you prefer. Some people like using uppercase characters for reserved words, or perhaps capitalizing the first character of a reserved word. The compiler really doesn't care. Whatever you do, though, be consistent. That's our second style guide. (The first was to put a blank line after the program statement.) There are lots of right ways to write a program, but you should always pick one, and stick with it. You will get used to seeing programs that way, and it will make it easier for you to come back at a later point and read your program. Throughout the course, I will suggest style guidelines for you. They are the ones I like. If you want to use a different style, all I ask, for your own sake, is that you be consistent with the style you pick.

What is a Compiler?

In several places, I have mentioned a thing called a compiler. A compiler is a program that translates the program you write into something the machine can execute. You see, your Apple IIGS does not understand Pascal, or C, or BASIC, or even assembly language. All it understands are the individual bits stuffed into bytes that computer types call machine language. No one in his right mind actually programs in machine language if given a choice. Instead, they program in some language, like Pascal, then use a program called a compiler. The compiler reads the program you type, and figures out what the program is supposed to do. The compiler then writes a machine language program that does the same thing. It's a lot like taking a translator with you to Japan. You tell the translator that you desperately need to find a good Italian restaurant. The translator takes your words and reforms them in Japanese, asking the bell clerk at the hotel. In computer terms, the translator would be called a compiler. If you spoke German instead of English, you would need a different translator. The same is true with compilers. If you want to write the program in C instead of Pascal, you need a different compiler.

One of the advantages of a language like Pascal is that it does not depend on the computer you are using. On the Apple IIGS, you are using ORCA/Pascal, which translates Pascal programs into machine language programs the Apple IIGS can run. A Macintosh computer cannot run this program, but you could use MPW Pascal on the Macintosh. MPW Pascal can read the same program we have typed, and create a machine language program for the Macintosh. This program will do exactly the same thing on the Macintosh that it does on the Apple IIGS.

More About Identifiers

```
program Hello(output);

begin
writeln('Hello, world.');
end.
```

When we looked at the structure of our program, you found that you could pick any name you wanted to

for the name of the program, as long as you didn't use a reserved word. The name of the program is called an identifier. There are also two other identifiers in our first program; they are output and writeln. Each of these identifiers is used for a distinct purpose, but they all follow a simple set of rules. Identifiers start with an alphabetic character. They can be followed by any number of alphabetic characters and digits. The case of the characters does not matter, and you can even change the case within the same program. That's bad style, but again, the compiler does not care. Identifiers can be as long as you like, provided you put the identifier on one line and don't include anything in it except alphabetic characters and numeric digits.

Almost every Pascal implementation you will encounter allows you to use the underscore character in a variable name. ORCA/Pascal is no exception. The underscore character can be used at the start of a variable name, or imbedded within the variable. Standard Pascal, though, does not allow the underscore character in a variable name, so we will avoid them in this course.

Problem 1.1. Rewrite the hello world program so it says hello to you. For example, my name is Mike, so I rewrote the program to say "Hello, Mike." Save this program as NAME.PAS.

Note: When you name a program, always choose ten or fewer characters that start with a letter and contain only letters and spaces. Append a .PAS onto the end.

Problem 1.2. What is the shortest legal Pascal program? How many lines does it require? If the program name is Short, you should be able to write the program with twenty-four keystrokes. Give it a try, then save your program as SHORT.PAS, and see if the compiler agrees with you.

How Programs Execute

With what we know now, we can start to write larger programs. Our first step will be to modify the hello, world program to write five lines instead of one.

Type in the program in Listing 1.1 and save it on your program disk as LIMERICK.PAS. Use the Go command in the Debug menu to run the program.

Did the program do what you expected? It does bring up an obvious point. Like sentences in a book, the compiler reads and processes your program in the order it is written. The first line is executed first, the second is executed second, and so on.

We will introduce a new capability to see this a bit better. Pull down the Debug menu again, but this time, choose the Step command. Instead of executing, everything stops. The only menu that is not highlighted is the Debug menu, so that is the only one that you can select anything from. You may have noticed the menu bar flashing through this state before and wondered why. This indicates that your program is still running. You can't use anything but the Debug menu until the program is finished.

Now look at your program window. There is an arrow pointing to the first line of the program. This arrow shows the next line to be executed. The program is waiting for you to tell it to go ahead. Pull down the debug menu again, and select Step a second time. This tells the program to step to the next line. The arrow moves down, and the first line of the limerick appears in the shell window. The keyboard equivalent for Step is ⌘[. Use that keystroke to step through the rest of the program.

What you have just used is called a source-level debugger. It lets you step through a program one line at a time to see what the program is actually doing. You see, computers have one very bad habit. They do what you tell them to, rather than what you want them to. When your program does something different than you

Listing 1.1

```
program Limerick(output);

begin
writeln('There was an old man with a beard');
writeln('Who said, "It is just as I feared!"');
writeln('    Two Owls and a Hen,');
writeln('    Four Larks and a Wren,');
writeln('Have all built their nests in my beard.');
end.
```

expect, the source level debugger can often help you find out what the computer is actually doing. Once you know that, it is usually easy to correct the program, so the computer does what you want it to. As you learn more about Pascal, you will also learn more about the debugger. For now, it would be a good idea to use the debugger to step through each of your programs, so you can see what the computer is actually doing.

Problem 1.3. Write a program that prints your name and address. Print the address on separate lines, just as you would on an envelope.

Problem 1.4. With a little work, you can create a readable letter by coloring in squares on a sheet of graph paper. The smallest number of squares that works well for uppercase only letters is seven high by five wide. This is the idea used to form characters on the computer screen from the small dots called pixels.

Write a program to write your first name to the screen in this form. Use the * character to fill in the squares. For example, I would ask the computer to write this to the screen:

```
*    *   ***   *    * *****
** **    *    *   *   *
* * *    *    * *    *
*   *    *    **     ***
*   *    *    * *    *
*   *    *    *  *   *
*   *   ***   *   *  *****
```

Graphics Programs

There's a lot you can do with text, but the Apple IIGS has some stunning graphics, too. It's time to start using some of that power. One word of caution, though: like most computer languages, Pascal does not have built-in graphics. The information in this section that deals with graphics is particular to the Apple IIGS. Other computers may do things a bit differently.

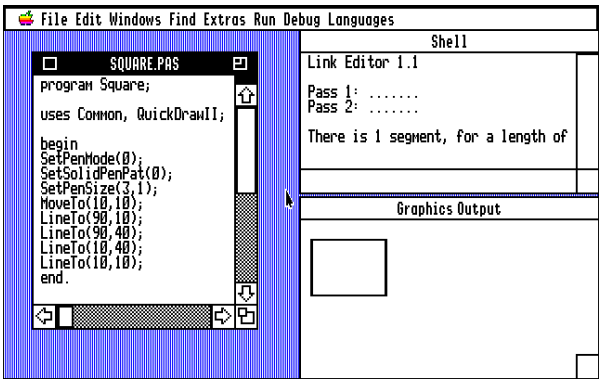
The Apple IIGS has a large number of built-in subroutines to do complicated tasks for you. These subroutines are called tools. They are grouped by function into groups called tool sets. The entire collection is what people refer to as the toolbox. The toolbox is a large and wonderful collection which we won't have time to explore fully, but we will use some of the tools to do some work for us from time to time. Graphics is one of those times. We will be using a tool set called QuickDraw II, which is the graphics tool set on the Apple IIGS. QuickDraw II is a powerful collection of low-level graphics routines. The following program will be our first venture into graphics.

```
program Square;

uses Common, QuickDrawII;

begin
SetPenMode(0);
SetSolidPenPat(0);
SetPenSize(3,1);
MoveTo(10,10);
LineTo(90,10);
LineTo(90,40);
LineTo(10,40);
LineTo(10,10);
end.
```

Type in this program, and save it as SQAURE.PAS, but don't run it yet. Text programs write characters to the shell window, as you have already learned. The shell window, though, is for text. Graphics output is written to a special graphics window. Before running your program, you must open the graphics window. Do this now by pulling down the Run menu and selecting the Graphics Window command. If you need to, resize your program window so you can see the entire graphics window. Now run the program. You will see a square, about one inch high and once inch wide, on your screen. (Depending on the monitor you are using and how it is adjusted, the size of the square may vary a bit.)



One difference from previous programs is that the program statement doesn't have the output file after the program name. Putting it there won't hurt anything, but since we aren't doing any text output, we can leave it out. Leaving out stuff that isn't needed is always a good idea when you program. In this case it doesn't matter, but leaving out things that aren't needed often makes a program smaller and causes it to run faster.

Right after the program statement is something new, a uses statement. This is followed by two names, separated by a comma. This statement tells the compiler to load and process two things called interface files. Interface files tell the compiler about the subroutines, constants and data types in the QuickDraw II toolkit, and some common definitions used by all of the toolkits. The files are quite long; you may have noticed that it took longer to compile this program than it took to compile the earlier programs. Almost all of the difference is due to the interface files.

The first three lines in the program body tell QuickDraw II how you want to draw lines. SetPenMode(0) tells QuickDraw II to replace any existing dots with new dots. That makes sense, so you might wonder why you need to bother. QuickDraw II can do other things when it draws, so we need to start by telling it to do the simplest of the alternatives. The next line, SetPenSize(3,1), tells QuickDraw that lines are three dots wide and one dot high. You can pick other widths and heights for the line. Since each dot on the graphics screen is about three times as high as it is wide, the choice in the example gives lines that are about the same thickness, whether they are horizontal or vertical. Try some other values to see what they look like. Finally, SetSolidPenPat(0) tells QuickDraw II to draw black lines.

The next five lines draw a square in the graphics window. To understand how they work, we need to start by examining the coordinate system used by QuickDraw II. To QuickDraw, the top left dot in any window is at 0,0. As you move to the right, the first number increases. In other words, 90,0 is 90 dots to the right of 0,0, but on the same line. As you move down, the second number increases. The point 0,40 is 40 dots below 0,0. You can use numbers so large they go outside of the graphics window. In that case, you can't see the lines, but QuickDraw II will still draw all of the line that is in the window. Give this a try by extending the square so the lower-right corner is at 500,500, rather than 90,40.

Incidentally, if you didn't try stepping through the program with the debugger, I highly recommend doing that now.

Problem 1.5. In the 640 mode that ORCA/Pascal runs in, there are a total of four colors that you can use. The SetSolidPenPat call is used to choose from these colors. In our example, we used color 0 to draw the square in black. The other three colors are 1, 2 and 3. Try these. What happens when the pen color is set to 3?

Problem 1.6. An equilateral triangle is a triangle where each of the three sides are the same length. Write a new program to draw an equilateral triangle with 1 inch sides in the graphics window. Make the bottom flat, with one point on the top.

Problem 1.7. Modify the program in problem 1.7 to draw a six sided star by drawing two equilateral triangles, one pointed up and one pointed down, and overlapping the triangles. Make the star green, and use thick lines.

Problem 1.8. Write your name in the graphics window by drawing lines. If your name has letters with curves, use a few short lines to approximate the shape of the letter.

Lesson One
Solutions to Problems

Solution to problem 1.1.

```
program HiMike(output);

begin
writeln('Hello, Mike.');
end.
```

Solution to problem 1.2.

```
program Short;begin end.
```

Solution to problem 1.3.

```
program Address(output);

begin
writeln('Mike Westerfield');
writeln('4700 Irving Blvd. NW, Suite 207');
writeln('Albuquerque, NM  87114');
end.
```

Solution to problem 1.4.

```
program Name(output);

begin
writeln('*   *  ***  *   * *****');
writeln('** **   *   *  *  *    ');
writeln('* * *   *   * *   *    ');
writeln('*   *   *   **    ***  ');
writeln('*   *   *   * *   *    ');
writeln('*   *   *   *  *  *    ');
writeln('*   *  ***  *   * *****');
end.
```

Solution to problem 1.5.

The four colors available in 640 mode without changing the color palettes are:

```
0  black
1  purple
2  light green
3  white
```

When the pen color is set to 3 (white), the program still draws a square, but the square is the same color as the background of the window, and does not show up.

```
program PurpleSquare;

uses Common, QuickDrawII;

begin
SetPenMode(0);
SetSolidPenPat(1);
SetPenSize(3,1);
MoveTo(10,10);
LineTo(90,10);
LineTo(90,40);
LineTo(10,40);
LineTo(10,10);
end.
```

```
program GreenSquare;

uses Common, QuickDrawII;

begin
SetPenMode(0);
SetSolidPenPat(2);
SetPenSize(3,1);
MoveTo(10,10);
LineTo(90,10);
LineTo(90,40);
LineTo(10,40);
LineTo(10,10);
end.
```

```
program WhiteSquare;

uses Common, QuickDrawII;

begin
SetPenMode(0);
SetSolidPenPat(3);
SetPenSize(3,1);
MoveTo(10,10);
LineTo(90,10);
LineTo(90,40);
LineTo(10,40);
LineTo(10,10);
end.
```

Solution to problem 1.6.

```
program Triangle;

uses Common, QuickDrawII;

begin
SetPenMode(0);
SetSolidPenPat(0);
SetPenSize(3,1);
MoveTo(110,50);
LineTo(190,50);
LineTo(150,24);
LineTo(110,50);
end.
```

Solution to problem 1.7.

```
program Star;

uses Common, QuickDrawII;

begin
SetPenMode(0);
SetSolidPenPat(0);
SetPenSize(3,1);
MoveTo(110,50);
LineTo(190,50);
LineTo(150,24);
LineTo(110,50);
MoveTo(110,33);
LineTo(190,33);
LineTo(150,59);
LineTo(110,33);
end.
```

Solution to problem 1.8.

```
program DrawMike;

uses Common, QuickDrawII;

begin
SetPenMode(0);
SetSolidPenPat(0);
SetPenSize(3,1);

MoveTo(20,70);
LineTo(20,20);
LineTo(45,45);
LineTo(70,20);
LineTo(70,70);

MoveTo(90,20);
LineTo(100,20);
MoveTo(90,70);
LineTo(100,70);
MoveTo(95,20);
LineTo(95,70);

MoveTo(120,20);
LineTo(120,70);
MoveTo(170,20);
LineTo(120,45);
LineTo(170,70);

MoveTo(240,20);
LineTo(190,20);
LineTo(190,70);
LineTo(240,70);
MoveTo(190,45);
LineTo(220,45);
end.
```

Lesson Two
Variables and Loops

Integer Variables

You have probably heard that computers are very good at dealing with numbers. This is quite true. In this lesson, we will start to use numbers and variables in our programs. If you aren't a math whiz, though, don't panic. We won't be dealing with anything more complicated than simple arithmetic in this chapter. Let's start by typing in the program in Listing 2.1.

One of the first things you will see in our program is a comment. Comments start with a { character and end with a } character. You can type anything you want except the } character or the pair of characters *) inside of a comment. The compiler ignores them completely. You can always replace a comment with a space, and the compiler will produce exactly the same program as it did when the comment was there. Why, then, do we bother?

If your memory was as good as the computer's, and if no one else ever read your programs, you wouldn't need comments. Comments are for your benefit, as well as the benefit of all those poor lost souls who will have to figure out what you did later. There are two places where you should put a comment in every program you write. The first is at the beginning of the program, identifying quickly what the program is for. It's not a bad idea to put your name and the date the program was written there, too. Finally, you will notice that I put a comment after the variables to tell what they are for. This, too, is a very good habit to form.

Computers can work with a vast array of number formats, each of which has a special purpose. The two most common number formats are integers and reals.

Listing 2.1

```
{ This program prints a table of numbers and squares of the numbers }

program Squares(output);

var
   i,s: integer;               {i is a number, s is its square}

begin
i := 1;
s := i*i;
writeln(i:10, s:10);
i := i+1;
s := i*i;
writeln(i:10, s:10);
i := i+1;
s := i*i;
writeln(i:10, s:10);
i := i+1;
s := i*i;
writeln(i:10, s:10);
i := i+1;
s := i*i;
writeln(i:10, s:10);
end.
```

Integers are whole numbers, like 4, -100, or 1989. Real numbers include the numbers between the whole numbers, like 1.25 or 3.14159.

The memory of a computer is made up of a vast series of numbers, but in a language like Pascal, we don't have to deal with them the same primitive way the computer does. Instead, we can define variables. A variable is just a place where you can put a numeric value. In our program, we define two integer variables called i and s. Within certain limits, we can put any integer number we like in these variables. It's exactly like putting two names for numbers on a sheet of paper and continuously erasing the number to replace it with a new one.

In Pascal, all variables are grouped in a special part of the program that comes between the program statement and the begin statement. You start with the keyword var. To make the program easier to read, it is a good idea to skip to a new line to actually define the variable, as shown in the sample program. You then name the variables you want to define, separating them with commas if there are more than one. This is followed by a colon, and the type of the variable. In this program, we are using integers, so the type is integer.

These variables are put to use in the program body. The first thing we need to do is learn to put a number in a variable. We do this with something called an assignment statement. The line

```
i := 1;
```

tells the computer to place the number 1 in the variable i. The := characters are called the assignment operator. When Pascal types read this line, they will say "i gets 1." The very next line puts this value to use.

```
s := i*i;
```

Here, we multiply i by itself and put the result in the second variable, s. The * character is used in computer languages for multiplication because a computer would confuse x in "i x i" with a variable named x. The result is saved in the location named s. Finally, we write the values.

```
writeln(i:10, s:10);
```

The writeln statement deserves a little more attention, since there are several new concepts here. We have already used the writeln statement to write characters to the shell window. In this case, though, we are writing two numbers. Any time we use the write statement to write two things, the two things are separated by a comma. We can also mix strings and numbers in the same writeln statement.

In this writeln call, each of the numbers is followed by a field width specifier. That's just a simple way to put the numbers in columns. If you tried

```
writeln(i:1, s:1);
```

the program would still work, but the two numbers would be shoved together, with no space in between. Leavingspacesoutmakesthingshardtoread, so we put the the numbers in columns that are ten characters wide. If we don't put any number for the field width, Pascal assumes a field width of 8 characters. The field width assumed by Pascal varies from one compiler to another, though.

You can probably figure out what the rest of the program does on your own, but let's learn to use a new

Using Cut and Paste

If you look closely at the sample program, you will see that many of the lines are repeated more than one time. You can use a technique called cutting and pasting to make it easier to type in the program.

Start by typing the program through the first writeln statement. Now move the mouse so the cursor is just to the left of the statement

```
i := 1;
```

Press the mouse button. The entire line will switch to white letters on a black background. Holding the mouse button down, drag the mouse down until the three lines ending with the writeln statement are shown in inverse (white letters against a black background), then let up on the mouse. The inverted lines are now selected. One of the things you can do with selected text is to copy the text into an internal buffer called the scrap buffer. Do that now by pulling down the Edit menu and selecting Copy.

The next step is to move the insertion point to the end of the program. Move the mouse to the start of the first blank line after the writeln statement and click. The selected text will go away, and the familiar blinking insertion point will appear, ready to type a new line. This time, though, pull down the Edit menu and select Paste. The lines you copied into the scrap buffer are written into the program file.

debugger tool. Later, when our programs are much more complicated, knowing how to use the debugging tools will be much more important.

Be sure you typed the program in properly by running it one time. Now pull down the debug menu and select step, but this time, don't step through the program right away. Instead, pull down the debug menu again and select Variables Window. A new window will show up on the desktop. This window is used to look at the values stored in variables while the program is running. You will be using it a lot to see how programs work and to find out why your own programs fail.

Click in the body of the new window; you will get a line edit box. Type the name of the variable i and hit return. Click below this variable, and add s to the list. Now step through your program in the normal way. As you go, the values of the variables are updated, one line at a time.

Problem 2.1: The Fibonacci series is a sequence of numbers obtained by adding the two previous numbers in the series. The series starts with 0 and 1. Write a program with three integer variables named last, current, and next. Set last to 0 and current to 1.

Now do the following steps five times:

1. Compute next by adding current to last and saving the result in next.
2. Print next.
3. Assign current to last.
4. Assign next to current.

The result should be the numbers 1, 2, 3, 5 and 8, all on a different line. Be sure to run your program through the debugger. It is important that you understand how we are using sequential execution and variables to gradually step through the sequence of Fibonacci numbers.

Fibonacci numbers seem to occur frequently in nature; no one is quite sure why. The number of petals in a flower and the number or leaflets on a compound leaf are often Fibonacci numbers.

The For Loop

So far, all of our programs have executed one statement at a time, starting with the first and proceeding to the last. In our last sample and problem, this started to get a little tedious, as we repeated the same thing over and over, incrementing a number by one each time. Computers are real good at doing tedious things, but most people are not. The for loop is the first in a series of statements we will look at that help remove some of the tediousness of writing a program.

Type in the sample program in Listing 2.2 and run

Listing 2.2

```
{ Draw a fan shape in the graphics window }

program Fan;

uses Common, QuickDrawII;

var
   i: integer;                          {loop variable}

begin
SetPenMode(0);                          {set up for graphics}
SetSolidPenPat(2);
SetPenSize(2,1);
for i := 1 to 25 do begin               {draw the fan}
   MoveTo(160, 70);
   LineTo(i*12-10, 10);
   end; {for}
end.
```

it. Before you read further, take a crack at figuring out what it is doing on your own. Be sure and use the debugger. Also, since this program draws in the graphics window, be sure you open the graphics window and resize all of your windows so you can see it before running the program.

Most of the things in this program should be familiar by now, although some of them are being used in new ways. There are three new concepts to master, though. The first is the idea of the for loop itself. In Pascal, we use a for loop whenever we need to do something a specific number of times. This could be calculating ten values, or drawing twenty-five vanes of a fan, as our program does.

The for loop starts with the reserved word `for`, followed by an assignment. In our case, we are starting our for loop with i set to 1. The two statements right after the for loop get executed once with i set to 1. What happens is exactly the same as if we substitute 1 for i in the statements, like this:

```
MoveTo(160, 70);
LineTo(1*12-10, 10);
```

It doesn't stop there, though. After executing these statements, i is set to two, and the statements are executed again. This continues until i is twenty-five. After executing the statements one last time with i set to twenty-five, the compiler moves on to the line after the for loop.

If you look up the for loop in a Pascal reference book, you will find out that it executes one statement, then loops. In our sample, though, we wanted to execute two statements, MoveTo and LineTo. To do this, we have to have a friendly talk with the compiler about what a single statement is. We use the familiar reserved words, begin and end, around the two statements. This tells the compiler to treat the group of lines between the begin and end as a single statement, so both the MoveTo and the LineTo get executed as part of the loop. To see the difference, try removing the begin and end, like this:

```
for i := 1 to 25 do
   {draw the fan}
   MoveTo(160, 70);
   LineTo(i*12-10, 10);
```

Use the debugger to trace through the program both ways. Watch the arrow as the program gets executed. As you can plainly see, the second line is not executed as part of the loop unless you use the begin and end. That's one advantage of the debugger, and a good reason to use it on all of your programs: you can plainly see what happens. When a program doesn't work, it isn't always easy to tell why by looking at the source code. The debugger can show you what the computer is actually doing, rather than what you think the computer is doing.

Technically, a series of statements grouped together by a begin and end is called a compound statement. Whether you remember that or not won't make you a better or worse programmer, but when you read books about Pascal, the authors will talk about compound statements instead of continually saying "those statements grouped together with the reserved words begin and end." Saying it's a compound statement just takes a little less room. It also makes you sound like you know something, impressing the natives who can't program.

Problem 2.2: Our first sample in this chapter created a table of numbers and squares. It did this in a fairly clumsy way, by using separate statements to step from 1 to 5. Rewrite this sample using a for loop.

Problem 2.3: In the last chapter, we drew a square by drawing its sides with constant integers. We could also draw the rectangle using variables, like this:

```
top := 10;
bottom := 70;
left := 10;
right := 100;
MoveTo(left, top);
LineTo(right, top);
LineTo(right, bottom);
LineTo(left, bottom);
LineTo(left, top);
```

Use a for loop to draw five rectangles, one inside the other. Set top, bottom, left and right before the for loop starts. Inside the for loop, draw the rectangle, then add six to top and left, and subtract six from bottom and right.

Indenting: Programmers Do It With Style

In the last section, our for loop looked like this:

```
for i := 1 to 25 do begin
   {draw the fan}
   MoveTo(160, 70);
   LineTo(i*12-10, 10);
   end; {for}
```

You no doubt noticed that we moved over three spaces on each line that is in the for loop. This is called indenting. The compiler really doesn't care if you indent or not. As far as the compiler is concerned, this line does exactly the same thing:

```
   for i := 1 to 25 do begin
MoveTo(160, 70); LineTo(i*12-10, 10);
end;
```

Most folks, though, find it easier to read the first loop.

You also didn't need to indent at all. It is almost as easy to read

```
   for i := 1 to 25 do begin
   {draw the fan}
   MoveTo(160, 70);
   LineTo(i*12-10, 10);
   end; {for}
```

as it was to read the first loop. The reason we indent is to make it easy to see which lines are in the loop. This makes it easier to read the program and tell what it is doing. On the other hand, you should never depend on indenting. When I removed the begin and end statement to show you that the for loop only executes one statement at a time, and how the begin and end statements convince the compiler to treat two lines as a single statement, I wrote

```
   for i := 1 to 25 do
      {draw the fan}
      MoveTo(160, 70);
      LineTo(i*12-10, 10);
```

From the indenting, you would expect the LineTo command to be part of the loop. The compiler doesn't care about indenting, though. In short, if your mommy was a programmer, she would have told you never to depend on comments or indenting when you are trying to find a problem in a program. The comments or the indenting may be wrong.

In the first lesson, I made the outrageous claim that there is more than one right way to do almost anything in programming. Indenting is one of those places where this is particularly true. I always use three spaces when I indent. I always put the begin on the same line as the for statement. Here, though, are some alternate indenting styles. All of them work, and all serve the same purpose. I like mine because it presents more information in fewer lines using fewer columns than the others do, but other people defend their style with vigor, too. The long and short of it is that it really doesn't matter what style you use, as long as you find one you like and use it consistently.

```
   for i := 1 to 25 do
      begin
      MoveTo(50, 70);
      LineTo(i*4-50, 10);
      end;

   for i := 1 to 25 do
      begin
         MoveTo(50, 70);
         LineTo(i*4-50, 10);
      end;

   for i := 1 to 25 do
     begin
       MoveTo(50, 70);
       LineTo(i*4-50, 10);
     end;
```

Some Thoughts on Comments

You may notice more and more comments slipping into our programs. As the programs get longer and more complicated, you will see the trend continue.

One habit that has saved me a lot of grief over the years is documenting what I am ending. Returning to that darned for loop one last time, notice the comment after the end statement.

```
   for i := 1 to 25 do begin
      {draw the fan}
      MoveTo(50, 70);
      LineTo(i*4-50, 10);
      end; {for}
```

This comment simply tells me that I am ending a for statement, as opposed to ending the program or some other compound statement. In this program, it doesn't make much difference, but as your programs get bigger you will find that it helps. Get into the habit now, and it will save you a lot of grief later.

Another new use of comments is to label parts of the program. For example, the for loop is labeled with a comment that says the loop draws the fan. These are a great help. You can read the statements by now, and you know what each one does. No one has to tell you what MoveTo(50, 70) does, for example. On the other hand, it is certainly not obvious to me that these lines of code draw a fan shape. The comment tells me that, and suddenly the purpose behind the statements is clear. By

putting these comments on the right side of the page, you can quickly scan through the program, looking for a particular section.

Operator Precedence

By now, you are getting used to the idea that computers step through a program in a fairly orderly way. Statements are executed top to bottom, left to right, the same way you read. Try the following program, but see if you can figure out what will be printed before you run the program.

```
{ A look at operator precedence }

program Precedence(output);

begin
writeln(1+2*3);
end.
```

There are two perfectly reasonable ways to compute a value from the expression

```
1+2*3
```

The first is to work left-to-right:

```
1+2*3
3*3
9
```

The second is to follow the rules you may remember from algebra class, and do the multiply first.

```
1+2*3
1+6
7
```

As you can see from running the program, Pascal uses the same rules as algebra teachers. Pascal was, after all, designed by a college professor who took a lot of math courses in his day. Not all languages follow these rules; APL, for example, does work left to right. The way a language determines what order to do operations in is called operator precedence. We might as well call it the operator pecking order; it means the same thing. Computer types like to sound official, though, so we better stick to precedence.

The table below shows all of the operators in Pascal. All of the operators on the same line have the same precedence. The ones at the top are done first. If two operators with the same precedence appear together, they are evaluated left-to-right.

You will learn to use most of these operators as the course continues. For now, the important thing is to remember that this table exists. You will need to refer back to it many times.

```
not    ~     **    @     *     /
&      <<    >>    div   mod   and
+      -     |     !     or    =
<=     >=    <     >     <>    in
```

Note: The operators `~ ** @ & << >> |` and `!` are extensions to standard Pascal.

In our original expression, if you really wanted to compute the value 9, you could have used parentheses. Pascal does all operations inside of parentheses as a group, and uses the result in the rest of the expression.

```
(1+2)*3
3*3
9
```

The Maximum Integer

Growing up with a last name like Westerfield, I quickly learned that computers had limits. It seemed like all of the people who programmed had names like Wirth, or Ritchie, or Steele. All of those silly forms that asked me to put each letter into a separate block had ten blocks. It upset me: my name wasn't Westerfiel, it was Westerfield. The protests of a seven year old are seldom heeded, though.

Computers have become a lot more friendly since then, perhaps in part due to the fellow protests of people like Joe Jabinoslowski. But they still have fixed limits on just about everything. The limit may be very large, but it is there. Integers are no exception. Every Pascal compiler imposes some upper limit on integers – some largest number that can be stored in an integer variable. On most microcomputer based Pascals, this value is... but wait, there is a better way to find out.

```
{ Find the largest integer }

program Big(output);

begin
writeln(maxint);
end.
```

Maxint is a predefined constant that is present in all Pascal compilers. It tells you the largest number you can save in an integer variable. Running this program, you find out that the largest integer is 32767. There is a good reason for that. It has to do with the way numbers are stored in a computer. We really don't need to delve into that at the moment, though. The important thing is that you know that there is a maximum.

There is a minimum integer, too. In Pascal, it is always -maxint.

Pascal books will tell you that the "result is undefined" if you do something that will create a number too big to store in an integer and try to store it there. That's a polite way to say that the compiler can do whatever it feels like. That makes it tough for your program to work well. In short, don't use numbers that big.

So how can you tell what will happen? Well, lets try it.

```
{ Overflowing an Integer }

program Error(output);

var
   i: integer;

begin
i := 500;
writeln(i*i);
end.
```

The correct answer is 250000, of course, but that is bigger than 32767, so the value cannot be represented as an integer. The fact that the value was not stored does not matter. Since both of the numbers being multiplied were integers, the result of the multiply must fit in an integer space.

Full implementations of Pascal generally have a safety net that you can use to find errors like this. In ORCA/Pascal, you turn on this safety check by putting the line

```
{$rangecheck+}
```

at the beginning of your program. This is a special kind of comment called a compiler directive. A compiler directive is a special instruction that tells the compiler how you want the program compiled. All compiler directives start with a $ right after the opening { character. This one turns on range checking, which tells the compiler to check all math operations (and several other things you don't know about yet) to make sure the results are legal. If not, the program stops with an error message. You can step through with the debugger to find out where the error actually occurred, and fix it.

One disadvantage of range checking is that the compiler must create a longer, slower program to do all of the necessary checks. You should only use range checking while you are debugging a program. Once the program is finished, turn range checking off by removing the directive from the program.

Many Pascal compilers have a special kind of integer that can hold numbers bigger than maxint. In ORCA/Pascal, the type of these integers is longint. We can get the correct answer from our program using these longer integers.

```
{ Long Integers }

program Long(output);

var
   i: longint;

begin
i := 500;
writeln(i*i);
end.
```

Long integers can hold numbers as small as -2147483647 and as large as 2147483647. While some form of long integer is a common extension to the Pascal language, it is not something that is found in all Pascal compilers, so we won't use them much in this course.

Real Numbers

As everyone knows, programmers drive Porches. At least, many of the folks I meet seem to have that impression. I have never met a programmer that drove a Porche myself. Still, you may be aspiring to high goals, so let's see how long you will be paying off your dream car. We will assume that you want a new car, but not necessarily a fancy one. We'll spend $24,000 on our car. We'll assume that you know a banker real well, and can get your car loan at 10% APR, which works out to a monthly interest rate of about 0.83%. That would make the initial interest payment for the first month

```
24000*0.0083
$199.20
```

Let's assume you are generous and want to pay $250 a month. The program below finds out how many months you will be paying.

The negative number after the last payment shows that you didn't quite have to pay $250.00 the last month to pay off the loan. The number of months this takes shows why I own a Datsun. An old one.

This program builds on your previous knowledge, but it also introduces a wealth of new ideas. The first is a brand new section after the program statement where constants are defined.

```
const
   cost = 24000.0;  {initial cost of car}
   APR = 10.0;   {annual percentage rate}
   payment = 250.0;     {monthly payment}
```

Like the variables section, the constant section starts with a reserved word. In this case it is const. This helps you to see where the sections start, and helps the compiler do the same. The constant section must come before the variable section.

Constants are similar to variables in some respects. Like variables, constants have types. While our sample only defined real variables, it is possible to define integer variables, too. Also like variables, constants give us a way to associate a name with a number. The big difference is that you cannot change a constant. A constant has the same value for the entire time the program runs.

Constants have several uses. Like comments, they make it easier to read, understand, and change a program. For example, it is very easy to change the interest rate in this program. It is also very easy to change the price of the car and the size of the payment. It also gives you a chance to label a number. For example, it is easy to see that 250.0 is the monthly payment on the car, but it isn't as easy to see what is happening in the program in listing 2.4.

The program does exactly the same thing, but the first is easier to change and understand. Finally, if a number is used more than one time in the program, using a constant lets us change it in one place, rather than searching through the entire program for all of the places the number is used.

The difference between real numbers and integers is that real numbers can have values other than whole numbers, but they don't have to. In our constant

Listing 2.3

```
{ Why I don't own a Porche }

program IWannaPorche(output);

const
   cost = 24000.0;               {initial cost of car}
   APR = 10.0;                   {annual percentage rate}
   payment = 250.0;              {monthly payment}

var
   month: integer;               {number of months}
   principal: real;              {amount left to pay}

begin
month := 0;                      {no payments made, yet}
principal := cost;               {we start owing this much}
while principal > 0.0 do begin   {keep going until we're out of debt}
   month := month+1;             {it's a new month}
   principal := principal        {add in this month's interest}
      + principal*APR/100.0/12.0;
   principal := principal-payment; {make the payment}
   writeln(month:4, principal:10:2); {write the status}
   end; {while}
end.
```

declarations, for example, all of the values happen to be whole numbers. We need a way to tell the compiler the difference between a whole number that is an integer and a whole number that is a real number, because they are very different things to the computer. We do that by including a decimal point in the number. The value 250.0 is a real number, but 250 is an integer. Any time you use a decimal point in a number, you must have a digit on both sides. The table below shows some illegal real numbers, and the way they should be written.

illegal	correct
250.	250.0
.1	0.1
0	0.0

We are also using a completely new way to loop over a group of statements. The while loop executes a statement (or, in this case, a compound statement) as long as a condition is true. In our while loop,

```
while principal > 0.0 do
```

the condition is that the principal must be greater than zero. The > character is a comparison operator. It compares the number to the left of the operator to the number to the right of the operator. If the left-hand number is bigger than the right-hand number, the result is true. If the left-hand number is smaller than or equal to the right-hand number, the result is false. The loop continues to execute the statements as long as the condition is true. In our program, the program continues until the car is paid off, at which time the principal is less than zero or equal to zero. There are a total of six comparison operators. The table below lists the operators and what they test for.

operator	test for...
a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b
a = b	a equal to b
a <> b	a not equal to b

For loops and while loops have much in common. Both are used to execute a statement more than one time. In the case of the for loop, though, we must know how many times the loop will be executed before we start. In the case of the while loop, we can loop until some condition is satisfied, without knowing in advance how many times through the loop it will take to satisfy the condition.

As with strings and integers, we can write a real number using the writeln statement. Unlike strings and integers, though, there are two numbers, not one, that tell the compiler how to format the number.

```
writeln(month:4, principal:10:2)
```

Listing 2.4

```
{ Why I don't own a Porche }

program IWannaPorche(output);

var
   month: integer;                {number of months}
   principal: real;               {amount left to pay}

begin
month := 0;                       {no payments made, yet}
principal := 24000.0;             {we start owing this much}
while principal > 0.0 do begin    {keep going until we're out of debt}
   month := month+1;              {it's a new month}
   principal := principal         {add in this month's interest}
      + principal*10.0/100.0/12.0;
   principal := principal-250.0;  {make the payment}
   writeln(month:4, principal:10:2); {write the status}
   end; {while}
end
```

The first number, 10, tells the compiler to put the number in a field ten characters wide, just as it would for a string or integer. The new number, 2, tells the compiler to print two digits to the right of the decimal point. In our case, we used this method to print the value as dollars and cents, with two digits for the cents.

Problem 2.4: Modify the sample program to find out how big the payments need to be to pay off the car in four years.

Hint: Start with a payment of $600, then increase or decrease the payment to get to a solution. You are playing a guess-the-payment game. If you pay off the loan in less than 48 months, or if you need to pay a lot less than the payment on the 48th month, you need to decrease the payment size. If it takes longer than 48 months, make the payment larger. You should only go to the nearest cent. The amount will not work out exactly.

Problem 2.5: Let's assume that you are working with the planning board of the local city government. You live in a pleasant city, but due to the local geography, the city can't expand indefinitely. You don't want the city to become too crowded, either. The current population size is 30,000 people. Everyone seems to agree that if the city gets any bigger than 50,000 people, it will be overcrowded.

One councilman has proposed new legislation to prevent the city from growing at more than 10% per year. At this rate, how long will it be before the city hits the limit of 50,000 people? Use a program very much like the sample program, but with a growing population instead of a shrinking principal to find out. Do you feel this is acceptable?

This is not an idle problem. While the numbers were different, this is exactly the same situation faced several years ago by the city of Boulder, Colorado. The answer they found caused some changes in the thinking of the city planners, and affected the outcome of some zoning legislation.

Problem 2.6: Inflation has been running at about 4% for the past few years. On average, then, something that costs $1.00 at the beginning of the year will cost $1.04 by the end of the year. Assuming a gallon of gas costs $1.00 today, what will it cost in ten years if inflation continues at 4%? Now try the same problem with a $100,000 house.

A few years ago, inflation was running at about 12%. Try this inflation figure. Is this rate a problem?

The Trace and Stop Commands

If you have been running the debugger on the past few programs, you may have noticed a disturbing trend. It takes a lot more steps to make it through one of our recent samples than it took to get through the samples in the last chapter. You may, in fact, want to step through a program a bit faster. The trace command will help you do this. The trace command does all of the things that the step command does. It moves the arrow in the source code window. It updates the variables window if you have one open. The difference is that the step command stops and waits after it executes a statement, while the trace command keeps on going. You can use the two commands together, starting off in step mode, then switching to a trace to execute several lines in a row. You can switch back to single-stepping at any time by selecting the step command again.

The trace command gives you one quick way to trace through to the end of a program. You can also select go, which runs even faster, since the computer does not draw the arrow or update the variables window. There is one case, however, when none of this will do any good. Type in the program in Listing 2.5, but don't run it until you have read the entire section.

You may notice that the only change from our previous sample was to change the interest rate to 15%. What is wrong with this program? Try to figure it out by doing the calculations through the end of the first loop by hand.

When you run this program, you will find that the monthly payment does not cover the interest. The principal grows, rather than shrinking. The result is that the program will never stop. This is one form of the infamous infinite loop. Once you realize that your program is in an infinite loop, trace or go won't get you to the end; there is no end. Instead, in cases like this, pull down the Debug menu and select the Stop command. That will stop the program in its tracks. Other than turning the computer off or resetting it, it is the only way out of an infinite loop. Considering how long it takes to boot from floppy disks, it's nice to know the way out.

Exponents

Integers were limited to a specific size. Real numbers have limits, too, but the limits are of a slightly different nature. This is because real numbers use

exponents to represent very large and very small numbers.

Exponents are the computers way of dealing with something called scientific notation. An exponent is a power of ten that follows the real number. For example,

```
2.5e2
```

means 2.5 times 10 raised to the power of two. You can also think of the power as the number of zeros to add to the 1. Ten to the power two is 100, for example. One-hundred times 2.5 is 250, so 2.5e2 is 250.

Exponents can also be zero. An exponent of zero means a 1 with no zeros, or just 1. Multiplying by one gives the original number, so 2.5e0 is just 2.5.

Finally, exponents can be negative. A negative exponent means to divide by ten to the indicated power, so 2.5e-3 means to divide 2.5 by 1000, giving 0.0025.

A quick way to work with exponents is to move the decimal point to the right for positive exponents, or to the left for negative exponents.

Real numbers can get quite large and quite small, but there is a limit to the size. In ORCA/Pascal, real numbers can have exponents in the range 1e-38 to 1e38. There is also a limit to the number of digits that can be handled. It's a lot like a calculator with a ten-digit display. If you need numbers with more than ten digits of accuracy, you have to get a different calculator. ORCA/Pascal real numbers have seven digits of accuracy. In other Pascals, you would have to check the manual to find out how accurate the numbers are, but these are fairly common values.

Many Pascals also support another type called double. Double values are handled just like real values, but they can have bigger exponents and are more accurate. In ORCA/Pascal, double values can have exponents that range from 1e-308 to 1e308, and can display seventeen digits accurately. The disadvantage to double values is that they are not a part of standard Pascal, so they are not available in all Pascal compilers. For that reason, we will avoid their use in this course.

Listing 2.6 shows how to use real numbers to represent very large numbers. By leaving off the width field in the writeln statement, we are telling the compiler to write the value in exponent format. This is also the format used by the debugger to display real numbers.

Listing 2.5

```
{ I owe, I owe, ... }

program IWannaPorche(output);

const
   cost = 24000.0;               {initial cost of car}
   APR = 15.0;                   {annual percentage rate}
   payment = 250.0;              {monthly payment}

var
   month: integer;               {number of months}
   principal: real;              {amount left to pay}

begin
month := 0;                      {no payments made, yet}
principal := cost;               {we start owing this much}
while principal > 0.0 do begin   {keep going until we're out of debt}
   month := month+1;             {it's a new month}
   principal := principal        {add in this month's interest}
      + principal*APR/100.0/12.0;
   principal := principal-payment; {make the payment}
   writeln(month:4, principal:10:2); {write the status}
   end; {while}
end.
```

Listing 2.6

```
{ There are about 5 billion people in the world.  Assuming    }
{ a growth rate of 3% per year, how many people will there    }
{ be in 100 years?                                            }

program People(output);

var
   people: real;                  {number of people}
   year: integer;                 {current year}

begin
people := 5e9;                    {5e9 is 5000000000, or 5 billion}
for year := 1989 to 2089 do
   people := people*1.03;
writeln('At 3% growth, there will be ', people,
   ' people in 2089.');
end.
```

Problem 2.7: Some germs can reproduce every twenty minutes. They reproduce by fission, where one germ splits in half to make two new germs. Assuming nothing stopped their growth, how many germs would there be after one day, starting with a single germ?

Lesson Two
Solutions to Problems

Solution to problem 2.1.

```
{ Write out five Fibonacci numbers }

program Fibonacci(output);

var
   last: integer;              {last number}
   current: integer;           {current number}
   next: integer;              {new number}

begin
last := 0;
current := 1;

next := last+current;
writeln(next);
last := current;
current := next;

next := last+current;
writeln(next);
last := current;
current := next;

next := last+current;
writeln(next);
last := current;
current := next;

next := last+current;
writeln(next);
last := current;
current := next;

next := last+current;
writeln(next);
last := current;
current := next;
end.
```

Solution to problem 2.2.

```
{ write a table of squares }

program Squares(output);

var
   i: integer;                  {loop variable}
   s: integer;                  {square of i}

begin
for i := 1 to 5 do begin
   s := i*i;
   writeln(i:10, s:10);
   end; {for}
end.
```

Solution to problem 2.3.

```
{ draw five concentric rectangles }

program Rectangles;

uses Common, QuickDrawII;

var
   i: integer;                  {loop variable}
   top, bottom, left, right: integer; {sides of the rectangle}

begin
SetPenMode(0);
SetSolidPenPat(0);
SetPenSize(3,1);

top := 10;
bottom := 70;
left := 10;
right := 100;

for i := 1 to 5 do begin
   MoveTo(left, top);
   LineTo(right, top);
   LineTo(right, bottom);
   LineTo(left, bottom);
   LineTo(left, top);
```

```
   left := left+6;
   right := right-6;
   top := top+6;
   bottom := bottom-6;
   end; {for}
end.
```

Solution to problem 2.4.

```
{ Pay off the car in 48 months }

program IWannaPorche(output);

const
   cost = 24000.0;               {initial cost of car}
   APR = 10.0;                   {annual percentage rate}
   payment = 608.71;             {monthly payment}

var
   month: integer;               {number of months}
   principal: real;              {amount left to pay}

begin
month := 0;                      {no payments made, yet}
principal := cost;               {we start owing this much}
while principal > 0.0 do begin   {keep going until we're out of debt}
   month := month+1;             {it's a new month}
   principal := principal        {add in this month's interest}
      + principal*APR/100.0/12.0;
   principal := principal-payment; {make the payment}
   writeln(month:4, principal:10:2); {write the status}
   end; {while}
end.
```

Solution to problem 2.5.

```
{ See how long it will take to reach a population of 50,000 }

program Population(output);

const
   currentPopulation = 30000.0;  {initial population of the city}
   growth = 10.0;                {population growth rate}

var
   year: integer;                {year}
   population: real;             {population during year}
```

```
begin
year := 1988;                      {current year -1}
population := currentPopulation;  {initialize the population}
while population < 50000.0 do begin {keep going until we hit 50,000}
   year := year+1;                 {new year}
   population := population        {add in this year's growth}
      + population*growth/100.0;
   writeln(year:6, population:10:1); {write the status}
   end; {while}
end.
```

Solution to problem 2.6.

By writing the solution to the first part of the problem carefully, you can reuse most of the program to answer the second two parts of the problem.

```
{ gas cost at 4% inflation }

program Inflation(output);

const
   currentCost = 1.00;             {current cost of the item}
   inflation = 4.0;                {inflation rate}

var
   year: integer;                  {number of years}
   cost: real;                     {cost of the item}

begin
cost := currentCost;
for year := 1990 to 1999 do
   cost := cost + cost*inflation/100.0;
writeln('Cost of a ', currentCost:10:2, ' item after 10 years is ',
   cost:10:2);
end.
```

All we have to change now is the starting cost of the item.

```
{ house cost at 4% inflation }

program Inflation(output);

const
   currentCost = 100000.00;        {current cost of the item}
   inflation = 4.0;                {inflation rate}

var
   year: integer;                  {number of years}
   cost: real;                     {cost of the item}
```

```
begin
cost := currentCost;
for year := 1990 to 1999 do
   cost := cost + cost*inflation/100.0;
writeln('Cost of a ', currentCost:10:2, ' item after 10 years is ',
   cost:10:2);
end.
```

For part 3, we need to change the inflation rate.

```
{ gas cost at 12% inflation }

program Inflation(output);

const
   currentCost = 1.00;            {current cost of the item}
   inflation = 12.0;              {inflation rate}

var
   year: integer;                 {number of years}
   cost: real;                    {cost of the item}

begin
cost := currentCost;
for year := 1990 to 1999 do
   cost := cost + cost*inflation/100.0;
writeln('Cost of a ', currentCost:10:2, ' item after 10 years is ',
   cost:10:2);
end.
```

Solution to problem 2.7.

```
{ Starting with 1 germ and assuming that each germ divides by }
{ fusion every 20 minutes, producing two new germs, how many  }
{ germs will there be after 24 hours?                         }

program Germs(output);

var
   germs: real;                {number of germs}
   time: integer;              {time loop counter}

begin
germs := 1;
for time := 1 to 24*3 do
   germs := germs*2;
writeln('After 24 hours, there will be ', germs, ' germs!');
end.
```

Lesson Three
Input, Loops and Conditions

Input

So far, all of your programs have only done one thing. No matter how many times you ran the program, unless you changed the program itself, it always did the same thing. The reason, of course, is that the programs could never ask you for any information. It's time to start controlling our programs a bit more through the use of input.

Your first program was a pretty simple one; it used the writeln procedure to write a string to the shell window. You have already learned to write integers and real numbers using writeln. Pascal uses the readln procedure in much the same way to read numbers and strings. When you used the writeln procedure to write to the shell window, you had to put the output file in the program statement. Whenever you use the readln procedure to read characters from the keyboard, you have to put the input file in the header. This lets the compiler know that you will be reading characters from the keyboard so it can take any steps needed to set up keyboard input. In our sample program, we are using both, so we must separate them with a comma.

Type in the following program and run it. Be sure to shrink the program window to less than half the width of the screen before you run the program, so you will be able to see the entire shell window.

```
{ Read an integer and write it to the screen. }

program ReadInteger(input, output);

var
   i: integer;

begin
readln(i);
writeln(i);
end.
```

When the program gets to the readln statement, it stops. In the shell window, you can see an inverse box. This is the input cursor; your program is waiting for you to type a number and press the return key. Go ahead and do that; when you do, the program reads the value, placing it in the variable i, and then writes it back to the screen.

When a Pascal program tries to read a number, it starts by skipping any blanks that you type. It also skips over the end of line character when you press the RETURN key. It keeps on skipping until it finds some non-blank character. If that character is not a number, the variable is set to 0. If the variable is a number, all numeric characters are read and converted into a number, which, in our program, is saved in the variable i. You can also start your number with a plus sign or minus sign.

Something very interesting happens if the number you type is too big. In lesson 2, you learned that integer variables have an upper limit on size; this limit is 32767. If you type a number bigger than 32767, the program will stop and write the message, "Subrange exceeded," to the shell window. This is called a run-time error. What that means is that the program is a legal Pascal program, but while it was running, something happened that the program could not deal with.

Problem 3.1: Try predicting what the program will write when you type in each of the following strings. Try running the program to see if you are right.

a. `3`
b. `4+9`
c. `3.14159`
d. `three`
e. `-8`
f. `+6 9`
g. `- 9`
h. press the RETURN key, then type: `7`
i. `8,536,912`
j. `8536912`

Our First Game... er, Computer Aided Simulation

Well, let's have some fun. Now that we can hold a simple conversation with the computer, we can write our first simple computer game, shown in Listing 3.1.

There are a lot of new concepts in this program, and we will spend a lot of time examining it in detail, but first type it in and run it. As with your first

Listing 3.1

```
{ Guess a number }

program Guess(input, output);

var
   i: integer;                          {input value}
   value: integer;                      {the number to guess}

begin
{Introduce the game}
writeln('In this game, you will try to');
writeln('guess a number.  I need a hint to');
writeln('help me pick numbers, though.');
writeln;

{get a seed for the random number generator}
writeln('Please type a number between 1');
write  ('and 30000: ');
readln(i);
seed(i);

{pick a number from 1 to 100}
value := RandomInteger mod 100 + 1;

{let the player guess the number}
repeat
   write('Your guess: ');             {get the guess}
   readln(i);
   if i > value then                  {check for too high}
      writeln(i:1, ' is to high.');
   if i < value then                  {check for too low}
      writeln(i:1, ' is to low.');
until i = value;                      {if the guess was wrong then loop}
writeln(i:1, ' is correct!');
end.
```

program, be sure and shrink the program window so you can see the shell window before running the program. You can't change the size of windows or stop the program while it is waiting for you to type a number, so this is an important step to remember.

The Repeat Loop

One of the new things in our program is a new statement, called the repeat loop. This is the third looping statement you have learned in Pascal. The first two, of course, are the for loop and the while loop. The repeat loop is also the last looping statement in Pascal! You're getting there...

Like the while loop, the repeat loop loops until some condition is satisfied. Unlike the while, loop, the condition is tested after the body of the loop has been executed at least one time. This means that the statements in the repeat loop are always executed at least one time, while the statements in the while loop can be skipped altogether. This is an important difference, and the key to why there are two loops instead of just one. To understand this difference, let's look at while loops and repeat loops from some of our programs, and compare the two.

Listing 3.2

```
while principal > 0.0 do begin    {keep going until we're out of debt}
   month := month+1;              {it's a new month}
   principal := principal         {add in this month's interest}
      + principal*APR/100.0/12.0;
   principal := principal-payment; {make the payment}
   writeln(month:4, principal:10:2); {write the status}
   end; {while}
```

Listing 3.3

```
repeat
   write('Your guess: ');             {get the guess}
   readln(i);
   if i > value then                  {check for too high}
      writeln(i:1, ' is to high.');
   if i < value then                  {check for too low}
      writeln(i:1, ' is to low.');
until i = value;                      {if the guess was wrong then loop}
```

In the last lesson, we wrote a program that showed how many payments were needed to buy a car, as shown in listing 3.2.

In this case, we needed to loop until the amount we needed to pay off was zero. It would be possible, although in this case not very likely, for the principal to be zero before the loop was ever executed. This is the key test for a while loop: you must ask yourself if it is possible for the condition that stops the loop to be true before you start. In other words, you want to know if it is possible that you may not want to execute the statements in the loop at all. If that is the case, a while loop should be used.

The repeat loop looks very similar. The only real difference is that the condition is evaluated at the end of the loop, not the beginning, as shown in listing 3.3.

The repeat loop is generally used in cases where the condition doesn't make sense until after the statements in the body of the loop have been executed at least one time. For example, it would seem to make sense to use a while loop that looks like the one in listing 3.4 to do the same job.

There is a major problem with this code, though. Before the loop is executed, you don't know what the value of i is. In fact, the random number generator has a one in 65536 chance of returning i itself for the first number, so your program will actually fail every once in a while. That, in computer terms, is called a bug. This is the worst kind of bug, because the program will work most of the time. When it does fail, you may be inclined to thing that the person who tells you about it is wrong. After all, you may have used the program

Listing 3.4

```
while i <> value do begin             {loop until the guess is correct}
   write('Your guess: ');             {get the guess}
   readln(i);
   if i < value then                  {check for too high}
      writeln(i:1, ' is to high.');
   if i > value then                  {check for too low}
      writeln(i:1, ' is to low.');
   end; {while}
```

several thousand times without seeing a problem!

There is a solution, though. You can start off by initializing i to a number different from value, as shown in listing 3.5.

This will work; the test will always fail the first time, so the person guessing the number always gets at least one chance to guess the number. On the other hand, the repeat loop works, to, but it doesn't require that you set the initial value before you start into the loop.

The acid test for when to use a repeat loop, then, is whether or not the test that ends the loop makes sense before the statements in the loop have been executed one time. In our example program, the test uses the number i, which is read in inside the loop. The test doesn't make sense until the number has been read at least one time, so we use a repeat loop.

You may have noticed one other difference between the while loop and the repeat loop. The while loop executes the statements if the condition is true, and stops when the condition is false. The repeat loop, on the other hand, loops if the condition is false, and stops when it is true. The names of the statements themselves will help you keep this straight. The while loop loops *while* a condition is true. The repeat loop loops *until* a condition is true.

The repeat loop is also the only Pascal statement that can loop over more than one other statement. In the case of the while loop and the for loop, we used the begin-end pair to group more than one statement together so the loop would loop over more than one statement. The repeat-until loop has its own pair of reserved words to group the statements together, so you don't have to use begin-end to do it. Just for the record, though, it will work: you can put a begin-end pair around any statements you like. If you prefer to put a begin-end pair in the repeat loop for style, it will work. I think it's a little silly, but it will, none-the-less, work. The code for this is shown in listing 3.6.

The Computer Bug

If you think about it, it's pretty peculiar that an error in a program would be called a bug. After all, it makes about as much sense to call an error an elephant, or a car. How did this happen?

Well, it all started back in the old days of computing. This was before microprocessors put an entire computer on a chip. It was before transistors put an entire computer in a small room. It was even before vacuum tubes allowed a calculator to fill only a small building (and allowed the operators to fry eggs on the hot components). This was in the good old days, when computers used relays.

For the fortunate uninitiated, a relay is a small switch. It uses a small electromagnet to control the flow of current. When electricity flows through the electromagnet, the electromagnet pulls on a small piece of metal, closing a switch so current can flow. When the electricity stops, a spring pulls the switch back open.

Back in those days, some intrepid souls had a program that didn't work. The darn thing should work; they went over the code again and again by hand. Finally, they decided that the program would fail if a particular bit in the computer would not work. A bit in the machine they were using was a relay, so they sent a repairman back to fix the broken component. Lo and behold, the component wasn't broken at all, but a miller moth had chosen the relay as a place to expire. It's body prevented the switch from closing. From that day on, a program that doesn't work is said to have a bug.

How Pascal Divides

Everyone knows that when you divide 3 by 2, the answer is 1.5. Unfortunately, the computer doesn't

Listing 3.5

```
i := value-1;                          {make sure we get into the loop}
while i <> value do begin              {loop until the guess is correct}
   write('Your guess: ');              {get the guess}
   readln(i);
   if i < value then                   {check for too high}
      writeln(i:1, ' is to high.');
   if i > value then                   {check for too low}
      writeln(i:1, ' is to low.');
   end; {while}
```

Listing 3.6

```
{It's a little strange to use begin-end in a repeat loop, }
{ but it does work!                                       }

repeat
   begin
      write('Your guess: ');
      readln(i);
      if i < value then
         writeln(i:1, ' is to high.');
      if i > value then
         writeln(i:1, ' is to low.');
   end;
until i = value;
```

know that, at least not when you are using integers. An integer can't have the value 1.5; integers can only be whole numbers. You could use real numbers most of the time, but real numbers take more space, and it takes a lot more time to do an operation using real numbers than using integers. The program in listing 3.7 will help us explore this, and many other curiosities about the div, mod and / operators.

Here are the results from running this program. If you run the program, you won't be able to see all of the output in the shell window at one time. You can see most of it, though, if you resize the shell window.

```
first       :-10
last        :10
denominator :4
```

Listing 3.7

```
{ Exploring div, mod and /}

program Divide(input,output);

var
   n : integer;                       {the "top" number in the division}
   d : integer;                       {the "bottom" number in the division}
   first,last: integer;               {the first/last number to divide}

begin
write('first       :');               {get the first/last numbers}
readln(first);
write('last        :');
readln(last);
write('denominator :');               {get the number to divide by}
readln(d);
writeln;                              {write the table}
writeln('     a     b  a mod b  a div b      a/b');
for n := first to last do
   writeln(n:6, d:6, n div d:9, n mod d:9, n/d:9:4);
end.
```

a	b	a mod b	a div b	a/b
-10	4	2	-2	-2.5000
-9	4	3	-2	-2.2500
-8	4	0	-2	-2.0000
-7	4	1	-1	-1.7500
-6	4	2	-1	-1.5000
-5	4	3	-1	-1.2500
-4	4	0	-1	-1.0000
-3	4	1	0	-0.7500
-2	4	2	0	-0.5000
-1	4	3	0	-0.2500
0	4	0	0	0.0000
1	4	1	0	0.2500
2	4	2	0	0.5000
3	4	3	0	0.7500
4	4	0	1	1.0000
5	4	1	1	1.2500
6	4	2	1	1.5000
7	4	3	1	1.7500
8	4	0	2	2.0000
9	4	1	2	2.2500
10	4	2	2	2.5000

The / operator gives the results we would normally expect when dividing two numbers. When we divide 5 by 4, for example, the table shows that the answer is 1.25. The / operator is the division operator for real numbers. You can use it to divide a real number by another real number, or, as we have done, to divide an integer by an integer. You can also mix real numbers and integers. The result, though, is always a real number.

It takes the computer a lot longer to do math with real numbers than it does to do math with integers. Real numbers also take up more room in the computer's memory; this is something that can be very important when you are working with large groups of numbers called arrays. (Arrays are covered in detail later in the course.) There are also places in the Pascal language where you can only use an integer. For these, and some other reasons you will discover later, programmers often choose to work with integers instead of real numbers. Since the / operator always returns a real number, we need some way in Pascal to get an integer result from a division. In Pascal, we use the div operator.

Unlike the / operator, the div operator only works with integers. It always gives an integer answer, too. Of course, that means that the div operator can't give 1.25 for the answer when you divide 5 by 4, as the / operator does. Instead, the div operator chops off any digits to the right of the decimal place, and returns the integer part that is left. The same is true when the answer is a negative number, as you can see from the table. When you divide -5 by 4 using the div operator, the answer is -1.

There are times when you need to know the remainder from an integer division. The mod operator is used to get the remainder. Like the div operator, the mod operator only works with integers, and always gives an integer for the answer. When both numbers are positive, the mod operator returns the remainder from the division. For example, 7 divided by 4 is 1 with a remainder of 3, so 7 div 4 is 1, and 7 mod 4 is 3.

Looking at the table from our sample program, you can see that the mod operator behaves a little strangely when one number is negative. For example, (-1) mod 4 is 3, not 1 or -1, as you might expect. The reasons for this are tied up in an obscure branch of mathematics called modular arithmetic. To figure out what the mod operator will give you when the first number is negative, take a close look at our table. The numbers repeat a series over and over again, even for negative arguments.

As you may know, it is not legal to divide by zero. This restriction is due to the nature of numbers. Pascal enforces this restriction; your program will stop with an error if you try to divide a number by zero. Modular arithmetic also imposes an additional restriction on the mod operator. You can use a negative number on the left-hand side of the operator, as our sample program does, but you cannot use a negative number (or zero) on the right-hand side of the mod operator.

All of this may seem a bit obscure and theoretical, and it really is. We can use this information to do some pretty neat stuff, though. Looking at the table from our sample program, you can see that the mod operator always returned a value from 0 to 3. In our number guessing game, we used the mod operator to restrict the value returned by the random number generator to a small range of numbers.

```
value := RandomInteger mod 100 + 1;
```

The RandomInteger function returns a number from -32767 to 32767. We could just expand the range of numbers that the game will let you guess to the full range of values returned by RandomInteger, but it might get boring trying to track down a number in that range. The result of

```
RandomInteger mod 100
```

is always in the range 0 to 99, though. By adding 1, we get a number in the range 1 to 100. To change our game so that you have to guess a number from 1 to 500, then, we just change this line to

```
value := RandomInteger mod 500 + 1;
```

Nesting Loops

If you are very sharp at logic, know a lot about probability, and remembered that the range of integers is -32767 to 32767, you may have noticed that our number guessing program has a slight flaw. It isn't one most people would ever notice, but it is there.

The problem is that not all numbers have an equal chance of being the number picked by the game. You see, there are 327 possible ways to pick each of the numbers 0 through 99 from the positive integers 0 to 32699, and 327 possible ways to pick the same numbers from the negative integers -32700 to -1. You can see why by looking at the results of the mod operator in the table in the last section. The result of RandomInteger mod 100 will be zero when RandomInteger returns 0, 100, 200, and so forth, up to 32600. All together, that's 327 ways to get a result of 0. The result will be 1 when RandomInteger returns 1, 101, 201, and so on. The same thing happens for negative numbers, where there are 327 more ways to get each of the numbers from 0 to 99.

But there is also another way to get the integers 0 to 67 from the positive numbers 32700 to 32767, and an extra way to get the integers 33 to 99 from the negative integers -32767 to -32701. If you play poker, and you know there is an extra ace of spades in the deck, you know it can effect the outcome of the game. Having a few extra ways to get some of the numbers from 0 to 99 also effects our number-guessing game.

The table below summarizes this, showing the chance that each number will be picked.

guess	ways to get it	probability it is the guess
0 to 32	327+327+1	0.0099947
33 to 67	327+327+1+1	0.0100099
68 to 99	327+327+1	0.0099947

Ideally, each number should have a 0.01 chance (one in a hundred) of being chosen. As you can see, the difference isn't all that great, but in some simulations it might just be important. A simulation of a roulette wheel, a craps game, or a poker game is one case where the difference could be critical.

One way to even out the chances that each number will be picked is to eliminate any answer that isn't in the range 1 to 100. The versatile repeat loop can be used to do this. We haven't learned how to test for one condition *and* another yet, but by nesting two repeat loops, we can get the same effect.

```
{pick a number from 1 to 100}
repeat
   repeat
      value := RandomInteger;
   until value > 0;
until value <= 100;
```

Looking at this another way, the outside loop does this:

```
repeat
   <get a number>
until value <= 100;
```

This loop loops until the statements in the loop return a number less than or equal to 100. This guarantees that the number isn't over 100, but it doesn't prevent numbers less than 1, like -2000.

The interior loop,

```
   repeat
      value := RandomInteger;
   until value > 0;
```

picks random numbers that are greater than zero. It loops, getting numbers, until it finds one that satisfies the condition. When it finds one, the outside loop does its test. Together, the loops guarantee that value will have a result between 1 and 100. It also evens out the chances that each number will be picked.

The point here isn't really to pick better random numbers for our simple guessing game; the numbers were really good enough for our program already. The point is that you can put anything inside of a repeat loop – even another repeat loop. You can also put a for loop inside a while loop, a while loop in a repeat loop, a for loop inside of a while loop that is inside of a repeat loop, and so on.

When you put one loop inside of another, programmers call it a nested loop. There is no theoretical limit to how many times you can nest loops. You can, for example, put a for loop inside of a for loop that is inside of a for loop that... well, you get the idea. In ORCA/Pascal, you will eventually run out of memory if you nest too many loops. You can generally count on nesting to 20 or 30 levels, though. In real programs, three or for nested loops are pretty unusual, so the limit isn't very important.

Problem 3.2. In this problem, we will go a little crazy with repeat loops. We will also be making several changes to our existing guessing game. Rather than making all of the changes at once, make each change, run the program, make sure it works, and

then move on. That way, if you make a mistake, you know exactly where the mistake is. This concept, called stepwise refinement, is a big help in developing programs quickly. The reason is that finding bugs is generally the hardest part of correcting a bad program. If you make a small change, and the program fails, you know exactly where to look for the problem. If you make a lot of changes, you have a lot more places to look for the error.

The first step is to use the two nested repeat loops outlined above to pick the random number, rather than the mod operator.

Next, add a repeat loop around the part of the program that picks a number and lets the player guess it. This repeat loop should loop until a new integer, named done, is zero. Initialize done to three before the loop starts, and subtract one from it inside the loop. At this point, the player gets to play the game three times, rather than one.

Remove the statement that initializes done to three, as well as the statement that subtracts one from the value. Instead, at the end of the loop, put in yet another set of nested repeat loops. Inside the loops, print a blank line, followed by

```
Play again?  Enter 0
for no, 1 for yes:
```

and read a number. Use the two repeat loops to make sure the answer is either a zero or one, using the same technique used earlier to make sure a value was between 1 and 100.

At this point, you have an impressive set of nested statements, each doing a specific job. One large repeat loop plays the game until the player wants to stop. Inside of this loop are two sets of nested repeat loops that make sure a number is in a particular range. There was already one repeat loop in the program, looping until the player guesses the right number. All together, that's six repeat loops. It sounds complicated, but by building the program up little by little, you have accomplished a complicated task without straining too many little grey cells. This technique of stepwise refinement is one of the most important ideas in modern programming practice. Simply put, you break a problem down by only worrying about a little of it at a time. You will see this idea over and over in this course.

Problem 3.3. In the text, there is an innocent looking statement: "it takes a lot more time to do an operation using real numbers than using integers." Write a program to test this statement.

Your program should use two nested for loops; this is necessary because programs run so fast on the Apple IIGS. The first loop should loop from 1 to 10000, while the interior loop should loop from 1 to 25. Just put a semicolon after the second for loop, like this:

```
for i := 1 to 10000 do
   for j := 1 to 25 do
      ;
```

The body of the loop does nothing, which is legal. Altogether, this means that the code in the loop (none at this point) will be executed 250000 times -- a quarter of a million times!

One other step you should take is too turn off debug code. ORCA/Pascal generates code to make the step-and-trace debugger work; this extra code takes extra time to execute. Your programs will execute faster if you turn off the debug code. (You will not be able to debug the programs, though, nor will you be able to stop them using the Stop command.) Turn off the debug code by selecting Compile from the Run menu, and clicking on the check box labeled "Generate debug code." (There should not be an X in the box when you are done.) Click on the Set Options button to accept the change.

Start by timing the program using a watch with a second hand, or a stopwatch if you have one. This is the time it takes the computer to do the nested for loops.

Next, define three integers, a, b and c. Before the first for loop, set a to 3, and b to 5. In the body of the loop, add a to b and save the result in c. Execute the program, and time it. You can figure out how long it takes to do a single integer add by subtracting the time for the empty loop from this time, and dividing by 250000. If you don't have a calculator handy, you can write a short program to do the math.

Repeat the process, but this time, define a, b and c as real numbers. The program will take a lot longer when you use real numbers, so be sure and

change the value in the first for loop to 100, rather than 10000. (If you don't, the program will run for several *hours*!) The loop will look like this:

```
for i := 1 to 100 do
   for j := 1 to 25 do
      c := a+b;
```

The time needed to do one math operation is small for both integers and real numbers; less time than it takes to blink your eye. On the other hand, it takes a lot longer to add two real numbers than to add two integers. You can find out how much longer by dividing one result by the other. The answer is why programmers use integers whenever they can, and one reason why computers bother with having two different kinds of numbers, instead of using real numbers for everything.

You have already learned the point of this problem, but if you are curious, you might also see how long it takes to subtract, multiply and divide numbers. Also, for everything but integer addition and subtraction, the amount of time required varies depending on what the numbers are. You might try a few numbers to get an idea how much the difference can be. If you are just a little curious, but not curious enough to change the program and run it for each operation, you can find the times for the numbers used in this problem in the solutions.

Random Numbers

One of the new concepts used in our sample program is the random number. You have probably heard that computers are very precise, and that is certainly true. In our number guessing game, though, the last thing we want is for the computer to be precise. If we know beforehand what number the computer will pick, this game just won't be much fun. The program uses something called a random number generator to get around this problem.

A random number generator is basically a way for the computer to generate a number, or series of numbers, that seem to be random. Since the computer can only do very specific things, the numbers aren't really random, but they are very hard to predict, and that is good enough for a lot of programs. Since the numbers really aren't random, they are technically called pseudo-random numbers. That's a real mouthful, though, so we will continue to call them random numbers.

To learn more about random numbers, we will write a simpler program.

Type in the program in listing 3.8 and run it. It will ask you for a random number seed, and then print ten pseudo-random numbers based on the seed value you give the program. One of the things you should try is giving the program the same seed value more than one time. When you do, you will always get the same ten numbers. You should also try giving the program several different numbers. These will produce a different series of ten numbers. Within each group of ten numbers, though, it is hard to predict what the next one will be by looking at the numbers that come before it. This is the heart of a good random number generator.

We will use random numbers in many of our

Listing 3.8

```
{ A closer look at pseudo-random numbers }

program random(input,output);

var
   i: integer;                          {loop variable}
   seedValue: integer;                  {random number seed}

begin
write('Random number seed = ');
readln(seedValue);
seed(seedValue);
for i := 1 to 10 do
   writeln(RandomInteger);
end.
```

example programs. Random numbers help us to write programs that don't do exactly the same thing each time we use them; that's something we will need over and over again. Here are some places where random numbers are commonly used:

1. Random numbers are used in games like Chess. Games work by scoring moves; the move with the best score is the one the computer makes. If two moves have the same score, random numbers can be used to choose between them so the computer doesn't play exactly the same way each time. In a game like chess or checkers, there are also many good ways to make the first few moves; these are called opening books. Random number generators are used to pick an opening from the opening book.

2. Many dungeon and dragon style computer games work based on probabilities. For example, a character with a particular set of characteristics might have a probability of 0.4 of killing a giant ant with a broadsword. The ant, conversely, might have a 0.2 chance of damaging the player. A random number generator can be used to generate a number between 0 and 99, as our number-guessing game did. The player kills the ant if the number is less than 40. Next, another number is chosen, and the ant hurts the player if the number is less than 20.

3. Computers are often used to do serious simulations. Computer simulations are used to study traffic patterns, wars, and the spread of diseases. As an example, let's assume that you are trying to protect Yellow Stone National Park from forest fires. You could choose to "let it burn," letting nature take its course. You could choose to fight all fires aggressively, but that would lead to a gradual build-up of weeds and wood to burn. You might choose to cut fire lanes through the forest. All of these possibilities can be examined using computer simulations.

4. Random number generators are used in card games to shuffle cards. The random number generator is used to pick which card will be taken from the deck next, taking one card from the remaining cards that have not been dealt.

Problem 3.4. Write a program to throw two six-sided dice twenty times. Use the same ideas used in the number-guessing game. Write the number of spots showing on each of the dice to the screen.

Be sure and use constants for the number of dice and the number of sides on the dice. That way, if you want to use the program to throw fifty twenty sided dice, you can.

Problem 3.5. You can draw a dot in the graphics window by doing first a MoveTo, then a LineTo the same spot. For example,

```
MoveTo(10,10);
LineTo(10,10);
```

draws a dot a 10,10.

Write a program that gradually blackens the rectangle with a left edge of 10, a top of 10, a right edge of 200, and a bottom of 70. Do this using a for loop that loops from 1 to max, where max is defined at the top of your program as a constant. Use a value of 11651 for max.

Inside the for loop, pick two random numbers. The first should be in the range 10 to 200; assign this value to an integer variable called x. The second should be in the range 10 to 70; assign this one to the variable y. Draw a dot at this point using a MoveTo-LineTo sequence.

The result is a program that gradually fills the area with black snow.

There are 11651 dots in the area you are filling, but when the program finishes, not all dots are black. Why?

Problem 3.6. Change the program from problem 3.5 to create multicolored snow by picking the color of the dot randomly. The color should be in the range 0 to 3.

Multiple Reads with Readln

Up to this point, our use of readln has been pretty minimal. All we have used readln for is to read a single integer. Like writeln, readln can read more than one thing at a time. It can also read real numbers or even strings.

To look at how readln works, we'll use another short program.

```
{ Multiple reads }

program random(input,output);

var
   n1,n2,n3,n4: integer; {values to read}

begin
readln(n1,n2);
readln(n3,n4);
writeln(n1,n2,n3,n4);
end.
```

At the beginning of this lesson, you learned that readln skips blanks and end of line markers in search of a number to read. When it reads a number, it reads all digits (the characters 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 are digits) until a non-digit character is found. A space or the end of a line counts as a non-digit character. If there are two numbers to read, as in our program, Pascal again skips blanks and end of line marks in search of a number, reading the new number using the same rules as the first number.

Once all of the numbers have been read, readln skips all characters up to the next end of line marker. It doesn't matter if there are valid numbers, characters, or blanks remaining on the line; all characters get skipped. The end of line mark is also skipped. In our program, then, any characters appearing after the second number, but on the same line, will be skipped. The second readln starts looking for a number at the first character of a new line.

Problem 3.7. What will our test program print out for each of the following sets of input? After you decide, run the program and see if you are right.

first set:
```
1   2
3   4
```

second set:
```
1   2   a   4
5   6   7   8
```

third set:
```
1   a   2
3   4   5
```

fourth set:
```
1
2   3
4   5
```

fifth set:
```
1

2

3

4
```

Reading Real Numbers

So far, we have used the readln statement to read integers. It's just as easy to read real numbers like 1.5, or 10.0. The only difference, in fact, is that the variable you are reading should be declared as a real number, rather than an integer.

When your program reads a real number, you have a considerable amount of freedom concerning how you type the number. You can type it as an integer, in which case the number will be automatically converted to a real number. You can type the number with a leading decimal point, a trailing decimal point, or a decimal point imbedded in the number. You can also type an exponent, with or without a sign, with either an uppercase E or lowercase e. In fact, all of the following numbers are valid real numbers when you are typing a number as input to a program.

```
1           10.0          10.
.1          3.14159       1e10
2.56E+2     -16e004       327541e-16
.1e1
```

Problem 3.8. Modify the first sample program from this lesson to read real numbers, rather than integers. To do this, all you have to do is change the variable i from an integer to a real number. Run the program, giving it the numbers shown in the table, above.

Readln and Read

Just as there are two forms of the output statement (write and writeln), there are also two forms of the input statement. So far, we have used readln, which skips to the end of a line after reading the last value. The other form, read, works exactly the same way, but it doesn't skip to the end of a line after reading the last value. One way of looking at this is that

```
readln(n1,n2);
```

means exactly the same thing as

```
read(n1);
readln(n2);
```

The If Statement

Computer programs can make decisions. You have already written some programs that use this capability in the form of loops that keep going until some condition is satisfied. In some cases, though, we may only need to do something once, or we may not need to do it at all. That's where the if statement comes in.

```
if <condition> then
   <statement>;
```

The if statement evaluates the same kind of condition that you have already used in the repeat and while statements. The condition is followed by the reserved word then; this just tells the compiler that you are finished with the condition. If the condition is true, the next statement is executed. If not, the statement is skipped. In a way, the if statement is like a while loop that doesn't loop.

As with the while loop and for loop, the if statement only executes one statement. If more than one statement must be executed, you use the begin-end reserved words to group more that one statement into a compound statement, just as you do with the for loop and while loop.

To see how all of this works, let's try a simple example. In this example, we will use the if statement, along with the mod operator you met earlier in the lesson, to write a program that can count change.

Listing 3.9

```
{count change}

program Change(input, output);

var
   change: integer;

begin
write('How many cents in the change? ');     {get the amount}
readln(change);
writeln('Your change consists of:');         {write the header}
if change >= 100 then begin                  {count out the dollars}
   writeln(change div 100: 10, ' dollars');
   change := change mod 100;
   end; {if}
if change >= 25 then begin                   {count out the quarters}
   writeln(change div 25: 10, ' quarters');
   change := change mod 25;
   end; {if}
if change >= 10 then begin                   {count out the dimes}
   writeln(change div 10: 10, ' dimes');
   change := change mod 10;
   end; {if}
if change >= 5 then begin                    {count out the nickles}
   writeln(change div 5: 10, ' nickles');
   change := change mod 5;
   end; {if}
if change <> 0 then                          {count out the pennies}
   writeln(change: 10, ' pennies');
end.
```

In the program in listing 3.9, each if statement is used to see if the number of pennies left is large enough to give the customer at least one coin of a given size. For example, the first if statement checks to see how many dollars are in the change. Since we need to do two things – write the number of dollars and adjust the amount of change – the statements are enclosed in a begin-end sequence.

```
if change >= 100 then begin
   writeln(change div 100: 10, ' dollars');
   change := change mod 100;
   end; {if}
```

If there are more than 100 pennies due, then we can start by giving out some number of dollars. Since the div operator automatically truncates the result, it gives us the dollar amount with no hassles. (The automatic rounding can be useful, as well as a pain, depending on the application.) For example, if the number of pennies due in change is 536, then change will certainly be greater than or equal to 100. In that case, the program writes 536 div 100, which is 5, as the number of dollars. The mod operation then strips off the dollar amount; 536 mod 100 is 36. The program goes on from there.

It would be a good idea to run this program, using the debugger to step through it line by line until you are sure you know how the if statement works. Be sure and use the variables window to track the value of the variable change.

The Else Clause

There are many times when you need to do one thing or another, depending on some condition. In that case, you could use two different if statements, one after the other, but you can also use an else statement. As a simple example, let's say you are printing the number of tries it took to guess the number in our number guessing game. It's sort of tacky to print out "1 tries," or worse still, "2 try." With the if-then-else statement in listing 3.10, you can print something a bit prettier.

Problem 3.9. Modify the program from lesson 2 that showed payments for purchasing a car. Allow the user of the program to enter the cost of the car, the interest rate and the number of payments as real numbers. Use an if statement to see if the payment is larger than the amount of interest that will accumulate in one month. If not, print an appropriate error message. If the payment is large enough, execute the program as it worked before.

Those Darn Semicolons

There is one very important point to keep in mind. The semicolon in Pascal is a statement separator, not a statement terminator. In plain English, this means that the semicolon is used between two statements to tell the compiler where one statement ends and the next one starts. The if-then-else sequence is one big statement. If you put a semicolon after the first writeln, right before the else, the compiler thinks you want to end the if statement. When it finds the else, it screams bloody murder. Well, it prints an error message, anyway. Go ahead and try it now. If you are normal, you will make the mistake many times. Seeing what happens when you are thinking about the problem will help you to remember what is going on when it happens later.

One way to remember all of this is that you never, ever put a semicolon before an else.

Just for the record, the reserved word end marks the end of a compound statement. What that means is that a semicolon is not needed before an end. While it is not needed, it also doesn't hurt anything. In practice, as you change a program, you will get more errors from leaving the semicolons off before an end than from putting them on, since there are far more places where, with the end removed, you will need the semicolon than there are places where the semicolon will not be needed. If you are a purist, or have a purist instructor, you might want to leave them off. In all of my programs, though, I put them in.

You might wonder why it is legal to put a semicolon before and end if it isn't really needed. The reason it works is that a null statement is legal in Pascal. A null statement is a statement that does nothing. It also has no characters. You see, the statement

Listing 3.10

```
if tries = 1 then
   writeln('You guessed the number in 1 try!')
else
   writeln('It took ', tries:1, ' tries to guess the number.');
```

```
;
```

is perfectly legal in Pascal. Technically, the line

```
;;;
```

consists of three null statements. In fact, the program

```
program Goofy;

begin
;;;;;
end.
```

actually has six statements in it, none of which do anything.

Whoops, that was a misprint, right? After all, there are only five semicolons, not six, so there are five statements, right? No, there are six. Again, the point is that the semicolon is a statement separator. Any time you see a semicolon, there is a statement before it and a statement after it. Let's redo that program to see the point a little more clearly. We will put a number in the comment to number each statement.

```
program Goofy;

begin
{1};{2};{3};{4};{5};{6}
end.
```

Now you can see why putting an extra semicolon right before the end statement is legal in Pascal. What you are really doing is adding a new, null statement. For example, in the change counting program, the compound statement shown has three statements in it: a writeln statement, an assignment statement, and a null statement.

```
if change >= 100 then begin
   writeln(change div 100: 10, ' dollars');
   change := change mod 100;
   end; {if}
```

Another place where we used a null statement was the problem that timed math operations. To time the empty loop, we coded a null statement, like this:

```
for i := 1 to 10000 do
   for j := 1 to 25 do
      ;
```

In this case, the loop was doing exactly what we wanted it to do: nothing. The purpose was to see how long it took to do the loops, with no statements. It does, however, point out the second most common bug when using semicolons in Pascal. If you put at the end of each physical line, you are inserting a lot of null statements. You will get an error if you put a semicolon before an else clause, but unfortunately,

```
{a wrong way to use semicolons}
for i := 1 to 10 do ;
   writeln(i);
```

is legal in Pascal. You end up with a for loop which loops over a null statement, followed by a writeln statement that has nothing to do with the for loop. Since the statements are legal, the compiler doesn't complain, but the program doesn't do what you want it to do. The same thing can happen with a while loop.

Once you understand semicolons, and realize what it means for a semicolon to be a statement separator, you won't have problems like these very often. When you do, you will be able to spot the error right away. Just keep in mind that the semicolon is a statement separator, and not something you use at the end of every statement, and it will all make sense.

Nesting If Statements

You can put any statement you want in an if or else clause, including another if statement. This can be useful when you have several possibilities that you need to choose from. For example, let's assume that you want to print out a message like "that was your 3rd try." You can print the number of tries, followed by "rd," but that only works for some numbers. You would want to print

1st
2nd
3rd
4th
5th

and so on. One way to go about it is to print "that was your," followed by a series of if statements, followed by printing "try." The if statements can be used to decide the suffix for the number of tries. Here's a code fragment that does the job.

```
write('That was your ');
if try = 1 then
   write('1st')
else if try = 2 then
   write('2nd')
else if try = 3 then
   write('3rd')
else
   write(try:1, 'th');
writeln(' try!');
```

Note the indenting structure. It would be perfectly reasonable to indent this code fragment like this:

```
write('That was your ');
if try = 1 then
   write('1st')
else
   if try = 2 then
      write('2nd')
   else
      if try = 3 then
         write('3rd')
      else
         write(try:1, 'th');
writeln(' try!');
```

The second method shows more accurately what Pascal does to evaluate the statement. It starts by checking to see if try is 1. If so, "1st" is written. If not, it moves on to another check, and so on. The original method, though, with "else if <condition> then" all on one line, shows the logical flow of the program better. It clearly shows that we are choosing one of several alternatives, while this isn't exactly clear from the second example. The compiler doesn't care how, or even if you indent. It will create the same program either way. Indenting is for your benefit, not the compilers. For that reason, I would recommend the first method.

Problem 3.10. In this problem, you will write a bouncing ball program. You will move a small spot across the graphics window. When the spot gets to the edge of the graphics window, it will bounce off.

Compared to some of the programs you have written, this is a fairly long one, so we will develop it in steps. To make a ball bounce around in the graphics window, you will need to animate the ball. Start out by writing a short program that moves a spot from 0,0 to 50,50. You will start by initializing two integer variables, x and y, to 0. In a for loop, you will then set the pen color to white (a color of 3) and draw a spot using a MoveTo-LineTo pair, as we did earlier. Next, you increment both x and y by one, change the pen color to black (a color of 0), and draw the dot again. Get this program to work first: it should move the dot across the screen in a diagonal line.

One problem with this program is that the ball moves too fast. To slow it down, put a for loop inside the for loop that moves the ball. The new for loop should step from 1 to 1000, but doesn't need to do anything. To do nothing, you just code an empty statement by putting a semicolon in the program, like this:

```
for i := 1 to 1000 do
   ;
```

Next, ask the user for a starting x, a staring y, the number of iterations (put this in a variable called iter), and an x speed and y speed (put these in xSpeed and ySpeed). Check to see if the x and y values are in the graphics window using if statements. If not, adjust them to be in the window. (The graphics window starts out as 316 pixels wide and 83 pixels high.) Loop over your code to move the ball iter times.

On each loop, you will need to do the following:

1. Execute the dummy loop that loops from 1 to 1000 to slow things down a little.
2. Draw a white spot at the old x and y.
3. Add the xSpeed to x. This moves the ball over.
4. If x is off the screen to the left (less that zero), set it to zero and set xSpeed to -xSpeed.
5. If x is off the screen to the right (greater that 316), set it to 316 and set xSpeed to -xSpeed.
6. Add the ySpeed to y. This moves the ball up or down.
7. If y is off the screen to the top (less that zero), set it to zero and set ySpeed to -ySpeed.
8. If y is off the screen to the bottom (greater that 83), set it to 83 and set ySpeed to -ySpeed.
9. Draw a black spot at the new x and y.

The result is a bouncing ball, very much like the ball you may use in a break out game. The moving spot idea is also used for projectiles in shoot-em-up arcade games. The moving spot, while simple, is

also how all animation is done. One shape is erased, and another drawn quickly enough that the shape seems to move.

The practice of writing our the steps for a program in a kind of semi-English form is very useful for designing programs. The roughed-out version of the code is called pseudo-code. It won't run on any computer (at least, none that are available today!), but it helps when you are working on the logic of a program.

Lesson Three
Solutions to Problems

Solution to problem 3.1.

input	value read by readln
3	3
4+9	4
3.14159	3
three	0
-8	-8
+6 9	6
- 9	0
(RETURN)7	7
8,536,912	8
8536912	<run time error>

Solution to problem 3.2.

The first step uses repeat loops to pick a number between 1 and 100, rather than the mod operator:

```
{ Guess a number }

program Guess(input, output);

var
   i: integer;                          {input value}
   value: integer;                      {the number to guess}

begin
{Introduce the game}
writeln('In this game, you will try to');
writeln('guess a number.  I need a hint to');
writeln('help me pick numbers, though.');
writeln;

{get a seed for the random number generator}
writeln('Please type a number between 1');
write  ('and 30000: ');
readln(i);
seed(i);

{pick a number from 1 to 100}
repeat
   repeat
      value := RandomInteger;
   until value > 0;
until value <= 100;
```

```
{let the player guess the number}
repeat
   write('Your guess: ');
   readln(i);
   if i > value then
      writeln(i:1, ' is to high.');
   if i < value then
      writeln(i:1, ' is to low.');
until i = value;
writeln(i:1, ' is correct!');
end.
```

In the second step, we add a repeat loop to let the player play three games, rather that one:

```
{ Guess a number }

program Guess(input, output);

var
   i: integer;                         {input value}
   value: integer;                     {the number to guess}
   done: integer;                      {are we done yet?}

begin
{Introduce the game}
writeln('In this game, you will try to');
writeln('guess a number.  I need a hint to');
writeln('help me pick numbers, though.');
writeln;

{get a seed for the random number generator}
writeln('Please type a number between 1');
write  ('and 30000: ');
readln(i);
seed(i);

done := 3;
repeat
   {pick a number from 1 to 100}
   repeat
      repeat
         value := RandomInteger;
      until value > 0;
   until value <= 100;
```

```
   {let the player guess the number}
   repeat
      write('Your guess: ');
      readln(i);
      if i > value then
         writeln(i:1, ' is to high.');
      if i < value then
         writeln(i:1, ' is to low.');
   until i = value;
   writeln(i:1, ' is correct!');

   {play another round}
   done := done-1;
   writeln;
until done = 0;
end.
```

Finally, we let the player decide if he wants to play again, using nested repeat loops to make sure the number is in the correct range:

```
{ Guess a number }

program Guess(input, output);

var
   i: integer;                      {input value}
   value: integer;                  {the number to guess}
   done: integer;                   {are we done yet?}

begin
{Introduce the game}
writeln('In this game, you will try to');
writeln('guess a number.  I need a hint to');
writeln('help me pick numbers, though.');
writeln;

{get a seed for the random number generator}
writeln('Please type a number between 1');
write  ('and 30000: ');
readln(i);
seed(i);

repeat
   {pick a number from 1 to 100}
   repeat
      repeat
         value := RandomInteger;
      until value > 0;
   until value <= 100;
```

```
   {let the player guess the number}
   repeat
      write('Your guess: ');
      readln(i);
      if i > value then
         writeln(i:1, ' is to high.');
      if i < value then
         writeln(i:1, ' is to low.');
   until i = value;
   writeln(i:1, ' is correct!');

   {play another round?}
   repeat
      repeat
         writeln('Play again?  Enter 0');
         write  ('for no, 1 for yes:');
         readln(done);
      until done >= 0;
   until done <= 1;
   writeln;
until done = 0;
end.
```

Solution to problem 3.3.

```
{ Empty timing loop }

{ Note: be sure to run this program with debug turned off! }

program TimeIt(output);

var
   i,j: integer;                        {loop variables}

begin
writeln('Start timer...');
for i := 1 to 10000 do
   for j := 1 to 25 do
      ;
writeln('Stop timer.');
end.
```

```
{ Time 250,000 integer additions }

{ Note: be sure to run this program with debug turned off! }

program TimeIt(output);

var
   i,j: integer;                        {loop variables}
   a,b,c: integer;                      {values used in calculation}
begin
writeln('Start timer...');
a := 3;
b := 5;
for i := 1 to 10000 do
   for j := 1 to 25 do
      c := a+b;
writeln('Stop timer.');
end.


{ Time 2500 real additions }

{ Note: be sure to run this program with debug turned off! }

program TimeIt(output);

var
   i,j: integer;                        {loop variables}
   a,b,c: real;                         {values used in calculation}

begin
writeln('Start timer...');
a := 3.0;
b := 5.0;
for i := 1 to 100 do
   for j := 1 to 25 do
      c := a+b;
writeln('Stop timer.');
end.
```

Here are the results of the timing tests. All times are in seconds. The first column shows the operation that was performed in the loop. The raw time column shows the number of seconds it takes to execute the program. For real operations, the value has been multiplied by 100 to account for the smaller number of operations done. This means the times shown are for loops of 10000 and 25, not 100 and 25. The time per operation is the raw time minus the time for an empty loop divided by the number of times the loop executed. The ratio shows how much longer it takes to do the real operation as compared to the integer operation. For example, it takes 489 times longer to add the two real numbers than it takes to add the same two numbers if they are integers.

operation	raw time	time per operation	real to integer ratio
(none)	2.61	0.00001044	
3+5	4.35	0.00000696	
3-5	4.35	0.00000696	
3 div 5	61.63	0.00023608	
3*5	59.41	0.00022720	
3.0+5.0	853.67	0.00340423	489
3.0-5.0	844.00	0.00336556	484
3.0/5.0	1266.00	0.00505356	21
3.0*5.0	892.00	0.00355756	16

Solution to problem 3.4.

```
program Dice(input, output);

const
   sides = 6;                          {# of sides on the dice}
   numDice = 2;                        {# of dice to throw}

var
   i,j: integer;                       {input value; loop counters}

begin
write('Random number seed =');
readln(i);
seed(i);

writeln;
for i := 1 to 20 do begin
   for j := 1 to numDice do
      write(RandomInteger mod sides + 1);
   writeln;
   end; {for}
end.
```

Solution to problem 3.5.

While the program draws enough dots to completely blacken the area, the dots are drawn in random locations. Occasionally, a dot will be drawn where one already exists. Of course, every time this happens, you will end up with one dot that doesn't get filled in.

```
{ Gradually fill an area of the screen with black dots }

program Snow(input, output);

uses Common, QuickDrawII;

const
   max = 11651;                        {# of dots to fill}
```

```
var
   i: integer;                          {loop counter}
   x,y: integer;                        {coordinates of the point}

begin
seed(5463);                             {initialize for RandomInteger}
SetPenMode(0);                          {set up for graphics}
SetSolidPenPat(0);
SetPenSize(1,1);

for i := 1 to max do begin
   x := RandomInteger mod 191 + 10;
   y := RandomInteger mod 61 + 10;
   MoveTo(x,y);
   LineTo(x,y);
   end; {for}
end.
```

Solution to problem 3.6.

```
{ Gradually fill an area of the screen with colored dots }

program ColoredSnow(input, output);

uses Common, QuickDrawII;

const
   max = 11651;                         {# of dots to fill}

var
   i: integer;                          {loop counter}
   x,y: integer;                        {coordinates of the point}

begin
seed(5463);                             {initialize for RandomInteger}
SetPenMode(0);                          {set up for graphics}
SetPenSize(1,1);

for i := 1 to max do begin
   SetSolidPenPat(RandomInteger mod 4);
   x := RandomInteger mod 191 + 10;
   y := RandomInteger mod 60 + 10;
   MoveTo(x,y);
   LineTo(x,y);
   end; {for}
end.
```

Solution to problem 3.7.

first set	1 2 3 4
second set	1 2 5 6
third set	1 0 3 4
fourth set	1 2 4 5
fifth set	1 2 3 4

Solution to problem 3.8.

```
{ Read a real and write it to the screen. }

program ReadReal(input, output);

var
   i: real;

begin
readln(i);
writeln(i);
end.
```

Solution to problem 3.9.

```
{ Car Payment Calculator }

program BuyACar(input,output);

var
   APR: real;                    {annual percentage rate}
   payment: real;                {monthly payment}
   month: integer;               {number of months}
   principal: real;              {amount left to pay}
   minPayment: real;             {minimum allowed payment}

begin
month := 0;                      {no payments made, yet}
write('Cost of the car      ?');  {get the cost of the car}
readln(principal);
write('Annual percentage rate?'); {get the interest rate}
readln(APR);
write('Monthly payments     ?');  {get the payment}
readln(payment);
minPayment :=                    {make sure the payment is big enough}
   principal*APR/100.0/12.0;
```

```
if payment <= minPayment then
   writeln('Your minimum payment is ', minPayment:1:2)
else
   while principal > 0.0 do begin {keep going until we're out of debt}
      month := month+1;           {it's a new month}
      principal := principal      {add in this month's interest}
         + principal*APR/100.0/12.0;
      principal :=                {make the payment}
         principal-payment;
      writeln(month:4, principal:10:2); {write the status}
      end; {while}
end.
```

Solution to problem 3.10.

This problem is solved in two steps, developing the program as we go. Both of the steps are shown, so you can see how I broke the problem up. Your solution should resemble the second program.

part one

```
program Bounce(input,output);

uses Common, QuickDrawII;

const
   maxX = 200;
   maxY = 80;

var
   i: integer;                         {loop counter}
   x,y: integer;                       {coordinates of the point}

begin
SetPenMode(0);                         {set up for graphics}
SetPenSize(6,2);

x := 0;                                {initialize the ball position}
y := 0;

for i := 1 to 50 do begin              {animate the ball}
   SetSolidPenPat(3);
   MoveTo(x,y);
   LineTo(x,y);
   x := x+1;
   y := y+1;
   SetSolidPenPat(0);
   MoveTo(x,y);
   LineTo(x,y);
   end; {for}
end.
```

part two

```
program Bounce(input,output);

uses Common, QuickDrawII;

const
   maxX = 316;
   maxY = 83;

var
   i,j: integer;                        {loop counters}
   iter: integer;                       {# of movements of the ball}
   x,y: integer;                        {coordinates of the point}
   xSpeed,ySpeed: integer;              {speed the ball}

begin
SetPenMode(0);                          {set up for graphics}
SetPenSize(6,2);

write('Start x     :');                 {initialize the ball position}
readln(x);
write('Start y     :');
readln(y);
write('X speed     :');                 {initialize the ball speed}
readln(xSpeed);
write('Y speed     :');
readln(ySpeed);
write('iterations  :');                 {initialize the loop counter}
readln(iter);

if x < 0 then                          {restrict the initial ball position}
   x := 0
else if x > maxX then
   x := maxX;
if y < 0 then
   y := 0
else if y > maxY then
   y := maxY;
```

```
for i := 1 to iter do begin            {animate the ball}
   for j := 1 to 10000 do ;              {pause for a bit}
   SetSolidPenPat(3);                  {erase the ball}
   MoveTo(x,y);
   LineTo(x,y);
   x := x+xSpeed;                      {move in the x direction}
   if x < 0 then begin                 {check for off the edge}
      x := 0;
      xSpeed := -xSpeed;
      end {if}
   else if x > maxX then begin
      x := maxX;
      xSpeed := -xSpeed;
      end; {else}
   y := y+ySpeed;                      {move in the y direction}
   if y < 0 then begin                 {check for off the edge}
      y := 0;
      ySpeed := -ySpeed;
      end {if}
   else if y > maxY then begin
      y := maxY;
      ySpeed := -ySpeed;
      end; {else}
   SetSolidPenPat(0);                  {draw the ball in the new spot}
   MoveTo(x,y);
   LineTo(x,y);
   end; {for}
end.
```

Lesson Four
Subroutines

Subroutines Avoid Repetition

In the first few lessons of this course, all of the programs we are writing are fairly short. Many useful programs are short, but as you start to make your programs more sophisticated, the programs will get longer and longer. A simple game on the Apple IIGS, for example, is generally 1,000 to 3,000 lines long; most of the programs we have written so far are 20 to 60 lines long. As the size of your programs increase, you will need some new concepts and tools to write the programs. One of the most important of these is the subroutine.

For our first look at subroutines, we will start with the program in listing 4.1 that draws three rectangles on the graphics screen, filling each with a different color.

If you look at this program closely, you will see that there is very little difference between the parts that draw the green and purple rectangles. In fact, if we put the coordinates of the rectangles in variables called left, right, top and bottom, and put the color in a variable called color, we could use exactly the same lines of code to draw the green and purple rectangles. The code would look like listing 4.2.

While we don't really need to redraw the outline of the square for the black square, the same code could even be used to draw the black square. A few extra lines get executed when the outline is drawn (the outline is black, and so is the color that is filled in), but the same code could be used. One of the of the most common uses for a subroutine is just this situation. When your program needs to do essentially the same thing in several different places, you can write a subroutine to do the thing, and call it from more than one place. Let's try this in a program, shown in listing 4.3, and then look at what is happening in detail.

The Structure of a Procedure

The subroutine itself starts with the reserved word procedure. The procedure statement looks a little like a program statement, and the similarities are no accident. A procedure is, in many ways, a little program imbedded inside of your main program. Right after the reserved word procedure is the name of the procedure; ours is called Rectangle. You use this name in the rest of your program whenever you want to call the subroutine. ("Calling" a procedure is what programmers say when they mean that you want to execute the statements in the procedure.) Just like the program statement, the procedure statement ends with an optional parameter list inclosed in parenthesis, followed by a semicolon.

The parameter list is more complicated than the one in the program statement, so we will need to spend a little more time talking about it. In our first subroutine, the parameter list looks like this:

```
left, right, top, bottom, color: integer
```

It is no accident that this looks suspiciously like a variable declaration. In fact, of you put var before the list and a semicolon after it, the compiler would be happy to take this parameter list as a variable declaration. What the parameter list actually does, in fact, is to define these variables within the subroutine. Any statement within the procedure can use these variables. You can change them using an assignment statement, or use them in an expression, as we do in our program. A very important point to keep in mind, though, is that the variables actually go away after you leave the subroutine. We will see this more clearly in a moment with the debugger.

The procedure statement forms a sort of model that tells us how to call the procedure, as you can see by comparing the procedure statement in listing 4.4 with a procedure call from our sample program.

On the top line, you see the procedure statement. The second line shows a call to the procedure, with spaces inserted to line up the matching components. Pascal knows you are calling the procedure from the name itself. Since the compiler has already seen the definition of the procedure, it knows that Rectangle is a procedure. The compiler also knows that the procedure needs five integer parameters. It expects them to appear after the procedure name, enclosed in parenthesis, and separated by commas. If you forget one of the these parameters, put in too many, or use a parameter that isn't an integer, the compiler will complain.

When the procedure is called, the compiler starts by assigning the values you put in the parameter list to the variables you defined in the parameter list. In effect, for the call we are using as an example, the compiler does the following five assignments in listing 4.5 before the first statement of the procedure is executed.

Listing 4.1

```
program Rectangles;

uses Common, QuickDrawII;

var
   i:integer;                    {loop variable}

begin
SetPenMode(0);                   {set up for graphics}
SetSolidPenPat(0);
SetPenSize(3,1);

for i := 10 to 60 do begin       {draw a black rectangle}
   MoveTo(10,i);
   LineTo(250,i);
   end; {for}

SetSolidPenPat(1);               {draw a green rectangle}
for i := 31 to 49 do begin
   MoveTo(220,i);
   LineTo(270,i);
   end; {for}
SetSolidPenPat(0);               {outline it in black}
MoveTo(220,30);
LineTo(220,50);
LineTo(270,50);
LineTo(270,30);
LineTo(220,30);

SetSolidPenPat(2);               {draw a purple rectangle}
for i := 41 to 79 do begin
   MoveTo(50,i);
   LineTo(300,i);
   end; {for}
SetSolidPenPat(0);               {outline it in black}
MoveTo(50,40);
LineTo(50,80);
LineTo(300,80);
LineTo(300,40);
LineTo(50,40);
end.
```

When the procedure starts, then, the variables from the parameter list already have an initial value.

A moment ago, I mentioned that a procedure is something like a miniature program within your main program. At this point, we can see just how true that is. The statements in the procedure, like the statements in the program itself, are sandwiched between a begin and end. The only difference is the end of a program is followed by a period, while the end of a procedure is followed by a semicolon. In a program, you have learned to put constant and variable declarations before the begin. You do exactly the same thing in a

Listing 4.2

```
SetSolidPenPat(color);          {draw a rectangle}
for i := top+1 to bottom-1 do begin
   MoveTo(left,i);
   LineTo(right,i);
   end; {for}
SetSolidPenPat(0);              {outline it in black}
MoveTo(left,top);
LineTo(left,bottom);
LineTo(right,bottom);
LineTo(right,top);
LineTo(left,top);
```

Listing 4.4

```
procedure Rectangle(left, right, top, bottom, color: integer);

          Rectangle(10,   250,   10,  60,     0);
```

Listing 4.5

```
left := 10;                       {in effect, what the compiler does}
right := 250;
top := 10;
bottom := 60;
color := 0;
```

procedure. In the Rectangle procedure, in fact, you will find the variable i defined for use in a for loop. Like the parameters, the variables and constants defined within the procedure vanish after the end of the procedure. The only thing you can access from the program is the procedure itself.

How Subroutines are Executed

There are some subtle points about how procedures are executed, how variables are created, and how they go away that I have mentioned, but that may not have sunk in so far. To drive these points home, we will fire up the debugger.

If you haven't already tried the program, go ahead and type it in now, and run it to see what it does.

The first thing we will do with the debugger is to look at what happens when a procedure is called. Pull down the debug menu and select the Step command to start the program. By now, you are probably familiar with the arrow appearing on the first statement in the program. Step up to the point where the arrow is on the first call to the Rectangle procedure (Figure 4.1). The next statement executed is the call to the procedure itself. When you step, the arrow jumps to the first line of the subroutine (Figure 4.2). As you continue to step, the statements in the procedure are executed just as they would be if they were in a program. After the last statement is executed, you return to the first statement after the procedure call in the main program.

The point of all of this is that the procedure call transfers control to the statements in the procedure. The procedure acts just like a small program, executing each statement in turn. When all of the statements in the procedure have been executed, the program continues on with the first statement after the procedure.

You can define variables in a procedure by declaring them as parameters or using a var declaration. These variables are created when the procedure is called, and vanish when the procedure has finished executing. To see this, stop the program, or use Trace

Listing 4.3

```
program Rectangles;

uses Common, QuickDrawII;


   procedure Rectangle(left, right, top, bottom, color: integer);

   { This subroutine draws a colored rectangle and outlines it    }
   { in black.                                                    }
   {                                                              }
   { Parameters:                                                  }
   {    left,right,top,bottom - edges of the rectangle            }
   {    color - interior color of the rectangle                   }

   var
      i: integer;              {loop variable}

   begin {Rectangle}
   SetSolidPenPat(color);       {draw a rectangle}
   for i := top+1 to bottom-1 do begin
      MoveTo(left,i);
      LineTo(right,i);
      end; {for}
   SetSolidPenPat(0);           {outline it in black}
   MoveTo(left,top);
   LineTo(left,bottom);
   LineTo(right,bottom);
   LineTo(right,top);
   LineTo(left,top);
   end; {Rectangle}


begin
SetPenMode(0);                  {set up for graphics}
SetSolidPenPat(0);
SetPenSize(3,1);

Rectangle(10,250,10,60,0);      {draw a black rectangle}
Rectangle(220,270,30,50,1);     {draw a green rectangle}
Rectangle(50,300,40,80,2);      {draw a purple rectangle}
end.
```

or Go to finish it up. Bring up the variables window, and move it over the shell window (which we aren't using at the moment). Step to the first line of the program. Since there are no variables in the main part of the program, you will find that you cannot look at a variable. You might try looking at the variable left, from the procedure's parameter list, or i, defined with the procedure, to see this. The debugger will tell you that no such variable exists.

Continue stepping until you get to the first line of the procedure. Watch the variables window as you enter the procedure: the name at the top of the

variables window will change from ~_PASMAIN (the debugger's name for the main part of the program) to RECTANGLE. At this point, you can look at any of the variables from the parameter list, or you can look at the loop variable, i. It is sometimes entertaining to look at i before the loop has been executed. Since i has not been assigned a value, the number you see is just whatever value the memory location the compiler is using for i happened to contain before the procedure was called. It brings home the point that a variable should never be used before you assign a value to it.

Something else that is new is that one of the arrows is now black. While you are looking at the variables from the procedure, you cannot look at the variables in the main program. Technically speaking, these variables are located in separate areas, called stack frames, and the debugger only lets you look at one stack frame at a time. To look at the variables from the main program, you click on the up arrow to move "up" one stack frame. To look at the variables from the procedure again, you click on the down arrow to move "down" one stack frame.

If you have been using the arrows to switch between the stack frames, move back to the RECTANGLE procedure, so you are looking at the variables from the procedure. Continue stepping until you leave the procedure. As soon as you do, the variables window shifts back to ~_PASMAIN. The arrow can no longer be used to look at the variables from the procedure. In fact, the variables literally do not exist anymore; the memory the compiler was using for the variables while the procedure was executing can be used for other purposes. When you step into the procedure again, new memory is allocated for the variables. By chance, it might happen to be the same memory that was used on the last call, but that isn't something you can count on. One side effect of this is that the value of a variable does not usually survive between procedure calls. For example, the value of the variable i may or may not be the same as it was at the end of the last call; certainly, you should not write a program that depends on the value being the same. Later, we will learn how to write programs when you need to keep a value between calls.

Some Formatting Conventions

A program with one procedure isn't likely to be too

Listing 4.6

1. A description of the procedure, telling what the procedure does.

```
{ This subroutine draws a colored rectangle and outlines it    }
{ in black.                                                    }
```

2. A parameter declaration section that describes the meaning of each parameter that appears in the procedure statement.

```
{ Parameters:                                                  }
{    left,right,top,bottom - edges of the rectangle            }
{    color - interior color of the rectangle                   }
```

3. A variables section that lists any global variables used by the procedure. Our program did not use any global variables, but the sample below shows how I would define one.

```
{ Variables:                                                   }
{    x,y - the current location of the ball                    }
```

4. A notes section that gives any pertinent information about how the subroutine is implemented, and describes any unusual properties of the subroutine. Our procedure did not need a notes section, but the sample below shows what one looks like.

```
{ Notes:                                                       }
{    1. For a description of the insertion sort, see           }
{       "Algorithms + Data Structures = Programs," p. 85.      }
```

Listing 4.7

```
Rectangle(10,250,10,60,0);      {draw a black rectangle}
Rectangle(220,270,30,50,1);     {draw a green rectangle}
Rectangle(50,300,40,90,2);      {draw a purple rectangle}
```

confusing, but as our programs use more and more procedures, there are some formatting and commenting conventions that will help make the programs easier to read. One convention I use in virtually every programming language has to do with the way I choose names. If you look at the sample program, you will see that all variable names start with a lowercase letter, while the program name and the name of the procedure start with a capitol letter. This helps remind me about the kind of identifier I am looking at when I read a program.

The procedure itself is indented three spaces from the rest of the program. As you will see later, you can define a procedure within another procedure. This may sound strange, but you will learn how to organize your programs by imbedding one procedure inside another. To help see which procedures are defined within others, I indent each procedure three spaces from the procedure or program in which it is defined.

By far the most important convention, though, is how the procedure is commented. I always put the name of the procedure after the begin and end that mark the body of the procedure. When you start to nest procedures, this will help you see the structure of the program more clearly. This is such an important aid that some languages based on Pascal can actually check the name after a begin or end to make sure it matches the name of the procedure!

More important still is the block of comments that appear right after the procedure statement, and before any variables. This block of comments tells what the procedure does. I use a very rigid format with up to four sections to comment a procedure. These sections are shown in listing 4.6.

As with the other formatting and commenting conventions mentioned in this course, there are many correct ways to comment and format a procedure that are different from the ones I have shown you. The important point isn't which one you use; the important point is to find one you like that supplies the same information and use it consistently.

Procedures Let You Create New Commands

We have seen that a subroutine can be used to take a series of similar, repetitious commands and place them in a single subroutine, making our program shorter and easier to understand. Subroutines can also be used to create new commands, which helps organize the program, making it easier to read. The Rectangle subroutine we have already created is one example. Once you know what the Rectangle procedure does, it is a lot easier to read the lines in listing 4.7 than it was to read the original program. The idea of using subroutines to neatly package our program is a very powerful one. It takes some getting used to, but once mastered, the technique will help you write programs faster and find errors in programs easier.

There is another advantage, too. Most people tend to write a few general types of programs. For example, an engineer might write several programs to deal with complicated matrix manipulation, but never deal with graphics to any great degree. Another person might use his computer to write adventure games. Any time you start writing programs that fall into broad groups like this, you will find that there are sections of your program that get repeated over and over again. By packaging these ideas into subroutines, you can quickly move the proper sections of code from one program to another.

As an example, let's look at a small section of code that seems to appear at the beginning of nearly all of our graphics programs.

```
SetPenMode(0);  {set up for graphics}
SetSolidPenPat(0);
SetPenSize(3,1);
```

We can package these three lines into a procedure called InitGraphics, shown in listing 4.8.

With this new procedure, our program becomes even easier to read, as you can see in listing 4.9.

It may not be obvious yet, but there is still one more advantage to packaging even these three simple commands into a procedure. At some point, you may decide that you want to set up the graphics screen a bit differently. For example, you will eventually learn to resize the graphics window, or quickly color the entire window with a background color. With the graphics initialization in a neat little package, it will be easy to redo the package and quickly update all of your programs. You will also learn faster ways to color in a rectangle. If all of your programs use the Rectangle

Listing 4.8

```
procedure InitGraphics;

{ Standard graphics initialization.                          }

begin {InitGraphics}
SetPenMode(0);              {pen mode = copy}
SetSolidPenPat(0);          {pen color = black}
SetPenSize(3,1);            {use a square pen}
end; {InitGraphics}
```

Listing 4.9

```
begin
InitGraphics;                   {set up the graphics window}
Rectangle(10,250,10,60,0);      {draw a black rectangle}
Rectangle(220,270,30,50,1);     {draw a green rectangle}
Rectangle(50,300,40,80,2);      {draw a purple rectangle}
end.
```

procedure, you can easily update the procedure, quickly bringing all of your programs up to date. If the code to draw rectangles is scattered throughout your programs, though, it would be a daunting task to change them all, simply because it would be hard to find all of the places that need to be changed.

Problem 4.1. One use of the Rectangle procedure is to draw game boards. For example, a board for a Reversi game would consist of eight rows and eight columns of green squares with black outlines. A chess or checker board can be drawn as eight rows and eight columns of alternating black and white squares.

Use the Rectangle procedure to draw a checker board in the graphics window. Make each square 20 pixels wide and 8 pixels high, with the top left square at 30,12.
Hint: Use one for loop nested within another to loop over the rows and columns, like this:

```
for row := 1 to 8 do
   for column := 1 to 8 do
      <draw a square>
```

This way, you can locate the top of each square as (row-1)*8+12. The bottom of each square will be at row*8+12. The same idea can be used to find the left and right edge of each square.

More About Debugging Procedures

By now, you have probably become good friends with the source-level debugger. By stepping through a program, you can quickly see why a program does not work, or how a program does what it does. You may have also noticed that single stepping through a long program can get very tedious.

There are two debugger features that are designed specifically to help debug programs that use procedures. The first is called "Go to Next Return." Like the other debug commands, it is found in the Debug menu. To see how it works, start by single stepping through our example program until you are inside of one of the procedures. In a real debugging session, you might step carefully through the first few lines of a subroutine, tracking some action. Having learned what you need to know, you now want to continue stepping through the main program. Rather than single stepping through the remainder of the subroutine, you can choose the Go to Next Return command. The program is executed at full speed until you return from the subroutine, after which you are returned to single step mode.

Once you are back in the main program, you may find many more calls to the same procedure. In our sample, for example, there are three calls to the

Rectangle procedure. Having satisfied yourself that the procedure works, there is no need to step through it for the next two calls. The Step Through command can be used in situations like this. The Step Through command executes the entire procedure at full speed, returning you to single step mode after you get back to the main program.

Functions are Procedures that Return a Value

In the last lesson, we used a pseudo-random number generator in several programs to create simulations. One common theme in these simulations was to restrict the range of the random number. For example, in our number guessing game, we selected numbers from 1 to 100. To roll dice, on the other hand, we used the same idea to select a random number from 1 to 6. With what we have learned about procedures, it would seem that this would be an ideal candidate for packaging. There is a problem, though. The whole point of the random number code is to produce a number. We need a way to get a value back from the procedure. When we need a value back, Pascal gives us a new flavor of the procedure, called a function. A function is just a procedure that can return a single value. The program in listing 4.10 demonstrates this idea by packaging our random number generator.

There are really only two differences in the way you write a procedure and function. The first shows up in the function header, which starts with the reserved word function, rather than the reserved word procedure. The function returns a value. It is possible for this value to be an integer, a real number, or one of several other types you will learn about later. Naturally, you have to tell the compiler what type of value the function returns. You do this just like you would for a variable, by following the name of the function (and the parameter list, if there is one) with a colon and the type. In the case of our RandomValue function, the type is integer.

At some point, you need to specify what value the function should return. This is the second difference between a function and a procedure. Somewhere in the function, you need to assign a value to the name of the function itself. You can do this in more than one place, if you like, using if statements to determine which assignment decides the value of the function. You can also assign a value to the function more than once, perhaps starting it off with an initial value that may or may not get changed later. You must assign a value to the function at least one time, however. If you don't, the value returned by the function is not predictable.

The Peculiar Definition of an Error

There are several places in the Pascal language where you are supposed to do something, but it may be very difficult and time consuming for a practical implementation of Pascal to make sure you follow the rules. It doesn't make any sense at all, but the document that defines Pascal calls these errors. Failing to assign a value to a function before returning from the function is one of these errors. It is perfectly legal for a Pascal compiler to generate a run-time error, stopping your program in its tracks if you return from a function without setting the return value. In practice, though, this requires extra code, so most compilers, including ORCA/Pascal, do not check to be sure you have set a value. On the other hand, ORCA/Pascal can check while the program is being compiled to make sure that there is at least one assignment to the function name somewhere in the function. While this doesn't guarantee that the function will set the value in all cases, it does help to find some errors. Other compilers might not even make this check, though. Ultimately, it is up to you to make sure the function value is set.

You can use a function anywhere you could use a value within the Pascal language. In our program, we use the function in the statement

```
sum := sum+RandomValue(sides);
```

When the program gets to this statement, it calls the function. The function calculates a value, and returns it. The value is added to sum, just as the number 4 would be in the statement

```
sum := sum+4;
```

Problem 4.2. You can use a function anywhere you can use a value in Pascal. In particular, you can use the RandomValue function to decide how many times to loop through a for loop, like this:

```
for i := 1 to RandomValue(20) do
   ...
```

You can also use a function to set the value of a parameter for another procedure or function call.

Use these ideas to create a program that will draw a random number of rectangles, not to exceed 30, in the graphics window. The rectangles should have a left and right value between 1 and 316, and a top

Listing 4.10

```
{ This program rolls two dice 20 times.                          }

program RollEm(output);

const
   sides = 6;                  {sides on a die}
   rolls = 20;                 {times to roll}
   numDice = 2;                {number of dice to roll}

var
   i,j: integer;               {loop variables}
   sum: integer;               {number of spots showing}


   function RandomValue(max: integer): integer;

   { Return a pseudo-random number in the range 1..max.          }
   {                                                             }
   { Parameters:                                                 }
   {    max - largest number to return                           }
   {    color - interior color of the rectangle                  }

   begin {RandomValue}
   RandomValue := (RandomInteger mod max) + 1;
   end; {RandomValue}

begin
Seed(1234);                    {initialize the random number generator}
for i := 1 to rolls do begin
   sum := 0;
   for j := 1 to numDice do
      sum := sum+RandomValue(sides);
   writeln(sum);
   end; {for}
end.
```

and bottom value between 1 and 83. Use an if statement and a temporary variable to make sure the left side is less than or equal to the right side, and that the top is less than or equal to the bottom, like this:

```
if left > right then begin
   temp := left;
   left := right;
   right := temp;
   end; {if}
```

If you are not certain why these statements will insure that left is less that right, try tracing the code by hand or with the debugger for several values of left and right.

Finally, the color of the rectangle should be chosen at random, and should be in the range 0 to 3. You can get a value from 0 to 3 from the RandomValue function like this:

```
RandomValue(4)-1
```

The call to RandomValue to get the color of the rectangle should appear in the parameter list of the call to Rectangle.

Var Parameters

There are some places where we want to package some code that changes more than one value. A good example of this is the ball bouncing program from the last problem in Lesson 3. It would be nice to package the code that updates the position of the ball into a function, and return the new position of the ball. There is a problem, though. A function can only return one value, but we need to update both an X and Y coordinate.

Pascal has a special kind of parameter to handle this situation. It is called a var parameter. The parameters we have used so far are called value parameters. With a value parameter, calling the procedure or function creates an entirely new variable, and copies the value you pass into the new variable. If you change the variable inside the subroutine, you will only change the local copy of the variable, not the original value. A var parameter does not create a new variable. Instead, it gives the variable a new name that is used inside the subroutine. This name can be the same as, or different from, the name of the variable that is passed when the subroutine is called. With a var parameter, changing the value inside the subroutine also changes the value outside the subroutine.

This is a subtle and sometimes confusing point, but it is a very important one to understand. To get a good grasp on the difference between value parameters and var parameters, we will rewrite our bounce program from Lesson 3. The source code appears in Listing 4.11.

In our program, the MoveBall procedure is used to update the position of the ball on the screen. We pass four values to the MoveBall procedure; the current x and y position of the ball, and the current velocity of the ball. Each of these four variables can be changed by the procedure, so they are all declared as var parameters. As you can see, the only difference between the way a var parameter is declared and the way a value parameter is declared is the reserved word var right before the list of variables.

Run the program in step mode, and step up to the first line of the MoveBall procedure. Bring up the variables window, and take a look at the values for x, y, vx and xy in the procedure. You will see a strange value in the display. This is because the compiler actually passes the location where the variable is, rather than the variable itself, when the variable is a var parameter. To see the variable, place a ^ character after the name of the variable in the variables window. (The reason for this will become clear in a later lesson. For now, you just need to know how to see the value of the variable when it is a var parameter.) As you step through the procedure, changing the values for the parameters, switch back to the main program (~_PASMAIN) to check the value of the corresponding variables in the program. As you can see, the variables in the program change at the same time as the variables in the subroutine.

Now try the same program with var removed from the parameter list, like this:

```
procedure MoveBall(x,y,vx,xy: integer);
```

The program won't work, of course. When the value of the parameter is changed, the local copy changes, but the corresponding value in the main program does not.

There are some restrictions on what you can pass as a var parameter that make sense if you think about how var parameters work. With a var parameter, the only thing you can pass is a variable of the same type as the parameter. This is because the subroutine can assign new values to the parameter, which is really just a new, local name for the original variable that is passed. A value parameter, on the other hand, assigns a value to a new variable when the subroutine is called. Because of that, you can pass anything as a value parameter that could be assigned to a variable of the same type.

For example, any of the following could be used to set the initial value of an integer parameter, but all are illegal as var parameters.

```
4
RandomValue(4)
RandomInteger
a+17
```

Problem 4.3. By using our neatly packaged subroutine, you can quickly write a program to bounce more than one ball around on the screen. Modify the sample program to bounce 10 balls simultaneously.

Use the RandomValue function to choose the initial positions and speeds of the balls. Move the balls 100 times.

Because you are bouncing 10 balls instead of 1, you will not need a delay loop.

Listing 4.11

```
program Bounce(input,output);

uses Common, QuickDrawII;

const
   maxX = 316;
   maxY = 83;

var
   i,j: integer;                        {loop counters}
   iter: integer;                       {# of movements of the ball}
   x,y: integer;                        {coordinates of the point}
   xSpeed,ySpeed: integer;              {speed the ball}


   procedure MoveBall(var x, y, vx, vy: integer);

   { Move a ball in the graphics window.  If the ball   }
   { hits one of the sides (defined by the global       }
   { constants maxX and maxY), the direction of the     }
   { ball is changed.                                   }
   {                                                    }
   { Parameters:                                        }
   {    x,y - position of the ball                      }
   {    vx,vy - velocity of the ball                    }

   begin {MoveBall}
   SetSolidPenPat(3);                   {erase the ball}
   MoveTo(x,y);
   LineTo(x,y);
   x := x+vx;                           {move in the x direction}
   if x < 0 then begin                  {check for off the edge}
      x := 0;
      vx := -vx;
      end {if}
   else if x > maxX then begin
      x := maxX;
      vx := -vx;
      end; {else}
   y := y+vy;                           {move in the y direction}
   if y < 0 then begin                  {check for off the edge}
      y := 0;
      vy := -vy;
      end {if}
else if y > maxY then
   y := maxY;
```

(continued…)

(continuation)

```
   else if y > maxY then begin
      y := maxY;
      vy := -vy;
      end; {else}
   SetSolidPenPat(0);                  {draw the ball in the new spot}
   MoveTo(x,y);
   LineTo(x,y);
   end; {MoveBall}


begin
SetPenMode(0);                         {set up for graphics}
SetPenSize(6,2);

write('Start x     :');                {initialize the ball position}
readln(x);
write('Start y     :');
readln(y);
write('X speed     :');                {initialize the ball speed}
readln(xSpeed);
write('Y speed     :');
readln(ySpeed);
write('iterations  :');                {initialize the loop counter}
readln(iter);

if x < 0 then                          {restrict the initial ball position}
   x := 0
else if x > maxX then
   x := maxX;
if y < 0 then
   y := 0

for i := 1 to iter do begin            {animate the ball}
   for j := 1 to 10000 do ;            {pause for a bit}
   MoveBall(x,y,xSpeed,ySpeed);        {move the ball}
   end; {for}
end.
```

Variable Scope

There was an interesting detail in the last program that is worth looking into a bit further. In the const section of the main program, two constants were use to keep track of the size of the graphics window.

```
const
   maxX = 316;  {width of the graphics window}
   maxY = 83;  {height of the graphics window}
```

It makes perfect sense to use a constant to remember the size of the graphics window. After all, the size of the graphics window can be changed. If you look in the MoveBall procedure, though, you will find that these constants are used. You have already learned that a variable or parameter defined inside of a subroutine cannot be used outside of the subroutine.

This program shows that the reverse is not true. You can access a variable or constant defined in the main part of the program from within a procedure or function.

The name we use to describe this idea is the *scope* of a variable. The scope of a variable, simply put, is the range of statements that can access the variable. For a variable defined in a procedure, or for the parameter of a procedure, the scope starts when the variable is declared and continues to the end of the subroutine. The scope of a variable in the program itself starts with the declaration of the variable, and continues to the end of the program. What this means is that you can use a variable defined in the program in any procedure or function in the program.

There is an exception to this rule. You can use a variable name within a subroutine even if the variable has already been defined in the program. When you reuse a variable name, the new variable is used within the subroutine, and the old variable is again used outside of the subroutine. The program in listing 4.12 shows this idea, using the variable i as a for loop variable both inside and outside of a procedure.

Listing 4.12

```
{ This program prints all of the numbers that divide evenly      }
{ into another number.                                           }

program Factor(output);

const
   max = 20;                    {largest number to check}

var
   i: integer;                  {loop variable}


   procedure Factor(n: integer);

   { Print the factors of n.                                      }
   {                                                              }
   { Parameters:                                                  }
   {    n - number to check                                       }

   var
      i: integer;               {loop variable}

   begin {Factor}
   write(n: 4, ': ');
   for i := 2 to n-1 do
      if (n mod i) = 0 then
         write(i: 4);
   writeln;
   end; {Factor}


begin
for i := 1 to max do
   Factor(i);
end.
```

Lesson Four
Solutions to Problems

Solution to problem 4.1.

```
{ Draw a checker board                                              }

program Board;

uses Common, QuickDrawII;

var
   row,column: integer;         {position of the square}
   color: integer;              {color of the square}


   procedure InitGraphics;

   { Standard graphics initialization.                              }

   begin {InitGraphics}
   SetPenMode(0);               {pen mode = copy}
   SetSolidPenPat(0);           {pen color = black}
   SetPenSize(3,1);             {use a square pen}
   end; {InitGraphics}


   procedure Rectangle(left, right, top, bottom, color: integer);

   { This subroutine draws a colored rectangle and outlines it      }
   { in black.                                                      }
   {                                                                }
   { Parameters:                                                    }
   {    left,right,top,bottom - edges of the rectangle              }
   {    color - interior color of the rectangle                     }

   var
      i: integer;               {loop variable}

   begin {Rectangle}
   SetSolidPenPat(color);       {draw a rectangle}
   for i := top+1 to bottom-1 do begin
      MoveTo(left,i);
      LineTo(right,i);
      end; {for}
   SetSolidPenPat(0);           {outline it in black}
   MoveTo(left,top);
   LineTo(left,bottom);
```

```
   LineTo(right,bottom);
   LineTo(right,top);
   LineTo(left,top);
   end; {Rectangle}


   procedure SwapColor;

   { Changes the color between black and white.                }
   {                                                           }
   { Variables:                                                }
   {    color - color to change                                }

   begin {SwapColor}
   if color = 3 then
      color := 0
   else
      color := 3;
   end; {SwapColor}


begin
InitGraphics;                  {set up the graphics window}

color := 3;                    {start with a white square}
for row := 1 to 8 do begin
   for column := 1 to 8 do begin
                               {draw one square}
      Rectangle(column*20+10, column*20+30, row*8+4, row*8+12, color);
      SwapColor;               {flip the color}
      end; {for}
   SwapColor;                  {flip the color}
   end; {for}
end.
```

Solution to problem 4.2.

```
{ Draw up to 30 random rectangles                              }

program Rectangles;

uses Common, QuickDrawII;

var
   top,bottom,left,right: integer; {position of the rectangle}
   i: integer;                  {loop variable}
   temp: integer;               {temporary; used for swapping}
```

```
procedure InitGraphics;

{ Standard graphics initialization.                          }

begin {InitGraphics}
SetPenMode(0);               {pen mode = copy}
SetSolidPenPat(0);           {pen color = black}
SetPenSize(3,1);             {use a square pen}
end; {InitGraphics}


function RandomValue(max: integer): integer;

{ Return a pseudo-random number in the range 1..max.         }
{                                                            }
{ Parameters:                                                }
{    max - largest number to return                          }
{    color - interior color of the rectangle                 }

begin {RandomValue}
RandomValue := (RandomInteger mod max) + 1;
end; {RandomValue}


procedure Rectangle(left, right, top, bottom, color: integer);

{ This subroutine draws a colored rectangle and outlines it  }
{ in black.                                                  }
{                                                            }
{ Parameters:                                                }
{    left,right,top,bottom - edges of the rectangle          }
{    color - interior color of the rectangle                 }

var
   i: integer;               {loop variable}

begin {Rectangle}
SetSolidPenPat(color);       {draw a rectangle}
for i := top+1 to bottom-1 do begin
   MoveTo(left,i);
   LineTo(right,i);
   end; {for}
SetSolidPenPat(0);           {outline it in black}
MoveTo(left,top);
LineTo(left,bottom);
LineTo(right,bottom);
LineTo(right,top);
LineTo(left,top);
end; {Rectangle}
```

```
begin
InitGraphics;                    {set up the graphics window}
Seed(6289);                      {initialize the random number generator}

for i := 1 to RandomValue(30) do begin {draw the rectangles}
   left := RandomValue(316);     {get the left, right edges}
   right := RandomValue(316);
   if left > right then begin    {make sure left > right}
      temp := right;
      right := left;
      left := temp;
      end; {if}
   top := RandomValue(83);       {get the top, bottom edges}
   bottom := RandomValue(83);
   if top > bottom then begin    {make sure top > bottom}
      temp := top;
      top := bottom;
      bottom := temp;
      end; {if}
                                 {draw the rectangle}
   Rectangle(left, right, top, bottom, RandomValue(4)-1);
   end; {for}
end.
```

Solution to problem 4.3.

```
program Bounce10;

uses Common, QuickDrawII;

const
   maxX = 316;
   maxY = 83;

var
   i: integer;                          {loop counter}
   x1,x2,x3,x4,x5,x6,x7,x8,x9,x10: integer; {x coordinates of the point}
   y1,y2,y3,y4,y5,y6,y7,y8,y9,y10: integer; {y coordinates of the point}
                                        {ball velocities}
   vx1,vx2,vx3,vx4,vx5,vx6,vx7,vx8,vx9,vx10: integer;
   vy1,vy2,vy3,vy4,vy5,vy6,vy7,vy8,vy9,vy10: integer;


   procedure InitGraphics;

   { Standard graphics initialization.                              }

   begin {InitGraphics}
   SetPenMode(0);                {pen mode = copy}
```

```
SetSolidPenPat(0);           {pen color = black}
SetPenSize(3,1);             {use a square pen}
end; {InitGraphics}


procedure MoveBall(var x, y, vx, vy: integer);

{ Move a ball in the graphics window.  If the ball   }
{ hits one of the sides (defined by the global       }
{ constants maxX and maxY), the direction of the     }
{ ball is changed.                                   }
{                                                    }
{ Parameters:                                        }
{    x,y - position of the ball                      }
{    vx,vy - velocity of the ball                    }

begin {MoveBall}
SetSolidPenPat(3);           {erase the ball}
MoveTo(x,y);
LineTo(x,y);
x := x+vx;                   {move in the x direction}
if x < 0 then begin          {check for off the edge}
   x := 0;
   vx := -vx;
   end {if}
else if x > maxX then begin
   x := maxX;
   vx := -vx;
   end; {else}
y := y+vy;                   {move in the y direction}
if y < 0 then begin          {check for off the edge}
   y := 0;
   vy := -vy;
   end {if}
else if y > maxY then begin
   y := maxY;
   vy := -vy;
   end; {else}
SetSolidPenPat(0);           {draw the ball in the new spot}
MoveTo(x,y);
LineTo(x,y);
end; {MoveBall}


function RandomValue(max: integer): integer;

{ Return a pseudo-random number in the range 1..max.          }
{                                                             }
{ Parameters:                                                 }
{    max - largest number to return                           }
```

```
   {    color - interior color of the rectangle                  }

   begin {RandomValue}
   RandomValue := (RandomInteger mod max) + 1;
   end; {RandomValue}


begin
InitGraphics;                  {set up the graphics window}
SetPenSize(6,2);               {use a large ball}
Seed(6289);                    {initialize the random number generator}

                               {set the initial ball positions}
x1 := RandomValue(maxX);   y1 := RandomValue(maxY);
x2 := RandomValue(maxX);   y2 := RandomValue(maxY);
x3 := RandomValue(maxX);   y3 := RandomValue(maxY);
x4 := RandomValue(maxX);   y4 := RandomValue(maxY);
x5 := RandomValue(maxX);   y5 := RandomValue(maxY);
x6 := RandomValue(maxX);   y6 := RandomValue(maxY);
x7 := RandomValue(maxX);   y7 := RandomValue(maxY);
x8 := RandomValue(maxX);   y8 := RandomValue(maxY);
x9 := RandomValue(maxX);   y9 := RandomValue(maxY);
x10 := RandomValue(maxX);  y10 := RandomValue(maxY);

                               {set the initial ball velocities}
vx1 := RandomValue(19)-10;  vy1 := RandomValue(7)-3;
vx2 := RandomValue(19)-10;  vy2 := RandomValue(7)-3;
vx3 := RandomValue(19)-10;  vy3 := RandomValue(7)-3;
vx4 := RandomValue(19)-10;  vy4 := RandomValue(7)-3;
vx5 := RandomValue(19)-10;  vy5 := RandomValue(7)-3;
vx6 := RandomValue(19)-10;  vy6 := RandomValue(7)-3;
vx7 := RandomValue(19)-10;  vy7 := RandomValue(7)-3;
vx8 := RandomValue(19)-10;  vy8 := RandomValue(7)-3;
vx9 := RandomValue(19)-10;  vy9 := RandomValue(7)-3;
vx10 := RandomValue(19)-10; vy10 := RandomValue(7)-3;

for i := 1 to 100 do begin     {animate the ball}
   MoveBall(x1,y1,vx1,vy1);    {move the balls}
   MoveBall(x2,y2,vx2,vy2);
   MoveBall(x3,y3,vx3,vy3);
   MoveBall(x4,y4,vx4,vy4);
   MoveBall(x5,y5,vx5,vy5);
   MoveBall(x6,y6,vx6,vy6);
   MoveBall(x7,y7,vx7,vy7);
   MoveBall(x8,y8,vx8,vy8);
   MoveBall(x9,y9,vx9,vy9);
   MoveBall(x10,y10,vx10,vy10);
   end; {for}
end.
```

Lesson Five
Arrays and Strings

Groups of Numbers as Arrays

Computers can deal with very large amounts of data. On the Apple IIGS, you can easily write programs that will deal with thousands of numbers, names, zip codes, or whatever. So far, though, the methods we have for dealing with these values are fairly limited. A database of a hundred friends, each of whom has a name, street address, a city, a state, and a zip code would be a daunting task if each value had to be placed in a separate variable.

One way we have to deal with large amounts of data is called an array. An array is a group of values, each of which is the same type. We use an index to determine which of the values we want to access at a given time.

For our first look at an array, let's do a simulation of rolling dice. We've done this several times before, on a small scale, but this time we're going to roll the dice 10000 times, and keep track of how many times we get a 2, how many times we get a 3, and so forth. We could, of course, use a separate variable for each of the totals, but that would get to be a bit tedious. Instead, we will use an array.

To define an array, you need to specify how many things you want in the array, and what kind they are. In our case, we are adding up the number of times a particular value shows up on a pair of dice. We can get any value from 2 to 12 from a pair of dice, so we need an array of eleven integers. Pascal lets you specify the low entry as well as the high entry, so we can get our eleven integers using subscripts from 2 to 12. The array is defined like this:

```
{number of spots showing}
totals: array[2..12] of integer;
```

Listing 5.1

```
{ This program simulates rolling dice.  It counts the number of   }
{ times each value appears, printing a summary after the run is   }
{ complete.                                                       }

program Dice(output);

const
   rolls = 10000;              {times to roll}

var
   totals: array[2..12] of integer; {number of spots showing}


   function Random(max: integer): integer;

   { Return a pseudo-random number in the range 1..max.           }
   {                                                              }
   { Parameters:                                                  }
   {    max - largest number to return                            }
   {    color - interior color of the rectangle                   }
```

(continued…)

(continuation)

```
      begin {Random}
      Random := (RandomInteger mod max) + 1;
      end; {Random}


      procedure Simulation;

      { Roll the dice rolls times, saving the result.                }
      {                                                              }
      { Variables:                                                   }
      {    totals - array holding the total number of rolls          }

      var
         i: integer;                   {loop variable}
         sum: integer;                 {# of spots for this roll}

      begin {Simulation}
      for i := 2 to 12 do              {set the totals to zero}
         totals[i] := 0;
      for i := 1 to rolls do begin
         sum := Random(6)+Random(6);   {roll the dice}
         totals[sum] := totals[sum]+1; {increment the correct total}
         end; {for}
      end; {Simulation}

      procedure WriteArray;

      { Write the results.                                           }
      {                                                              }
      { Variables:                                                   }
      {    totals - array holding the total number of rolls          }

      var
         i: integer;                   {loop variable}

      begin {WriteArray}
      writeln('spots':8, 'times':8);
      for i := 2 to 12 do
         writeln(i,totals[i]);
      end; {WriteArray}

   begin
   Seed(1234);                     {initialize the random number generator}
   Simulation;                      {do the dice simulation}
   WriteArray;                      {write the dice array}
   end.
```

To get at a particular entry in the array, we need to specify both the name of the array and the element of the array that we want to access. To set the first element of the array to zero, for example, we would write

```
totals[2] := 0;
```

The value between the brackets can be an expression as well as a specific integer. We can use this fact to set all of the values in the array to zero with a loop, rather than eleven separate assignment statements. The loop variable can then be used as the array index. This technique of using a loop to deal with all of the elements of an array as a group is one of the reasons arrays are so handy for large amounts of data.

```
for i := 2 to 12 do
   totals[i] := 0;
```

You can use an element of an array anywhere that you could use an integer variable. You can, for example, write an element of an array, use it in an expression, or pass it as a parameter to a procedure. There are very few cases, though, where you can use the entire array. You can't write an array using writeln, for example. We will explore when and how you can use an entire array as we get to know arrays better.

Before you run this program in listing 5.1, I want to let you know that it will take a long time. In fact, this program will run for over six minutes! We will look at the reasons in a moment.

When you run this program, you should be sure and trace through a few loops with the debugger to see how values are assigned to an array. You can't look at an entire array with the debugger, though. Instead, you must enter each element of the array individually. For example, to look at all of the elements of the totals array in this sample program, you need to enter totals[2], totals[3], and so forth as separate variables in the variables window. There isn't enough room in the variables window to display all of the elements of the array when the variables window first shows up, but you can use the grow box to expand the variables window, just as you would with a program window.

As I pointed out a moment ago, and as you have no doubt noticed, the program runs for over six minutes. In fact, it runs for 6 minutes, 12.01 seconds. While it is very true that this program is doing a fair amount of work, rolling two dice 10,000 times, that is still a very long time. To see why, pull down the Run menu and select the Compile dialog. Turn the debug code off, and run the program again. This time, the program runs in 8.84 seconds. That's quite a difference. In fact, that

The Byte

The text mentions a thing called a byte. A byte is the unit of storage used by the computer. On the Apple IIGS, and in fact on most modern computers, especially microcomputers, a byte is made up of eight bits. Each bit is like a little switch that can be set to on or off; these states are labeled one and zero.

For the most part, we don't deal directly with bytes in Pascal, since Pascal gives us much more natural ways of dealing with information. The only time we will discuss bytes is when we are talking abut how much memory it takes to do something.

Talking about bytes in terms of how many bits there are makes as much sense to most people as discussing sex in terms of birds and bees. The parts involved just aren't the ones we are familiar with. From the Pascal programers' viewpoint, a much more reasonable way to look at the byte is in terms of how many bytes it takes to hold certain kinds of values. An integer variable, for example, needs two bytes of storage. A character also uses two bytes of storage, but really only needs one; if you declare a packed array of characters, each character only uses one byte of storage. Real numbers and long integers each require four bytes of storage, while a double-precision real number needs eight bytes of storage.

To find out how big an array is in bytes, multiply the number of things in the array by the number of bytes required for one of the array variables. For example, the array of integers in our dice sample has 11 array elements. The elements are integers, so each needs two bytes of storage. The entire array, then, uses 22 bytes of storage.

The amount of memory in your computer is usually given in terms of kilobytes. A good background in metric units would tell you that a kilobyte is 1000 bytes, but unfortunately, that isn't quite correct. Internally, modern computers are built around powers of two, so computer types define the kilobyte as 1024 bytes, which is 2 raised to the 10th power. A megabyte is a kilo-kilobyte, or 1048576 bytes. We use K to indicate a kilobyte, and M to indicate a megabyte.

Depending on which model of the Apple IIGS you have, you are using, as a minimum, 1.25M or 1.125M, most of which is free and can be used by your program. ORCA/Pascal can access all of the memory you have, although there are a few tricks you have to know to access more than 64K.

is over 40 times faster that with debug code on. This brings home, again, a point raised in an earlier lesson. The debugger is a wonderful tool for finding errors in a program, but once the program is debugged, you should turn the debug code off for best performance.

There is one other thing to notice about this sample program. In the last lesson, it was pointed out that a procedure can access any variable defined in the program, so long as a variable by the same name is not found in the procedure. This program demonstrates that by having each subroutine work on the same global array, the totals array.

Standard Pascal Strings

You may have notices that a string was the first data type we ever dealt with, but you haven't seen much of them. Back in lesson 1, our very first program wrote a string constant to the screen. Since then, we have made extensive use of integers and real numbers, but we have never defined a string variable. The reason is that strings in Pascal are actually arrays of characters. Now that you are learning about arrays, you can also learn a great deal about strings.

In Pascal, a string is simply a packed array of characters where the first index is 1, and the second index is greater than one. The second index gives the largest number of characters that can appear in the string. For example, the string

```
name: packed array [1..10] of char;
```

could be used to hold a name. It can hold Wirth, Jobs or Allred, but it is limited to names with 10 or fewer characters. This string can't hold Westerfield, since that name has 11 characters.

For the most part, this declaration looks just like the array we used in the dice sample, except that it is an array of characters instead of an array of integers. The one big difference is that this array is packed. When you pack an array, you are telling the compiler that it is more important to you that the array be stored efficiently than that the array be accessed efficiently. On some computers, this is much more important than it is on the Apple IIGS, but even on the Apple IIGS, it makes a difference. For example, ORCA/Pascal will reserve two bytes of storage for a character, since the Apple IIGS can load and store two bytes a little faster than it can load and store a single byte. On the other hand, you only need one byte of storage to hold a character. This isn't very important with a single character variable, but if you have an array of 1000 characters, the difference starts to become important. In a packed array, ORCA/Pascal will only use one byte

Why Use Two Bytes?

In the text, I mentioned that it takes longer to access a one byte variable than a two byte variable on the Apple IIGS. This may sound a bit strange. In fact, if you know a little machine language, you may even think I am wrong. This sidebar explains why what I said is true. You need to know a little machine language to understand it. If you don't know any machine language, feel free to skip this section. It has nothing directly to do with programming in Pascal, and you can easily do everything in this course without having the slightest idea what we are talking about here.

To load and store values on the Apple IIGS, the compiler generally uses a LDA instruction to load and an STA instruction to store. Even when it doesn't, the argument that follows still applies. The 65816 CPU can be used in either short (eight-bit) or long (sixteen-bit) mode. It loads and stores one byte in short mode, and two bytes in long mode. If you look at the number of cycles it takes to load and store, you will find that, with each addressing mode, it takes one cycle more to load or store in long mode than in short mode. It would seem, then, that it would be more efficient to access one byte than two bytes.

The problem is that most of the things a Pascal program does are much more efficient in long mode than in short mode. Array indexing, pointer calculations, integer math, and so forth, all take fewer instructions and run faster in long mode. For that reason, compiled programs runs in long mode, switching to short mode only when necessary.

When a program needs to load a one-byte value, it actually loads two bytes and uses and AND #$00FF to mask out the extra byte. This instruction takes three bytes and three cycles, more than wiping out the one cycle saving you would get from loading a short value. In fact, it also wipes out the one byte savings of using a single byte to store the value in the first place. To store a single byte is even worse. The compiler must use a REP #$20 to switch to short mode, do the store, then use a SEP #$20 to switch back to long mode. This takes five cycles longer and four bytes of storage more than if the store was to a two-byte value.

So, as you can see, the compiler really is better off storing a character in two bytes when the character is a single variable. The resulting program is both faster and shorter than if the character were stored in a single byte!

to store each of the characters in the array. In an unpacked array, ORCA/Pascal will use two bytes for each character in the array.

There is also another important reason to use packed arrays. In Pascal, a string gets some special treatment. There are a few things you can do with a string that you cannot do with other kinds of arrays. Pascal only treats an array of characters as a string if it is a packed array. In addition, the array must have a low index of one and a high index greater than one.

One of the things that you can do with a string that you cannot do with other arrays is to use readln to read the string, and writeln to write it. (Actually, Standard Pascal does not allow you to read a string with readln, but it does let you write a string with writeln. Reading a string with readln, though, is a very common extension to Pascal.) Along with the Length function, which is a built-in function that returns the number of characters in a string, we can write a simple program to reverse the order of characters in a string, as shown in listing 5.2.

With one exception, we have covered everything you need to know to understand this program. The one thing that may seem very odd is this:

```
if index < max then
   outString[index+1] := chr(0);
```

The reason for this statement is a bit involved, but it is important that you understand it if you will ever use another Pascal compiler on some other computer. Standard Pascal really doesn't handle strings very well. In fact, in Standard Pascal, most of what we did in this program would not work. The reason is that Standard Pascal does not have a concept of a variable length string. In Standard Pascal, the number of characters in a string is fixed, and is the same as the number of characters in the array that defines the string. This means there is no length function, and it is also the reason that Standard Pascal doesn't let you use readln to read a string. Almost every implementation of Pascal has some way of getting around this limitation; ORCA/Pascal, in fact, has two ways. In ORCA/Pascal, chr(0) is a special character. It means "ignore all of the characters from here to the end of the string." (Standard Pascal allows an implementation to define the meaning of a character this way.) That's why storing chr(0) right past the last character that we will put in outString is necessary; it tells writeln not to write any characters that happen to be in the rest of the string.

Incidentally, you probably figured this out from just looking at the program, but chr is a built-in function in Pascal that takes an integer and converts it to a character. Pascal needs a function like this because Pascal is a strongly typed language. What that really means is that Pascal wants to be very sure that you know what you are doing. So, even though the character chr(0) and the integer 0 are stored in memory the same way, and even though the program that is created would be the same if Pascal didn't require the chr function, Pascal makes you put it in as a check. You are telling Pascal, in effect, that you know 0 is an integer, and that outString[index+1] is a character, but that you want to convert the integer to a character and store it in the array, anyway. Another special function, called ord, does the opposite: it converts a character (and many other types, as you will discover later) into an integer.

Problem 5.1. When you read a string with readln, it reads all of the characters from the next character typed to the end of the line. You would hope, of course, that something reasonable would happen if you read a string that is too long for the string variable you are putting the characters into. To find out what does happen, change max in the sample program to 5, and try the program with strings of length 4, 5 and 6. Be sure and use the variables window to look at the contents of the array as the program runs.

By the way, after doing this, you should understand why the Reverse program used

```
if index < max then
   outString[index+1] := chr(0);
```

to mark the end of the string, rather than

```
outString[index+1] := chr(0);
```

Can you explain the reason?

Problem 5.2. My mother is fond of pointing out that "There's more than one way to skin a cat." She doesn't particularly dislike cats, you understand. We just come from hillbilly roots, and people are a bit graphic in that part of the country. There is also more than one way to write a program. In this course, I generally pick the method that is the clearest, but the Reverse sample program could be a lot shorter. By using a for loop that loops backwards, rather than forwards, you can eliminate the outString variable entirely, like this:

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

Listing 5.2

```
{ This program reads in a string, reverses the order of the       }
{ characters, and writes the string back to the shell window.  It }
{ continues doing this until a string of length zero is entered.  }
{ To get a string of length zero, press the RETURN key without    }
{ typing any other character.                                      }

program Reverse(input, output);

const
   max = 255;                   {max length of the input string}

var
   inString,                    {input string}
   outString: packed array[1..max] of char; {output string}


   procedure Reverse;

   { Reverse a string                                    }
   {                                                     }
   { Variables:                                          }
   {    inString - string to reverse                     }
   {    outString - reversed string                      }

   var
      i: integer;               {loop variable}
      index: integer;           {index into the reversed string}

   begin {Reverse}
   index := length(inString);
   if index < max then
      outString[index+1] := chr(0);
   for i := 1 to length(inString) do begin
      outString[index] := inString[i];
      index := index-1;
      end; {for}
   end; {Reverse}


begin
repeat                          {loop until there is no input string}
   write('String  :');          {get a string}
   readln(inString);
   Reverse;                     {reverse the string}
   writeln('Reversed:', outString); {write the reversed string}
   writeln;
until length(inString) = 0;
end.
```

Change the program to use this kind of a for loop. Run the new program, and compare the size reported by the linker for the new program to the size reported for the original. Does the difference seem significant?

NOTE: The linker reports the length of the program as a hexadecimal number. You will need to convert the number to decimal to understand what the linker is telling you. To convert a number from hexadecimal to decimal, you need to know two things: in hexadecimal notation, the characters A to F represent the decimal values 10 to 15, and each column represents 16 times to one to the right, rather than 10, as with a decimal number. So, for example, if the linker reports a length of $000F3B, you figure out the number of bytes by adding together the following values:

digit	multiplier	result
B	1	11
3	16	48
F	256	3840

The number of bytes, then, is 11+48+3840, or 3899.

P-Strings, C-Strings, and Other Confusions

There is no perfect programming language, and I guess there never will be. If you ask most people who know a lot about Pascal what the biggest weakness is, I think they would agree that it is the lack of good string handling. On the other hand, very few people who learn Pascal on a microcomputer are aware of this weakness. The reason is that, while Standard Pascal is pretty weak in the area of string handling, early microcomputer based implementations of Pascal introduced some very powerful string handling abilities that have become a defacto standard in modern implementations of Pascal. Unless you end up programming on a minicomputer or mainframe, where the compilers tend to adhere closer to the standard, you will find all of these string handling extensions.

Unfortunately, the people who invented these extensions also invented a new kind of string to use the extensions. It was not necessary to do this. In fact, ORCA/Pascal implements all of the string extensions using the string format specified by Standard Pascal. It is, however, the way it was done, and you need to understand what was done so you can write programs that will work on more than one computer, and understand why some programs written for another compiler don't work right away on the Apple IIGS.

So far in this lesson, we have talked about Standard Pascal strings. ORCA/Pascal adds one extension to standard Pascal, using chr(0) to indicate the end of a string. This is called a null terminator. If the string variable can hold 10 characters, and there are ten characters in the string, however, ORCA/Pascal knows how long the string is, even though there is no null terminator.

The C programming language popularized strings with a null terminator, so these kinds of strings are often called C-strings. The Apple IIGS toolkit has many built-in subroutines that can accept a C-string as a parameter. For those subroutines, you can use a Standard Pascal string like the ones used so far in this chapter, as long as you make sure the string variable is at least one character longer than the longest string it will hold, so there is room for the null terminator. For example, if your string variable needs to hold the string

```
'Hello, world.'
```

which has 13 characters, you would need to define the variable with a high index of at least 14, like this:

```
helloString: packed array [1..14] of char;
```

When microcomputer Pascals were developed, a new kind of string was invented. This new kind of string is called a p-string, which is short for Pascal-string. This is a bit ironic, since p-strings are not a part of any of the Pascal standards, but that's the way the world works, sometimes. The main difference between Standard Pascal strings with a null terminator and p-strings is the way the length is handled. In a p-string, each string starts with an extra character that is used to hold the length of the string. This "length byte," as it is called, can hold a number from 0 to 255. The maximum length of a p-string, then, is 255 characters. (The maximum length of a Standard Pascal string is maxint, which, in ORCA/Pascal is 32767.) In ORCA/Pascal, you define a p-string by using a starting index of 0, and a maximum index that is 255 or less, like this:

```
   pstring: packed array[0..max] of
char;    {max must be in [2..255]}
```

There are functions and procedures in Pascal to read and set the length of a string, but the use of the zeroth element of the array as the length byte is universal. For that reason, with a p-string, you can get the length with the length function or as

```
ord(pstring[0])
```

Similarly, you can set the length of a p-string to val using the statement

```
pstring[0] := chr(val);
```

There is a shortcut for defining strings. While you can always define a string as a packed array of characters, you can also define a string this way:

```
str: string[len];
```

In this case, len is the length of the string. If it is 255 or smaller, ORCA/Pascal creates a p-string, and the definition is completely equivalent to

```
str: packed array[0..len] of char;
```

If, on the other hand, len is 256 or greater, ORCA/Pascal creates a Standard Pascal string, since a p-string cannot be larger than 255 characters. In that case, the definition is completely equivalent to

```
str: packed array[1..len] of char;
```

Other than the confusion it causes, the most unfortunate thing about the fact that a new definition of strings sprung up on the early microcomputer Pascals is that most Pascals don't give you a choice. By and large, mainframe and minicomputer implementations of Pascal implement Standard Pascal strings, and do not give you all of the nice string handling features found on microcomputers. Microcomputer Pascals, on the other hand, generally only implement p-strings, so that many fine programs written in Standard Pascal do not work on microcomputers. ORCA/Pascal gives you both kinds of strings. All of the built-in subroutines that operate on strings will work with either kind of string in ORCA/Pascal. You can assign one string type to another or compare strings of either type, and ORCA/Pascal will make the necessary adjustments. In short, if you use the built-in subroutines to read and set the length of strings, then unless you are calling the toolkit, you don't have to worry about what type of string you are using with ORCA/Pascal.

Problem 5.3. The Reverse sample program uses the statement

```
if index < max then
   outString[index+1] := chr(0);
```

to set the length of the output string. Another way to set the length of the output string is to start by copying the input string to the output string, so that the compiler sets the length, like this:

```
outString := inString;
```

This also makes it easier to change the type of the string from a Standard Pascal string to a p-string. Make this change to the program, then change both inString and outString to be p-strings with a maximum length of 20 characters.

The Shell Sort

There are a few basic tasks that show up over and over when you are writing real programs. One of these is sorting. If you use a program to keep track of your Christmas list, for example, you might want to sort the list by zip code so the Post office will let you send the Christmas cards out by bulk mail. If you want to check your Christmas list to see who's been naughty and nice, though, and are trying to find E. Scrooge, you may want the same list sorted alphabetically by name.

There are many ways to sort an array; each has its advantages and disadvantages. You will learn about other ways to sort an array later in the course, but we will start out now with one of the classic sorting methods. While there are faster ways to sort large arrays, the shell sort is very easy to understand, very easy to implement, and actually works better on short arrays than the more complicated sorts you will learn later.

The idea behind the shell sort is very simple. You start by scanning the array from front to back. At each step, you look to see if the value that comes after the current one in the array is smaller than the current array element. If it is, you change them and continue scanning. As an example, we will sort the following array by hand.

index	value
1	6
2	43
3	1
4	6

We start off with the first array element, and check to see if the value is smaller than the value in the second element of the array. (The arrow shows which element of the array we are working on.)

	index	value
--->	1	6
	2	43
	3	1
	4	6

In this case, 6 is smaller than 43, so we do nothing. Moving on, we check the next element.

	index	value
	1	6
--->	2	43
	3	1
	4	6

This time, 1 is smaller than 43, so we exchange the values in the second and third spots, ending up with this array:

	index	value
	1	6
--->	2	1
	3	43
	4	6

Checking the third element, we find that 6 is also smaller than 43, so we again make a swap.

	index	value
	1	6
	2	1
--->	3	6
	4	43

We don't check the last element of the array, since there is nothing that follows it.

At this point, we have successfully moved 43 to the last spot in the array, where it belongs, but the array is still not completely sorted. To sort the array completely, we need to keep track of whether or not we swapped any array entries. If we didn't need to swap any entries, then the array is sorted. If we did swap two of the array elements, though, we need to make another pass over the array. Our second pass makes one swap, moving 1 to the first array element.

index	value
1	1
2	6
3	6
4	43

Notice that we only want to swap elements of the array if the next element is actually less than the one we are inspecting. If we swap elements when the values are equal, we would loop over our sample array over and over, swapping 6 with itself on each pass.

Before diving into an example program that shows an actual sort, let's take a moment to examine one new concept that we will use that has nothing to do with arrays. While we are sorting the array, one of the things we need to keep track of is whether or not we have swapped any entries in the array. If we have, we need to make another pass through the array; if we have not swapped any entries, the sort is complete, and we can stop. One way to keep track of whether any swaps have been made would be to keep track of the number of swaps, and check to see if the number is zero. We could be a little more efficient, and set a number to zero, then set it to one if any swaps were made. Pascal has a more natural way of keeping track of whether a condition is true or false, though. It involves a new type of variable, called a boolean. A boolean variable only has two values; it is either true or false. We can set the variable to true or false using predefined constants of the same name.

Boolean variables can make a program easier to read and understand, because they provide a natural data type to record whether something is true or false. There is another advantage of boolean variables, though. Statements that evaluate a condition, like the if statement and the while and repeat loops, can use a boolean value directly to determine if a condition is met. By avoiding a comparison, like comparing to see if a number is equal to zero, the program is not only easier to read, but in many cases it is also smaller and runs faster.

We will take a more careful look at boolean variables in the next lesson. For now, though, our sample sorting program in listing 5.3 gives you an idea how useful they can be.

This is your first sort, so it is well worth your time to study how it works carefully with the debugger. Bring up the variables window while you are in the Sort subroutine, and display the first five entries or so. (Remember, you can click on the arrows to show the global variables while you are in a subroutine, and to switch back to the local variables to monitor the temp and noswap variables.) Try the program with a list of five numbers that are the same. Try a list of five numbers that are already sorted. You might also try the values from the sorting example we worked at the start of this section; the values will be handled internally as real numbers, but as you learned in Lesson 3, the built-in procedure read can read an integer and convert it to a real number.

Listing 5.3

```
{ This program reads in an array of up to 100 real numbers.  It   }
{ then sorts the array, and prints the numbers in order.  Numbers }
{ are read until a zero is found.                                 }

program SortReals(input, output);

const
   max = 100;                  {max # of reals to sort}

var
   numbers: array [1..max] of real; {array to sort}
   num: integer;               {# of numbers actually read}


   procedure ReadEm;

   { Read the list of numbers.                                    }
   {                                                              }
   { Variables:                                                   }
   {    numbers - array of numbers read                           }
   {    num - number of numbers read                              }

   var
      rval: real;              {number read from the keyboard}

   begin {ReadEm}
   num := 0;
   repeat
      read(rval);
      if rval <> 0.0 then begin
         num := num+1;
         numbers[num] := rval;
         end; {if}
   until rval = 0.0;
   end; {ReadEm}


   procedure Sort;

   { Sort the list of numbers.                                    }
   {                                                              }
   { Variables:                                                   }
   {    numbers - array of numbers read                           }
   {    num - number of numbers read                              }
```

(continued…)

(continuation of Listing 5.3)

```
      var
         temp: real;                {temp variable; used for swapping}
         noswap: boolean;           {has a swap occurred?}
         i: integer;                {loop variable}

      begin {Sort}
      repeat                        {loop until the array is sorted}
         noswap := true;            {no swaps, yet}
         for i := 1 to num-1 do     {check each element but the last}
                                    {if a swap is needed then...}
            if numbers[i+1] < numbers[i] then begin
               noswap := false;     {note that there was a swap}
               temp := numbers[i];  {swap the entries}
               numbers[i] := numbers[i+1];
               numbers[i+1] := temp;
               end; {if}
      until noswap;
      end; {Sort}


      procedure WriteEm;

      { Write the list of numbers.                                      }
      {                                                                 }
      { Variables:                                                      }
      {    numbers - array of numbers read                              }
      {    num - number of numbers read                                 }

      var
         i: integer;               {loop variable}

      begin {WriteEm}
      for i := 1 to num do
         writeln(numbers[i]);
      end; {WriteEm}


   begin
   ReadEm;                         {read the list of numbers}
   Sort;                           {sort the numbers}
   WriteEm;                        {write the list of numbers}
   end.
```

Problem 5.4. The comparison operators can work on characters as well as they can on integers or real numbers. When the comparison operators are used on two characters, stored in variables a and b, a will be less than b if the character stored in a comes before the character stored in b in the alphabet. If one character is uppercase, though, and the other character is lowercase, the uppercase character is always less than the lowercase character. Numbers are also ordered, so that '3' is less than '4', and so on.

Use this fact to modify the Reverse sample from earlier in this chapter so it will sort the characters, rather than printing them in reverse order.

Problem 5.5. The sample program from this section sorts an array so that the smallest number comes first. Sometimes we want the largest number first. Change the sample to it sorts the values with the larges first, proceeding to the smallest.

Arrays of Arrays

So far, we have looked at arrays of integers and arrays of characters, but you can actually define an array of any type in Pascal, even an array of arrays. For example, as the following program shows, we can have an array of strings, which is actually an array of packed arrays of characters.

This may seem a bit complicated at first, but the idea is actually very simple and very powerful at the same time. Pascal allows you to have an array of absolutely anything that you can define as a variable. Putting these ideas together, it is easy to create a program that can sort an array of names, rather than an array of numbers.

This does involve one new idea. In order to sort an array of strings, we need to be able to compare one string to another. This is actually a lot easier than it could be: the same comparison operators you use to compare numbers can also be used to compare strings. Strings are compared by starting with the first character of each string and comparing them. If they are equal, the next two characters are checked, and so on, until two characters are found at the same position in the string that are different, or until the end of one string is found. If all of the characters in the strings are the same, and they are the same length, they are equal. If all of the characters are the same up to the end of one string, but one string is longer than the other, the longer string is greater than the shorter string. If a character does not match, then the result of comparing the strings is the same as the result of comparing the two characters.

It takes a while to describe how strings are compared, but it is really quite natural, once you figure out what all of the rules are. Basically, as long as the strings are either all uppercase or all lowercase, the strings are sorted the same way you would alphabetize them. Listing 5.4 demonstrates this.

Problem 5.6. Try to apply the rules for comparing strings to sort the following array of strings by hand. After you give it your best shot, use the sample program to sort the strings. If you were wrong, review the rules until you understand why.

```
Run, Sally, Run.
Mike
123
Run, Dick, Run.
microphone
1212
```

Listing 5.4

```
{ This program reads in an array of up to 100 strings, each of    }
{ which can have up to 100 characters.  It then sorts the array,  }
{ and prints the numbers in order.  Strings are read until a      }
{ string of length zero is found.                                 }

program SortStrings(input, output);

const
   max = 100;                  {max # of strings to sort}
   size = 100;                 {max size of a string}

var
                               {strings to sort}
   strings: array [1..max] of packed array [1..size] of char;
   num: integer;               {# of strings actually read}


   procedure ReadEm;

   { Read the list of strings.                                    }
   {                                                              }
   { Variables:                                                   }
   {    strings - array of strings read                           }
   {    num - number of strings read                              }

   var
      sval: packed array [1..size] of char; {string read from keyboard}

   begin {ReadEm}
   num := 0;
   repeat
      readln(sval);
      if length(sval) <> 0 then begin
         num := num+1;
         strings[num] := sval;
         end; {if}
   until length(sval) = 0;
   end; {ReadEm}

   procedure Sort;

   { Sort the list of strings.                                    }
   {                                                              }
   { Variables:                                                   }
   {    numbers - array of strings read                           }
   {    num - number of strings read                              }
```

(continued…)

(continuation of Listing 5.4)

```
      var
                                    {temp variable; used for swapping}
         temp: packed array [1..size] of char;
         noswap: boolean;           {has a swap occurred?}
         i: integer;                {loop variable}

      begin {Sort}
      repeat                        {loop until the array is sorted}
         noswap := true;            {no swaps, yet}
         for i := 1 to num-1 do     {check each element but the last}
                                    {if a swap is needed then...}
            if strings[i+1] < strings[i] then begin
               noswap := false;     {note that there was a swap}
               temp := strings[i];  {swap the entries}
               strings[i] := strings[i+1];
               strings[i+1] := temp;
               end; {if}
      until noswap;
      end; {Sort}


      procedure WriteEm;

      { Write the list of strings.                                    }
      {                                                               }
      { Variables:                                                    }
      {    numbers - array of strings read                            }
      {    num - number of strings read                               }

      var
         i: integer;               {loop variable}

      begin {WriteEm}
      for i := 1 to num do
         writeln(strings[i]);
      end; {WriteEm}


   begin
   ReadEm;                         {read the list of strings}
   Sort;                           {sort the strings}
   WriteEm;                        {write the list of strings}
   end.
```

Lesson Five
Solutions to Problems

Solution to problem 5.1.

The size of the string array also determines the maximum number of characters the array can hold. If you assign a string value that is equal to the maximum length of the string variable, there is no room for the terminating null character, so the if statement checks to make sure that the string is shorter than the maximum length before setting the terminal null character.

```
{ This program reads in a string, reverses the order of the       }
{ characters, and writes the string back to the shell window.  It }
{ continues doing this until a string of length zero is entered.  }
{ To get a string of length zero, press the RETURN key without    }
{ typing any other character.                                     }

program Reverse(input, output);

const
   max = 5;                    {max length of the input string}

var
   inString,                   {input string}
   outString: packed array[1..max] of char; {output string}


   procedure Reverse;

   { Reverse a string                                      }
   {                                                       }
   { Variables:                                            }
   {    inString - string to reverse                       }
   {    outString - reversed string                        }

   var
      i: integer;              {loop variable}
      index: integer;          {index into the reversed string}

   begin {Reverse}
   index := length(inString);
   if index < max then
      outString[index+1] := chr(0);
   for i := 1 to length(inString) do begin
      outString[index] := inString[i];
      index := index-1;
      end; {for}
   end; {Reverse}
```

```
begin
repeat                          {loop until there is no input string}
   write('String  :');          {get a string}
   readln(inString);
   Reverse;                     {reverse the string}
   writeln('Reversed:', outString); {write the reversed string}
   writeln;
until length(inString) = 0;
end.
```

Solution to problem 5.2.

The length of the original program is 1219 hexadecimal, or 4633 bytes in base ten. The modified program, shown below, is 104A hexadecimal, or 4170 decimal bytes long. The difference is 463 bytes, or about 10 percent.

In one sense, this is less significant than it seems. Of the savings, 256 bytes came from eliminating the outString array. That's a little more than half of the difference.

For this particular program, the difference is minimal, and probably not worth worrying about. A close inspection of the link map, though, would show that most of the program is actually made up of libraries. The number of bytes devoted to libraries does not go up much as a program increases in size. The net result is that, in a large program, hunting for inefficiencies like this one could have a much bigger impact on program size (as well as speed). Whether the difference is significant depends on what kind of program you are writing, and how important development time is compared to the amount of memory the program will need.

```
{ This program reads in a string, reverses the order of the       }
{ characters, and writes the string back to the shell window.  It }
{ continues doing this until a string of length zero is entered.  }
{ To get a string of length zero, press the RETURN key without    }
{ typing any other character.                                     }

program Reverse(input, output);

const
   max = 255;                   {max length of the input string}

var
   inString: packed array[1..max] of char; {input string}


   procedure Reverse;

   { Reverse a string                                      }
   {                                                       }
   { Variables:                                            }
   {    inString - string to reverse                       }
   {    outString - reversed string                        }

   var
      i: integer;               {loop variable}
```

```
   begin {Reverse}
   for i := length(inString) downto 1 do
      write(inString[i]);
   writeln;
   end; {Reverse}


begin
repeat                          {loop until there is no input string}
   write('String  :');          {get a string}
   readln(inString);
   Reverse;                     {reverse and write the string}
   writeln;
until length(inString) = 0;
end.
 input string}
   write('String  :');          {get a string}
   readln(inString);
   Reverse;                     {reverse the string}
   writeln('Reversed:', outString); {write the reversed string}
   writeln;
until length(inString) = 0;
end.
```

Solution to problem 5.3.

```
{ This program reads in a string, reverses the order of the       }
{ characters, and writes the string back to the shell window.  It }
{ continues doing this until a string of length zero is entered.  }
{ To get a string of length zero, press the RETURN key without    }
{ typing any other character.                                      }

program Reverse(input, output);

const
   max = 20;                    {max length of the input string}

var
   inString,                    {input string}
   outString: packed array[0..max] of char; {output string}


   procedure Reverse;

   { Reverse a string                                      }
   {                                                       }
   { Variables:                                            }
   {    inString - string to reverse                       }
   {    outString - reversed string                        }
```

```
   var
      i: integer;               {loop variable}
      index: integer;           {index into the reversed string}

   begin {Reverse}
   index := length(inString);
   outString := inString;
   for i := 1 to length(outString) do begin
      outString[index] := inString[i];
      index := index-1;
      end; {for}
   end; {Reverse}


begin
repeat                          {loop until there is no input string}
   write('String  :');          {get a string}
   readln(inString);
   Reverse;                     {reverse the string}
   writeln('Reversed:', outString); {write the reversed string}
   writeln;
until length(inString) = 0;
end.
```

Solution to problem 5.4.

```
{ This program reads in a string, sorts the characters, and       }
{ writes the string back to the shell window.  It continues doing }
{ this until a string of length zero is entered. To get a string  }
{ of length zero, press the RETURN key without typing any other   }
{ character.                                                       }

program Sort(input, output);

const
   max = 255;                   {max length of the input string}

var
   inString,                    {input string}
   outString: packed array[1..max] of char; {output string}


   procedure Sort;

   { Sort a string                                          }
   {                                                        }
   { Variables:                                             }
   {    inString - string to sort                           }
   {    outString - reversed string                         }
```

```
   var
      i: integer;                {loop variable}
      index: integer;            {index into the reversed string}
      temp: char;                {temp variable; used for swapping}
      noswap: boolean;           {has a swap occurred?}

   begin {Sort}
   outString := inString;        {make a copy of the string}
   index := length(outString);   {get the length}
   repeat                        {loop until the array is sorted}
      noswap := true;            {no swaps, yet}
      for i := 1 to index-1 do   {check each element but the last}
                                 {if a swap is needed then...}
         if outString[i+1] < outString[i] then begin
            noswap := false;     {note that there was a swap}
            temp := outString[i]; {swap the entries}
            outString[i] := outString[i+1];
            outString[i+1] := temp;
            end; {if}
   until noswap;
   end; {Sort}


begin
repeat                          {loop until there is no input string}
   write('String  :');          {get a string}
   readln(inString);
   Sort;                        {sort the string}
   writeln('Sorted:', outString); {write the sorted string}
   writeln;
until length(inString) = 0;
end.
```

Solution to problem 5.5.

```
{ This program reads in an array of up to 100 real numbers.  It   }
{ then sorts the array, and prints the numbers in order.  Numbers }
{ are read until a zero is found.                                 }

program SortReals(input, output);

const
   max = 100;                   {max # of reals to sort}

var
   numbers: array [1..max] of real; {array to sort}
   num: integer;                {# of numbers actually read}
```

```
procedure ReadEm;

{ Read the list of numbers.                                          }
{                                                                    }
{ Variables:                                                         }
{    numbers - array of numbers read                                 }
{    num - number of numbers read                                    }

var
   rval: real;               {number read from the keyboard}

begin {ReadEm}
num := 0;
repeat
   read(rval);
   if rval <> 0.0 then begin
      num := num+1;
      numbers[num] := rval;
      end; {if}
until rval = 0.0;
end; {ReadEm}


procedure Sort;

{ Sort the list of numbers in descending order.                      }
{                                                                    }
{ Variables:                                                         }
{    numbers - array of numbers read                                 }
{    num - number of numbers read                                    }

var
   temp: real;                 {temp variable; used for swapping}
   noswap: boolean;            {has a swap occurred?}
   i: integer;                 {loop variable}

begin {Sort}
repeat                         {loop until the array is sorted}
   noswap := true;             {no swaps, yet}
   for i := 1 to num-1 do      {check each element but the last}
                               {if a swap is needed then...}
      if numbers[i+1] > numbers[i] then begin
         noswap := false;      {note that there was a swap}
         temp := numbers[i];   {swap the entries}
         numbers[i] := numbers[i+1];
         numbers[i+1] := temp;
         end; {if}
until noswap;
end; {Sort}
```

```
   procedure WriteEm;

   { Write the list of numbers.                                    }
   {                                                               }
   { Variables:                                                    }
   {    numbers - array of numbers read                            }
   {    num - number of numbers read                               }

   var
      i: integer;               {loop variable}

   begin {WriteEm}
   for i := 1 to num do
      writeln(numbers[i]);
   end; {WriteEm}


begin
ReadEm;                         {read the list of numbers}
Sort;                           {sort the numbers}
WriteEm;                        {write the list of numbers}
end.
```

Solution to problem 5.6.

The sorted strings will appear in this order:

```
1212
123
Mike
Run, Dick, Run.
Run, Sally, Run.
microphone
```

The numbers appear first because they come before characters in the ASCII character set. 1212 comes before 123 because the first character that differs is the third, where the character '1' is less than the character '3'. Note that this means that numbers are sorted in string order, not numerical order.

The other possible surprise is 'microphone', which comes not only after 'Mike', but also after 'Run, Dick, Run.' The reason is that 'microphone' stars with a lowercase character, while the other strings start with uppercase characters. Uppercase characters come before lowercase characters in the ASCII character set.

Lesson Six
Types and More About Arrays

Preventing Array Overflows

Up until the last lesson, everything you had learned about Pascal was relatively safe. No matter what you did, your program could not crash the Apple IIGS. In the last lesson, though, we changed all of that.

To understand why, lets do a though experiment. Although you will probably do something like this eventually anyway, I would still not suggest that you try this program.

```
program oops;

var
   i: integer;
   goof: array[1..2] of integer;

begin
for i := 1 to maxint do
   goof[i] := i;
end.
```

Obviously, this program has a problem. The array goof can hold two integers, but the loop will try and store 32767 integers. What would happen if you were to run this program? The answer is that I don't know, and neither does anyone else. In fact, if you try it two different times, especially if you execute the program from different environments, two different things may happen.

Basically, as it stands, the program will make a valiant effort to fill in all 32767 integers. It might even succeed. Since the program only asked for space for two integers in the array, though, the last 32765 values will be stored in places that may be free, or may be used for other purposes. The program could wipe out part of the GS/OS operating system, which has copied of part of every disk in memory. Wiping out part of GS/OS could cause loss of data on one of your disks if you save anything to disk after running a program like this one. You could wipe out a desk accessory, causing it to crash or behave strangely when used. You could wipe out part of the ORCA/Pascal environment, causing it to crash or behave strangely. You will almost certainly wipe out part of the program that is running, so it may crash before returning control to the development environment. And unfortunately, the program might just work, especially if you were only storing to a few extra values, rather than several thousand. It would work, that is, most of the time, but crash when you least expect it.

This is a very common problem in all programming languages, but the Pascal language does have a solution. Pascal provides something called range checking. You ran into range checking once before, back in lesson 2, where you found out that range checking could detect integer math errors. Range checking will also make sure that anytime you access an array, the index you give is a legal one for the array. Range checking takes extra time and memory, so it is optional, and defaults to off.

Problem 6.1. Try the sample program shown above with range checking on. To review from lesson 2, to enable range checking, place the line

```
{$rangecheck+}
```

at the start of your program.

Char Variables

In the last lesson, we made use of character variables, generally in the form of packed arrays called strings. Individual character variables are also a data type, just like integers, boolean variables, and real numbers. Like any other data type, character variables can only take on a certain range of values. These values are defined by the ASCII character set.

Characters and integers enjoy a special relationship with each other. To decide what it means to compare two strings, for example, we need to decide if one character is less than another. While you can get pretty good agreement from most people whether the character 'a' is less than the character 'b', things get a little less definite when you ask if the character '^' is less than the character '*'. For this reason, we often convert characters to integers and integers to characters.

In Pascal, we deal with this issue by looking at the ordinal value of the character. In simple terms, the ordinal value is the number associated with the character. In particular, the ordinal value of a character is the number returned when you use the ord function on the character. That's also where the name of the function comes from.

The ASCII character set defines the relationship between the characters and their ordinal values. It also lists all of the characters you can use. It has one character for each of the values from 0 to 127. Some of these values are known as printing characters. For example, the numeric value 65 is used to represent an uppercase 'A'. The lowercase a is represented by 97. Some of the values in the ASCII character set are non-printing characters. These are used for special purposes. The character whose ordinal value is 13, for example, is used to separate lines in files of characters and to move to a new line on the text screen.

Table 6.1 shows the complete ASCII character set in tabular form. Non-printing characters are shown as the name of the value. To obtain the integer value used to represent one of the characters, add the number at the top of the column to the number at the start of the row.

The ASCII character set is the dominant character set on microcomputers, but it is not universal. Even though there is no universal character set, Pascal does make a few guarantees about comparisons of characters. The alphabetic characters, for example, are always ordered. What this means is that the character a is always less than the character 'b', and so on. There is no guarantee, however, that uppercase letters are less than lowercase letters, or that the ordinal value of the character b is one greater than the ordinal value of the character 'a'.

Pascal also guarantees that the digits used to represent characters (the characters '0' to '9') are ordered and sequential. This means that you are, in fact, guaranteed that the ordinal value of the character '1' is one greater than the ordinal value of the character '0'.

On the Apple IIGS, and on most microcomputers, you can write your programs specifically for the ASCII character set. If you will be writing programs that must run on a variety of computers, though, you should be aware that the ordinal values of characters may vary. If possible, find out what character set is used on the various machines before you start to write your program, and make sure it will work with all of the character sets. Even if you can't do that, judicious use of constants can help make it easier to adapt your program to various computers.

Character Constants and String Constants

Way back in lesson 1, you learned that

```
'Hello, world.'
```

is a string constant. A string constant consists of two or more characters enclosed in quote marks. As a special case, two quote marks with no characters represents a string with no characters in it at all; this is called a null string. A character constant is very similar: a character constant is a single character enclosed in quote marks. As with strings, to get the quote mark as a character, we

	0	16	32	48	64	80	96	112
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
10	lf	sub	*	:	J	Z	j	z
11	vt	esc	+	;	K	[	k	{
12	ff	fs	,	<	L	\	l	|
13	cr	gs	-	=	M	]	m	}
14	co	rs	.	>	N	^	n	~
15	si	us	/	?	O	_	o	rub

Table 6.1

put two quote marks in the constant. All of the following are legal character constants.

```
'a'   ' '   ''''  '6'   '!'   'A'
```

It is legal to assign a character constant to a string. This works the same way as when you assign an integer to a real number: Pascal converts the character constant to a string with one character, and makes the assignment. Just as with real numbers and integers, though, where you cannot assign a real number to an integer, you also cannot assign a string to a character variable. A character variable must hold exactly one character, so it doesn't make sense to assign a string constant, with two or more characters, to the character variable. Pascal will also not let you assign a string variable to a character variable, even if the string happens to contain exactly one character when the assignment is made.

Some of these restrictions may seem a bit severe, but they are all designed to keep you from making mistakes in your programs. In all of the cases, there is a way to get around the restriction if you need to, but you have to write your program so it is easy to see that you truly know what you are doing. For example, even though you cannot assign a string to a character, even when the string consists of a single character, you can always assign one character from the string to the character variable, since the string is an array of characters. The statement

```
ch := str[1]
```

will assign the first character in the string to the character variable, for example.

Problem 6.2. The for loop can actually be used to loop over characters as well as integers. For example, you could write the alphabet like this:

```
for ch := 'a' to 'z' do
   write(ch);
```

Write a program that prints each of the ASCII characters, starting with a blank and ending with ~, to the shell window. Print both the character and its ordinal value.

Uppercase and Lowercase in Comparisons

In lesson 5, we found out how to compare and sort strings. The way comparisons are done guarantees that when we sort a list of strings made up entirely of lowercase letters, the strings are sorted alphabetically. The same is true if all of the characters are uppercase. If the strings are made up of uppercase and lowercase letters, though, the uppercase characters will appear first. 'Zoro,' as far as Pascal is concerned, is less than 'apple'.

Listing 5.1 demonstrates one way to correct this problem by converting all of the alphabetic characters in each string to uppercase or lowercase before sorting the strings. Using the information from the last section, we can develop two useful functions that will return either uppercase or lowercase characters. They also point out that characters can be passed as parameters, and returned by a function, just as integers and real numbers can be passed and returned.

You can compare characters in Pascal, but you cannot use the arithmetic operators, like + and - to operators, on characters. These functions make heavy use of the fact that you can convert easily from characters to integers and back again to do the necessary arithmetic.

Problem 6.3. Modify the string sort program from the last lesson to sort strings in true alphabetical order, then print the strings in alphabetical order. Be sure and print the original string, though, not a string that has been converted to uppercase or lowercase.

Strings in Standard Pascal

We have alluded to the fact that strings are considerably more limited in Standard Pascal than they are in most microcomputer Pascal implementations. You may use a Standard Pascal compiler some day on a minicomputer or mainframe, so it is important to know what those limitations are. The limitations that Standard Pascal places on string operations are:

1. Strings in Standard Pascal are fixed length.
2. When a string is written, all of the characters in the array are written.
3. When you assign a string constant to a string, the string constant must have exactly the same number of characters as the string variable. This also means that you cannot assign a character constant to a string variable.
4. You cannot read a string variable using the read or readln procedures; instead, you must read individual characters and place them in the array that makes up the string.
5. None of the string functions you will learn, such as the length function that returns the length of a string, are present in Standard Pascal.

Listing 6.1

```
function ToUpper(ch: char): char;

{ If the character is a lowercase alphabetic character then    }
{ return its uppercase equivalent.  Otherwise, return the      }
{ original character.                                          }
{                                                              }
{ Parameters:                                                  }
{    ch - character to convert                                 }
{                                                              }
{ Notes:                                                       }
{    This function assumes that the computer is using the      }
{    ASCII character set.                                      }

begin {ToUpper}
if ch >= 'a' then
   if ch <= 'z' then
      ch := chr(ord(ch) - ord('a') + ord('A'));
ToUpper := ch;
end; {ToUpper}


function ToLower(ch: char): char;

{ If the character is an uppercase alphabetic character then   }
{ return its lowercase equivalent.  Otherwise, return the      }
{ original character.                                          }
{                                                              }
{ Parameters:                                                  }
{    ch - character to convert                                 }
{                                                              }
{ Notes:                                                       }
{    This function assumes that the computer is using the      }
{    ASCII character set.                                      }

begin {ToLower}
if ch >= 'A' then
   if ch <= 'Z' then
      ch := chr(ord(ch) - ord('A') + ord('a'));
ToUpper := ch;
end; {ToLower}
```

Hint: Define two parallel sets of arrays, one of which holds the original strings, and the other with an uppercase only copy of the original strings. Sort the uppercase array of strings, but each time you make a swap, make the same swap in the first array.

Boolean Variables

In the last lesson, we took a very brief look at a new type, boolean. Boolean values are very simple. They can only take on two values, true or false. The

values true and false are predefined constants, generally used to set the boolean variable to a particular value.

Boolean variables are usually used to keep track of conditions within a program. In the Sort program in lesson 5, that is exactly what we used them for.

It probably isn't obvious – at least, it wasn't obvious at first to me when I learned Pascal – but booleans are truly variables, just like other variables, so you can use them in expressions and set their values with expressions. In fact, you have been using boolean expressions all along. The condition that is evaluated by an if statement, a while loop, and a repeat loop is always something that is either true or false; these are boolean values. So, for example, when you write

```
if numbers[i+1] < numbers[i] then
```

the condition,

```
numbers[i+1] < numbers[i]
```

is a *boolean expression*. "Boolean expression" is just a 50 cent way of saying that the type of the result is boolean. An integer expression is something that results in an integer value, and a real expression is something that results in a real value. We tend to talk about boolean expressions more often, though, because the if statement and it's pals all require a boolean expression to decide what to do.

Now we can put some of these ideas together to rapidly expand what we can do with the Pascal language. For example,

```
numbers[i+1] < numbers[i]
```

is a boolean expression, so we can assign it to a boolean variable. If less is a boolean variable, then the assignment

```
less := numbers[i+1] < numbers[i];
```

works perfectly well in Pascal. Also, as you saw briefly in the Sort sample in lesson 5, you can use a boolean variable as the condition in an if statement, repeat loop, or while loop.

Unless it is an election year, most people wouldn't think that it makes much sense to add true to false, or subtract true from true. Nicholas Wirth apparently agreed when he designed Pascal. On the other hand, it makes perfect sense to ask if rich and thin are true, or if tall or slim is true. It also makes sense to say that something is not false. These easy to understand concepts give rise to the three operators that are valid on boolean values, the and, or and not operators.

Like the familiar math operators +, -, * and /, and the new div and mod operators you have learned in Pascal, the and and or operators work on two values. Both of the values must be boolean values. Of course, the boolean values can be the constants true or false, a boolean variable, or the result of a boolean operator. The and operator returns true if both of the operands are true. This is the same way we use the word and in the English language when we are talking about logical consequences. For example, I will drive a Porche if I get rich *and* if my wife says it's OK. If either condition is not met, there is grave doubt about my future as a street racer, at least in a Porche. The or operator also works the way we use the word or in the English language. For example, Dan Quale will be president if he is elected in the next election *or* if George Bush dies in office. These concepts actually move the the Pascal language in an almost English form:

```
getAPorch := rich and wifeApproves;
presidentQuale := QualeElected or BushDies;
```

The not operator is the boolean form of negation. It returns the opposite of the argument. Again, it works just like the word not is used in English. I'm sure you have heard someone say, "that is not true!" Well, in Pascal, as in English, not true is the same as false. The not operator is very handy in conditions.

```
if not rich then
   writeln('Keep the Datsun.');
```

As with math operators, boolean operators use operator precedence to determine the order they are applied to the arguments. The not operator has the highest precedence, and is applied first. In fact, the not operator has the highest precedence of any operator. Next comes and, followed by or. The precedence of the operators, and their relationship with the other operators in Pascal, is shown in the following table, which you saw for the first time in lesson 2.

```
not  ~    **   @
*    /    &    <<   >>   div  mod  and
+    -    |    !    or
=    <=   >=   <    >    <>   in
```

Frankly, though, I don't recommend that you depend on precedence when you are writing boolean expressions. Most people who are familiar with programming languages or algebra are not surprised (which, as you now know, means they are unsurprised) to find out that

```
1+2*3
```

is 7, rather than 9. On the other hand, there is no general agreement on what

```
truth or beauty and justice
```

means. For that reason, I would recommend using parenthesis in your boolean expressions, even when they are not technically necessary, as a form of comment. With the parenthesis in the expression, it is easy for anyone to read the expression and see what you mean.

In addition to and, or and not, there are six other hybrid operators you are already familiar with that take numeric or string arguments and return a boolean result. The result of these operators can be used in an expression anywhere a boolean variable can be used. The operators, of course, are the six comparison operators:

```
<    >    <=   >=   <>   =
```

The write or writeln statements can be used to write a boolean value. If the boolean value is true, the string 'true' is printed; for false, the string 'false' is printed. You can use field widths to select a few characters or to right-justify the values in a field. This

Listing 6.2

```
{ This program flips a coin 500 times, presenting the results as  }
{ t for heads or f for tails.  The information is packed closely  }
{ into the shell window by displaying many values on one line.    }

program FlipCoin(output);

const
   rows = 10;                  {rows of characters}
   columns = 30;               {columns of characters}

var
   i,j: integer;               {loop variables}


   function Random(max: integer): integer;

   { Return a pseudo-random number in the range 1..max.           }
   {                                                              }
   { Parameters:                                                  }
   {    max - largest number to return                            }
   {    color - interior color of the rectangle                   }

   begin {Random}
   Random := (RandomInteger mod max) + 1;
   end; {Random}

begin
Seed(1234);                     {initialize the random number generator}
for i := 1 to rows do begin     {flip the coins}
   for j := 1 to columns do
      write(Random(2) = 1:1);
   writeln;
   end; {for}
end.
```

principal is illustrated in the program in listing 6.2, which writes a series of t and f characters to indicate if a flip of a coin came up heads or tails.

While you can write boolean values, you cannot read a boolean value from the keyboard.

Like characters, boolean values have an ordinal value, and you can use the ord function to find it. The ordinal value of false is zero, while the ordinal value of true is 1. This curious fact is sometimes used in expressions, like the following line that writes the allowance due to my daughters. (Hey, it isn't much, but they're young.)

```
writeln(1.5 * ord(cleanRoom) *
 ord(mindParents) * ord(inBedOnTime));
```

In this case, the value is 1.5 if all of the conditions are met, since, if all of the conditions are true, the ordinal value of each of the boolean values is true. On the other hand, one transgression gives an ordinal value of zero, and, as you know, zero times any value is still zero.

I don't generally recommend the practice of using the ordinal value of a boolean variable in an expression. It has come in handy on a very few occasions, but your program will generally be easier to read, and often more efficient, if you stick with boolean expressions rather than doing math with the ordinal values of boolean numbers. After all, ones and zeros, on and off, true and false, and so forth are the forte of the computer, whose memory is made up of bits. The Apple IIGS computer can do boolean operations more efficiently than any of the math operations. On the other hand, if you move on to the world of tool box programming when this course is over, you will find that it is very important to know the ordinal value of true and false.

Problem 6.4. Back in lesson 3, we used two repeat loops to choose a random number, like this:

```
repeat
   repeat
      value := RandomInteger;
   until value > 0;
until value <= 100;
```

While this was a good way to learn a little about repeat loops, it is terrible Pascal.

Use what you now know about boolean expressions to combine the two repeat loops into a single repeat loop. With this accomplished, write a program that selects and prints ten random integers in the range 1 to 100.

Problem 6.5. The ord function can be used to convert a character or boolean value to it's equivalent integer value. The chr function can convert an integer into a character value, but the Pascal language has no built-in mechanism for converting an integer into a boolean value.

Write a function called bool that will convert an integer into a boolean value. If the argument is zero, return false. If the argument is not zero, return true. Test this function with a program that calls it with the values -1, 0, 1 and 2, and writes both the integer and the boolean result.

Problem 6.6. In this problem, you will develop a program to play the game Hangman. Just in case you grew up somewhere where this game wasn't played, or you grew up long enough ago that your little grey cells have dumped the rules of the game in favor of more recent information, we'll start off by giving the rules of the game.

Hangman is basically a word guessing game. When the game starts, you are told how many letters are in the word. The computer will tell you this by displaying one dash for every letter in the word. You then guess a character. If the character is in the word, you are told where. If the letter appears more than one time in the word, you are told about all of the places where the character appears, not just the first. The computer game will show you the positions by writing the word after each guess, showing any characters you have guessed correctly, and displaying a dash for any you still have not guessed.

If the character you guess is not in the word, your played gets one step closer to a meeting with Jack Ketch. On the first wrong guess, your head is in the noose. The second wrong guess adds a body to the figure. The next four wrong guesses add two arms and two legs. After six wrong guesses, the game ends with the character hung.

Your program will be developed in stages. I suggest that you write the program and get it to work after each of the following steps, rather than trying to write the entire program all at once.

1. Start your program with a procedure that fills in the array of words that the player can guess. The array should be declared globally. A global constant called maxWords indicates

Pseudocode for Problem 6.6

```
<declare an array of boolean values called found that is maxChars long>

<get a word from the word list>;
for i := 1 to <the length of the word> do
   found[i] := false;
done := false;
wrong := 0;
repeat
   write('The word is: "');
   for i := 1 to <the length of the word> do
      if found[i] then
         write(<word[i]>)
      else
         write('-');
   writeln('"');
   ch := <character typed by the player>;
   instring := false;
   for i := 1 to <the length of the word> do
      if <word[i]> = ch then begin
         found[i] := true;
         instring := true;
         end; {if}
   if instring then
      writeln(ch, ' is in the string.')
   else begin
      writeln(ch, ' is not in the string.');
      wrong := wrong+1;
      write('Your ');
      if wrong = 1 then
         write('head')
      ...
      else
         write('right leg');
      writeln(' is now in the noose!');
      end; {else}
   if <six wrong guesses> then begin
      writeln('Sorry, Jack Ketch got you!');
      writeln('The word was ', <word>);
      done := true;
      end; {if}
   if <there are no unknown characters> then begin
      writeln('You got it!  The word is ', <word>);
      done := true;
      end; {if}
until done;
```

how many words are in the array. A global variable called maxChars tells how many characters can be in each word. Add a subroutine to print the array and run the

program. The subroutine to print the array is only used to check the results so far; once you are sure that you are filling the array correctly, remove the subroutine that prints the array.

2. Create a subroutine that asks for a random number seed, and uses it to initialize the random number generator by calling the Seed procedure. Add the Random subroutine from lesson 4 to your program. Test your work so far by calling Random with a maximum value of maxWords, and printing the corresponding word in the word list.

3. Add a new subroutine that plays hangman. The pseudo-code in the listing box shows you what the subroutine should do.

4. Add a loop in your main program that lets the player play more than one time by asking if they want to continue whenever the program returns from the subroutine written in step 3. Do this by creating a boolean function called PlayAgain that asks the player if he wants to play another game, and returns true or false, as appropriate. Use boolean expressions to handle character responses of 'y', 'Y', 'n', or 'N'.

Trigonometry Functions

In the next section, we are going to put arrays to use by looking at how objects are rotated in the graphics window. In the process, we will be making use of some trigonometry functions. Trigonometry is complicated enough by itself. Tossing it at you with no warning in the process of talking about arrays seems a bit unfair, so we will discuss the trigonometry now.

Before we go any further, though, I want to stop the panic rising in those of you who don't like math. It's only fair to point out to those who do like math, or at least, who have learned some math and deal with it from time to time, how to do the math in Pascal. If you don't know trigonometry, and don't really care, then don't worry too much about this section. Go ahead and read it to get an idea what we are talking about, but if you don't know what a sine is, it isn't that important to your understanding of the programming portions of the next two sections.

In all of the samples and problems in this section and the next, all of the math will be done for you. If you understand it, great. If not, treat it as a pure programming problem, and just implement the equations you are given.

Trigonometry is the algebra of angles. For example, you can use trigonometry to figure out how tall a building is based on how far away from the building you are, and what the angle is between the ground and the top of the building. (Assuming, of course, that the building is straight and the ground level.)

When most of us think of angles, we think in terms of degrees. Ninety degrees, for example, is a right angle, or the angle between the sides of a box. A full circle is 360 degrees. Unfortunately, Pascal uses radians to deal with angles, not degrees. The reason for this is tied up in the way the basic trigonometry functions are calculated; it's not really important to know why Pascal uses radians, only to remember that it does. In radians, a full circle is 2π radians, or about 6.28319 radians. A right angle (90 degrees) is $\pi/2$ radians, or about 1.57080 radians.

One of the first things we must do, then, is figure out how to convert from the degrees of every day life to the radians used by Pascal, and back again. To convert from degrees to radians, we can multiply by π, and divide by 180. The opposite operation converts back. The two subroutines shown in listing 6.3 package these ideas neatly; we can use them in our programs to do all of our conversions.

The basic trigonometry functions are sine, cosine, and arc tangent. All of the other trigonometry functions can be derived from these three, although there are sometimes problems with extreme values of the functions. Pascal gives you these three functions,

The ArcTan Function

Some non-standard implementations of Pascal use the name atan instead of arctan. The atan function does the same thing as the Standard Pascal arctan function. You can use this procedure to patch programs written with the older compilers.

```
procedure atan(val: real): real;

{ patch for the atan function       }
{                                   }
{ parameters:                       }
{    val - input to arctan function }

begin {atan}
atan := arctan(val);
end; {atan}
```

A similar function can be used if you are moving a program from ORCA/Pascal to a non-standard Pascal.

Listing 6.3

```
function DtoR(degrees: real): real;

{ Convert from degrees to radians.                              }
{                                                               }
{ Parameters:                                                   }
{    degrees - angle in degrees                                 }

const
   factor = 0.017453293;            {pi/180}

begin {DtoR}
DtoR := degrees * factor;
end; {DtoR}


function RtoD(radians: real): real;

{ Convert from radians to degrees.                              }
{                                                               }
{ Parameters:                                                   }
{    radians - angle in radians                                 }

const
   factor = 57.295780;              {180/pi}

begin {RtoD}
RtoD := degrees * factor;
end; {RtoD}
```

calling them sin, cos and arctan.

The sin and cos functions accept an angle, expressed in radians, as the argument. They return the sine or cosine of the argument. The following program makes use of the sin and cos functions to determine the height of a building. The program assumes that the building is straight (it would not work on some buildings in Pizza, Italy), and that the observer is an the same height as the bottom of the building. This assumption means that there is a right angle between the side of the building and the line from the person who measures the angle and the base of the building. The distance to the building is the distance from the person measuring the angle.

I have used the same idea in a hand-held calculator to find the approximate altitude reached by a model rocket, so you can see that there are some extremely important applications awaiting the program in listing 6.4!

The legal range for the trigonometry functions, as well as the range of values returned, has more to do with mathematics than with programming, so we won't go into them here. You can check the Pascal reference manual for details.

If you know much trigonometry, you may have gagged a bit when I used sine/cosine, rather than the tangent, which is the same thing. It seems inefficient the way I did it, and it is. There is, however, a good reason. Standard Pascal does not have a tangent function.

I'm really not sure why Pascal has so few mathematical functions. It probably has to do with the fact that Pascal was developed by and for computer scientists, rather than engineers. In fairness, it has all of the ones that are really necessary; it is trivial to create a tangent function if you already have a sine and cosine function, after all. Still, the function you create will not be as fast, nor, in some cases, as accurate, as a function that specifically implements the tangent function.

Listing 6.4

```
{ Find the height of a building.                                    }

program Height(input,output);

var
   distance: real;              {distance to building}
   angle: real;                 {angle between base & top}


   function DtoR(degrees: real): real;

   { Convert from degrees to radians.                               }
   {                                                                }
   { Parameters:                                                    }
   {    degrees - angle in degrees                                  }

   const
      factor = 0.017453293;            {pi/180}

   begin {DtoR}
   DtoR := degrees * factor;
   end; {DtoR}


begin
writeln('This program finds the height of');
writeln('a building.  You must supply the');
writeln('distance to the building and the');
writeln('angle between the base and top of');
writeln('the building (in degrees).');
writeln;
write('distance='); readln(distance);
write('angle='); readln(angle);
writeln('height = ', sin(DtoR(angle))/cos(DtoR(angle))*distance:1:2);
end.
```

Like many Pascals, ORCA/Pascal has several additional trigonometry functions. The additional trigonometry functions offered by ORCA/Pascal are arccos, arcsin, arctan2, and tan. You should use these functions when you can on the Apple IIGS, but be prepared to create your own versions from trigonometry identities if you need to move your program to other computers.

Rotation

Over the past three lessons, you have learned two new data types – character and boolean – and two powerful new concepts – arrays and subroutines. You also learned a powerful programming idea, the sort. In this section, we are going to put some of those new ideas to use to develop an equally powerful idea for dealing with pictures: rotation. Rotation is a very powerful concept in graphics programs. Using rotation, you can create beautiful spiral shapes, draw circles, spin a shape, or flip characters around to a new orientation.

The method we will use to rotate an object is to rotate each of the points in the object individually. In fact, we use a separate formula to calculate the new horizontal coordinate and vertical coordinate. If you know enough trigonometry to figure out why the

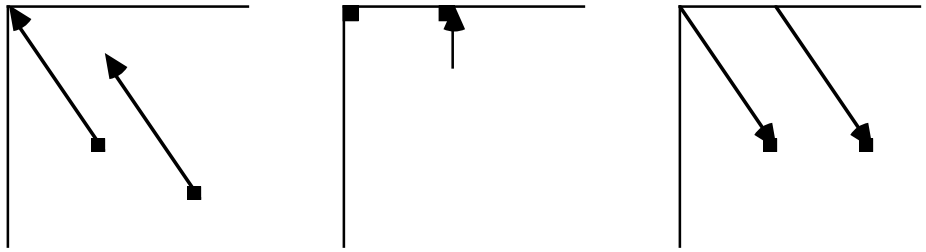


Figure 6.2: Rotating About An Arbitrary Point

following formulas work, it might be fun for you to work them out for yourself. If you are curious, but don't already know how do get the formulas on your own, there are many fine books on computer graphics that can point you in the right direction. Deriving the formulas is a bit beyond the scope of this course, though.

Of course, even if you don't know why the math works, it is important for you to be able to picture what the subroutine does. Basically, to rotate and object, we rotate each of the points in the object. To understand how a point is rotated, imagine that we attach the point to the origin with a rigid rod. (The origin is the place in the graphics window where both the horizontal and vertical coordinates are zero. In all of our programs, that is the top left corner of the graphics window.) We will let the point move, and we will let the rod spin around the origin, but the rod keeps the point exactly the same distance from the origin no matter where we move the point. Spinning the point completely around the origin would move the point in a circle.

To rotate the point, then, we need to know where the point is, and what angle we should rotate the point through.

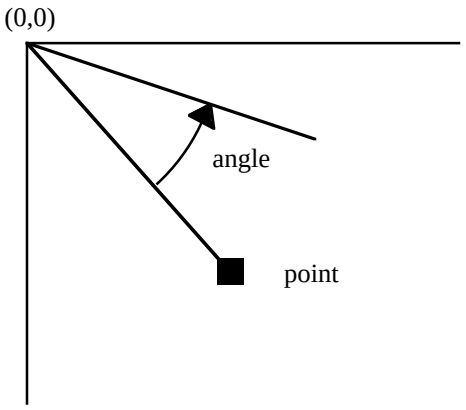


Figure 6.1: How a Point is Rotated

Once we know the coordinates of the point and the angle, we can figure out where the new point is using these formulas, shown here in Pascal:

```
newX := oldX*cos(angle) + oldY*sin(angle);
newY := oldY*cos(angle) - oldX*sin(angle);
```

Of course, this would be pretty worthless if we could only rotate a point around the top left corner of the graphics window. What we really want to do is to be able to rotate the point around some other point we provide the rotation subroutine. Just to make it easier to talk about what we are doing, we will call the point we want to rotate P, and the point we want to rotate P around O. To rotate the point, then, we can subtract O from each of the points. This effectively moves O to the origin, and P to a place that is in the same direction and at the same distance from O that it was when we started. Since O is now the origin, we can rotate the point, then move them back to their new locations by adding O to each point again. In the actual subroutine, we don't really need to move the original point around, but the idea is the same.

There is one last problem that we have to deal with to rotate an object. As you have already found out, a pixel on the Apple IIGS graphics screen is taller than it is wide. We are going to deal with this important issue by keeping track of our object using real variables in a perfect, imaginary graphics window where moving one unit horizontally looks the same as moving one unit vertically. We will use two constants, xScale and yScale, to convert one of our perfect points to the coordinate system used in the graphics window. Of course, we also need to convert the real number to an integer, something Pascal will not do for us automatically. To do that, we will use a new function, called round. The round function converts a real number to the integer that is closest to it. For example, round(1.3) is 1, and round(1.7) is 2.

These ideas are (finally!) put to use in the program in listing 6.5, which spins a square in the graphics window.

When I was taking Physics classes, one of the things instructors used to delight in doing was to start a one semester course by writing a few equations on the board. "There," they would say. "That's all you need to learn this semester."

Well, in a way they were right, of course. From a few basic principals, we learned to do some amazing things. The same thing happens in Pascal. You already know all of the basic principals involved in this program, but some of them are being put to very new uses. Let's step through them and explain what is happening.

First, this program is another good example, of using subroutines to organize and simplify a program. For example, we have packages the ideas of drawing and rotating a square into procedures. We have also reused our standard procedure, InitGraphics, to get

Listing 6.5

```
{ Rotate a cube in the graphics window.         }
{                                               }
{ This program makes use of two constants,      }
{ xScale and yScale, to decide how to convert   }
{ from the real numbers used to represent the   }
{ points of the cube into the integer           }
{ coordinates used by QuickDraw.  These values  }
{ will convert from inches to pixels in 640     }
{ mode on a 12" monitor.                        }

program RotateCube;

uses Common, QuickDrawII;

const
   xScale = 86;                 {x conversion factor}
   yScale = 33;                 {y conversion factor}

   pi = 3.1415927;              {circumference of a circle}

var
   x,y,oldX,oldY: array[1..4] of real; {points in the square}
   i: integer;                  {loop variable}


   procedure InitGraphics;

   { Standard graphics initialization           }

   begin {InitGraphics}
   SetPenMode(0);               {pen mode = copy}
   SetSolidPenPat(0);           {pen color = black}
   SetPenSize(3,1);             {use a square pen}
   end; {InitGraphics}
```

(continued)

(continuation of Listing 6.5)

```
procedure Rotate (var x,y: real; angle,ox,oy: real);

{ Rotate the point x,y about ox,oy through   }
{ the angle given.                           }
{                                            }
{ Parameters:                                }
{    x,y - point to rotate                   }
{    angle - angle to rotate (in radians)    }
{    ox,oy - point to rotate around          }

var
   cosAngle,sinAngle: real;  {sin and cos of angle}
   nx: real;                 {new x}

begin {Rotate}
x := x-ox;                   {move the point}
y := y-oy;
cosAngle := cos(angle);      {this takes time - save the results}
sinAngle := sin(angle);
nx := x*cosAngle + y*sinAngle; {rotate the point}
y := y*cosAngle - x*sinAngle;
x := nx+ox;                  {move the point back}
y := y+oy;
end; {Rotate}


procedure RotateSquare;

{ Rotate the square 9 degrees                }
{                                            }
{ Variables:                                 }
{    oldX,oldY - coordinates of square       }

var
   i: integer;               {loop variable}

begin {RotateSquare}
for i := 1 to 4 do
   Rotate(oldX[i], oldY[i], pi/20.0, 1.5, 1.5);
end; {RotateSquare}
```

(continued)

ready to draw to the graphics window.

One of the things you see in the main program that is a little new is two assignment statements on one line.

(continuation of listing 6.5)

```
      procedure DrawSquare (color: integer);

      { Draw the square                              }
      {                                              }
      { Parameters:                                  }
      {    color - color to draw                     }

      begin {DrawSquare}
      SetSolidPenPat(color);       {set the pen color}
                                   {draw the square}
      MoveTo(round(x[1]*xScale), round(y[1]*yScale));
      LineTo(round(x[2]*xScale), round(y[2]*yScale));
      LineTo(round(x[3]*xScale), round(y[3]*yScale));
      LineTo(round(x[4]*xScale), round(y[4]*yScale));
      LineTo(round(x[1]*xScale), round(y[1]*yScale));
      end; {DrawSquare}


   begin
   InitGraphics;                   {set up the graphics window}
   x[1] := 1.0;   y[1] := 1.0;     {initialize the square}
   x[2] := 2.0;   y[2] := 1.0;
   x[3] := 2.0;   y[3] := 2.0;
   x[4] := 1.0;   y[4] := 2.0;
   DrawSquare(0);                  {draw the square}

   for i := 1 to 10 do begin
      oldX := x;                   {save the current location}
      oldY := y;
      RotateSquare;                {rotate}
      DrawSquare(3);               {erase the old square}
      x :=oldX;                    {set the new location}
      y := oldY;
      DrawSquare(0);               {draw the square}
      end; {for}
   end.
```

```
   {initialize the square}
   x[1] := 1.0;   y[1] := 1.0;
   x[2] := 2.0;   y[2] := 1.0;
   x[3] := 2.0;   y[3] := 2.0;
   x[4] := 1.0;   y[4] := 2.0;
```

While tastes vary among programmers, there are many people who find that putting two statements on the same line like this can make a program easier to read. In this program, the organization shows more clearly that points are being set up, not just two separate arrays.

Take a close look at these statements:

```
   oldX := x; {save the current location}
   oldY := y;
```

Now look at the declaration for the variables involved:

```
   {points in the square}
   x,y,oldX,oldY: array[1..4] of real;
```

We have blissfully used the assignment statement to assign one variable to another throughout this course. These lines show that Pascal can assign variables of any type, so long as both variables are of the same type. Right now, the only way you know to create two arrays of the same type is to declare them on the same line, but the ability to assign arrays is a lot nicer than the alternative offered by some other languages, where you have to use a loop to assign individual elements of the array, like this:

```
{save the current location}
for j := 1 to 4 do begin
   oldX[j] := x[j];
   oldY[j] := y[j];
   end; {for}
```

In fact, assigning an array to another array is more efficient than using a loop. This is especially true with very large arrays.

The header for the Rotate procedure is the first example so far of two types of variables passed as parameters to the same procedure. As with declarations of variables in the program, you separate the different types with a semicolon.

```
procedure Rotate (var x,y: real;
   angle,ox,oy: real);
```

Finally, the call to Rotate,

```
Rotate(oldX[i], oldY[i], pi/20.0,
   1.5, 1.5);
```

shows that you can not only pass an element of an array as a parameter, but by making it a variable parameter, the procedure can even change the element of the array.

Problem 6.7. The sample program rotated a square about its center, erasing the old square as it went. This animated the square, giving a low-quality movie of a moving square.

One of the surprising things about computers is that they can be used to create computer art. With just a few small changes, the sample program can create a very pretty picture.

Change the program so that it rotates about the point 1.25, 1.25 instead of 1.5, 1.5. This will give the square an off-center rotation. Remove the code that erases the old square, and increase the loop counter from 10 to 40 so the square rotates through a complete circle. Finally, change the color from black to green to purple as the square is drawn.

Getting rid of the animation will also let you clean up the loop in the main program. In particular, you should eliminate the oldX and oldY variables.

Problem 6.8. Rotation can be used to create some very interesting shapes. One simple one is a circle. Instead of rotating a square, you can rotate an individual point. Then, by connecting the points, you can draw a circle.

Create a procedure that draws a circle using this method. The procedure should accept four parameters:

cx, cy	The location of the center of the circle, in inches. The location is measured from the top left corner of the graphics window. Use the scaling mechanism shown in the sample program to convert from inches to screen coordinates.
radius	The radius of the circle, in inches. The radius is the distance from the center of the circle to the edge.
color	The color of the circle.

The following pseudo-code outlines the steps you must take. Note that the pseudo-code assumes that moveto and lineto will work with real coordinates. Naturally, you need to scale the coordinates and convert them to integers using the methods shown in the sample.

```
x := cx;
y := cy+radius;
moveto(x,y);
for i := 1 to 40 do begin
   Rotate(x,y,pi/20.0,1.5,1.5);
   lineto(x,y);
   end; {for}
```

Use this procedure to draw a green circle. The center should be at 1.5, 1.5, and the radius should be 0.5.

Problem 6.9. You can quickly convert the procedure from the last problem to draw a star, rather than a circle. Change the procedure to draw a five pointed star, with the points of the stay on the edges of the old circle. To do this, start with a point at (2.0, 0.5) and rotate it around the point at the top of the circle. You will need to rotate the

point four times, by an angle of pi*2.0/5.0, which is one-fifth of a circle. Next, draw lines from point-to-point to form the star, using the same method you use to draw a five-pointed star by hand.

Test this procedure with a program that draws a purple star with a center at 1.5, 1.5, and a radius of 0.5.

Lesson Six
Solutions to Problems

Solution to problem 6.1.

```
{$rangecheck+}

program oops;

var
   i: integer;
   goof: array[1..2] of integer;

begin
for i := 1 to maxint do
   goof[i] := i;
end.
```

Solution to problem 6.2.

```
{ Print the printing ASCII characters and their ordinal values. }

program ASCII(output);

var
   ch: char;                              {loop variable}

begin
for ch := ' ' to '~' do                    {write the printing ASCII chars}
   writeln(ord(ch), ch:8);
end.
```

Solution to problem 6.3.

```
{ This program reads in an array of up to 100 strings, each of    }
{ which can have up to 100 characters.  It then sorts the array,  }
{ and prints the numbers in order.  Strings are read until a      }
{ string of length zero is found.                                 }

program SortStrings(input, output);

const
   max = 100;                  {max # of strings to sort}
   size = 100;                 {max size of a string}
```

```
var
                              {strings to sort}
   strings: array [1..max] of packed array [1..size] of char;
                              {uppercase version of strings array}
   upper: array [1..max] of packed array [1..size] of char;
   num: integer;              {# of strings actually read}


   function ToUpper(ch: char): char;

   { If the character is a lowercase alphabetic character then    }
   { return its uppercase equivalent.  Otherwise, return the      }
   { original character.                                          }
   {                                                              }
   { Parameters:                                                  }
   {    ch - character to convert                                 }
   {                                                              }
   { Notes:                                                       }
   {    This function assumes that the computer is using the      }
   {    ASCII character set.                                      }

   begin {ToUpper}
   if ch >= 'a' then
      if ch <= 'z' then
         ch := chr(ord(ch) - ord('a') + ord('A'));
   ToUpper := ch;
   end; {ToUpper}


   procedure ReadEm;

   { Read the list of strings.                                    }
   {                                                              }
   { Variables:                                                   }
   {    strings - array of strings read                           }
   {    upper - uppercase version of strings array                }
   {    num - number of strings read                              }

   var
      i: integer;                      {loop variable}
      sval: packed array [1..size] of char; {string read from keyboard}
```

```
begin {ReadEm}
num := 0;
repeat
   readln(sval);
   if length(sval) <> 0 then begin
      num := num+1;
      strings[num] := sval;
      for i := 1 to length(sval) do
         sval[i] := ToUpper(sval[i]);
      upper[num] := sval;
      end; {if}
until length(sval) = 0;
end; {ReadEm}


procedure Sort;

{ Sort the list of strings.                                          }
{                                                                    }
{ Variables:                                                         }
{    strings - array of strings read                                 }
{    upper - uppercase version of strings array                      }
{    num - number of strings read                                    }

var
                                {temp variable; used for swapping}
   temp: packed array [1..size] of char;
   noswap: boolean;             {has a swap occurred?}
   i: integer;                  {loop variable}

begin {Sort}
repeat                          {loop until the array is sorted}
   noswap := true;              {no swaps, yet}
   for i := 1 to num-1 do       {check each element but the last}
                                {if a swap is needed then...}
      if upper[i+1] < upper[i] then begin
         noswap := false;       {note that there was a swap}
         temp := strings[i];    {swap the entries}
         strings[i] := strings[i+1];
         strings[i+1] := temp;
         temp := upper[i];
         upper[i] := upper[i+1];
         upper[i+1] := temp;
         end; {if}
until noswap;
end; {Sort}
```

```
   procedure WriteEm;

   { Write the list of strings.                                    }
   {                                                               }
   { Variables:                                                    }
   {    numbers - array of strings read                            }
   {    num - number of strings read                               }

   var
      i: integer;               {loop variable}

   begin {WriteEm}
   for i := 1 to num do
      writeln(strings[i]);
   end; {WriteEm}


begin
ReadEm;                         {read the list of strings}
Sort;                           {sort the strings}
WriteEm;                        {write the list of strings}
end.
```

Solution to problem 6.4.

```
{ Print ten pseudo-random integers.                               }

program RandomIntegers(output);

const
   numIntegers = 10;            {# of integers to print}

var
   i: integer;                  {loop variable}
   value: integer;              {random value}

begin
Seed(1234);
for i := 1 to numIntegers do begin
   repeat
      value := RandomInteger;
   until (value >= 1) and (value <= 100);
   writeln(value);
   end; {for}
end.
```

Solution to problem 6.5.

```
{ Test the bool function, which converts an integer to a boolean. }

program TestBool(output);

var
   i: integer;                          {loop variable}


   function bool (i: integer): boolean;

   { Convert an integer to a boolean.                          }
   {                                                           }
   { Parameters:                                               }
   {    i: integer to convert                                  }
   {                                                           }
   { Returns:                                                  }
   {    False if i = 0; true otherwise.                        }

   begin {bool}
   bool := i <> 0;
   end; {bool}


begin
for i := -1 to 2 do
   writeln('bool(', i:1, ') = ', bool(i));
end.
```

Solution to problem 6.6.

The program developed in this problem is a fairly long one, so it is developed in stages. The four listings below show the program at the four stages of development listed in the problem.

Step 1

```
{ Hangman                                                  }
{                                                          }
{ This program plays the game of Hangman.  When the        }
{ game starts, you are given a word to guess.  The         }
{ program displays one dash for each letter in the         }
{ word.  You guess a letter.  If the letter is in the      }
{ word, the computer prints the word with all letters      }
{ you have guessed correctly shown in their correct        }
{ positions.  If you do not guess the word, you move       }
{ one step closer to being hung.  After six wrong          }
{ guesses, you loose.                                      }

program HangMan (input, output);
```

```
const
   maxWords = 10;              {possible words}
   maxChars = 8;               {number of characters in each word}

var
   words: array[1..maxWords] of string[maxChars]; {word array}


   procedure FillArray;

   { Fill the word array.                                  }
   {                                                       }
   { Variables:                                            }
   {    words - word array                                 }

   begin {FillArray}
   words[1] := 'computer';
   words[2] := 'whale';
   words[3] := 'megabyte';
   words[4] := 'modem';
   words[5] := 'chip';
   words[6] := 'online';
   words[7] := 'disk';
   words[8] := 'monitor';
   words[9] := 'window';
   words[10] := 'keyboard';
   end; {FillArray}


   procedure PrintArray;

   { Print the word array                                  }
   {                                                       }
   { Variables:                                            }
   {    words - word array                                 }

   var
      i: integer;              {loop variable}

   begin {PrintArray}
   for i := 1 to maxWords do
      writeln(words[i]);
   end; {PrintArray}


begin
FillArray;                     {fill the word array}
PrintArray;                    {print the word array}
end.
```

Step 2

```
{ Hangman                                                  }
{                                                          }
{ This program plays the game of Hangman.  When the        }
{ game starts, you are given a word to guess.  The         }
{ program displays one dash for each letter in the         }
{ word.  You guess a letter.  If the letter is in the      }
{ word, the computer prints the word with all letters      }
{ you have guessed correctly shown in their correct        }
{ positions.  If you do not guess the word, you move       }
{ one step closer to being hung.  After six wrong          }
{ guesses, you loose.                                      }

program HangMan (input, output);

const
   maxWords = 10;               {possible words}
   maxChars = 8;                {number of characters in each word}

var
   words: array[1..maxWords] of string[maxChars]; {word array}


   procedure FillArray;

   { Fill the word array.                                  }
   {                                                       }
   { Variables:                                            }
   {    words - word array                                 }

   begin {FillArray}
   words[1] := 'computer';
   words[2] := 'whale';
   words[3] := 'megabyte';
   words[4] := 'modem';
   words[5] := 'chip';
   words[6] := 'online';
   words[7] := 'disk';
   words[8] := 'monitor';
   words[9] := 'window';
   words[10] := 'keyboard';
   end; {FillArray}
```

```
   procedure GetSeed;

   { Initialize the random number generator             }

   var
      val: integer;             {seed value}

   begin {GetSeed}
   write('Please enter a random number seed:');
   readln(val);
   seed(val);
   end; {GetSeed}


   function RandomValue(max: integer): integer;

   { Return a pseudo-random number in the range 1..max.           }
   {                                                              }
   { Parameters:                                                  }
   {    max - largest number to return                            }
   {    color - interior color of the rectangle                   }

   begin {RandomValue}
   RandomValue := (RandomInteger mod max) + 1;
   end; {RandomValue}


begin
FillArray;                     {fill the word array}
GetSeed;                       {initialize the random number generator}

writeln(words[RandomValue(maxWords)]); {write a random word}
end.
```

Step 3

```
{ Hangman                                              }
{                                                      }
{ This program plays the game of Hangman.  When the    }
{ game starts, you are given a word to guess.  The     }
{ program displays one dash for each letter in the     }
{ word.  You guess a letter.  If the letter is in the  }
{ word, the computer prints the word with all letters  }
{ you have guessed correctly shown in their correct    }
{ positions.  If you do not guess the word, you move   }
{ one step closer to being hung.  After six wrong      }
{ guesses, you loose.                                  }

program HangMan (input, output);
```

```
const
   maxWords = 10;               {possible words}
   maxChars = 8;                {number of characters in each word}

var
   words: array[1..maxWords] of string[maxChars]; {word array}


   procedure FillArray;

   { Fill the word array.                                    }
   {                                                         }
   { Variables:                                              }
   {    words - word array                                   }

   begin {FillArray}
   words[1] := 'computer';
   words[2] := 'whale';
   words[3] := 'megabyte';
   words[4] := 'modem';
   words[5] := 'chip';
   words[6] := 'online';
   words[7] := 'disk';
   words[8] := 'monitor';
   words[9] := 'window';
   words[10] := 'keyboard';
   end; {FillArray}


   procedure GetSeed;

   { Initialize the random number generator             }

   var
      val: integer;             {seed value}

   begin {GetSeed}
   write('Please enter a random number seed:');
   readln(val);
   seed(val);
   end; {GetSeed}
```

```
function RandomValue(max: integer): integer;

{ Return a pseudo-random number in the range 1..max. }
{                                                     }
{ Parameters:                                         }
{    max - largest number to return                   }
{    color - interior color of the rectangle          }

begin {RandomValue}
RandomValue := (RandomInteger mod max) + 1;
end; {RandomValue}


procedure Play;

{ Play a game of hangman.                             }
{                                                     }
{ Variables:                                          }
{    words - word array                               }

var
   allFound: boolean;        {used to test for unknown chars}
   ch: char;                 {character from player}
   done: boolean;            {is the game over?}
                             {characters found by the player}
   found: array[1..maxChars] of boolean;
   len: integer;             {length of word; for efficiency}
   i: integer;               {loop variable}
   inString: boolean;        {is ch in the string?}
   word: string[maxChars];   {word to guess}
   wrong: integer;           {number of wrong guesses}

begin {Play}
                             {pick a word}
word := words[RandomValue(maxWords)];
len := length(word);         {record the length of the word}
for i := 1 to len do         {no letters guessed, so far}
   found[i] := false;
done := false;               {the game is not over, yet}
wrong := 0;                  {no wrong guesses, yet}
ch := ' ';                   {initialize the character}
```

```
repeat
   writeln;                  {write the word}
   write('The word is: "');
   for i := 1 to len do
      if found[i] then
         write(word[i])
      else
         write('-');
   writeln('"');
   write('Guess a character:'); {get the player's choice}
   readln(ch);
   inString := false;        {see if ch is in the string}
   for i := 1 to len do
      if word[i] = ch then begin
         found[i] := true;
         inString := true;
         end; {if}
   if inString then          {handle a correct guess}
      writeln(ch, ' is in the string.')

   else begin                {handle an incorrect guess}
      writeln(ch, ' is not in the string.');
      wrong := wrong+1;      {one more wrong answer...}
      write('Your ');        {tell the player how they are doing}
      if wrong = 1 then
         write('head')
      else if wrong = 2 then
         write('body')
      else if wrong = 3 then
         write('left arm')
      else if wrong = 4 then
         write('right arm')
      else if wrong = 5 then
         write('left leg')
      else {if wrong = 6 then}
         write('right leg');
      writeln(' is now in the noose!');
      end; {else}

   if wrong = 6 then begin   {see if the player is hung}
      writeln('Sorry, Jack Ketch got you!');
      writeln('The word was ', word);
      done := true;
      end; {if}
   allFound := true;         {check for unknown characters}
   for i := 1 to len do
      if not found[i] then
         allFound := false;
```

```
      if allFound then begin    {see if the player got the word}
         writeln('You got it!  The word is ', word);
         done := true;
         end; {if}
   until done;
   end; {Play}


begin
FillArray;                     {fill the word array}
GetSeed;                       {initialize the random number generator}
Play;                          {play a game}
end.
```

Step 4: The finished program

```
{ Hangman                                                 }
{                                                         }
{ This program plays the game of Hangman.  When the       }
{ game starts, you are given a word to guess.  The        }
{ program displays one dash for each letter in the        }
{ word.  You guess a letter.  If the letter is in the     }
{ word, the computer prints the word with all letters     }
{ you have guessed correctly shown in their correct       }
{ positions.  If you do not guess the word, you move      }
{ one step closer to being hung.  After six wrong         }
{ guesses, you loose.                                     }

program HangMan (input, output);

const
   maxWords = 10;              {possible words}
   maxChars = 8;               {number of characters in each word}

var
   words: array[1..maxWords] of string[maxChars]; {word array}


   procedure FillArray;

   { Fill the word array.                                 }
   {                                                      }
   { Variables:                                           }
   {    words - word array                                }
```

```
begin {FillArray}
words[1] := 'computer';
words[2] := 'whale';
words[3] := 'megabyte';
words[4] := 'modem';
words[5] := 'chip';
words[6] := 'online';
words[7] := 'disk';
words[8] := 'monitor';
words[9] := 'window';
words[10] := 'keyboard';
end; {FillArray}


procedure GetSeed;

{ Initialize the random number generator             }

var
   val: integer;             {seed value}

begin {GetSeed}
write('Please enter a random number seed:');
readln(val);
seed(val);
end; {GetSeed}


function RandomValue(max: integer): integer;

{ Return a pseudo-random number in the range 1..max. }
{                                                    }
{ Parameters:                                        }
{    max - largest number to return                  }
{    color - interior color of the rectangle         }

begin {RandomValue}
RandomValue := (RandomInteger mod max) + 1;
end; {RandomValue}
```

```
procedure Play;

{ Play a game of hangman.                                  }
{                                                          }
{ Variables:                                               }
{    words - word array                                    }

var
   allFound: boolean;        {used to test for unknown chars}
   ch: char;                 {character from player}
   done: boolean;            {is the game over?}
                             {characters found by the player}
   found: array[1..maxChars] of boolean;
   len: integer;             {length of word; for efficiency}
   i: integer;               {loop variable}
   inString: boolean;        {is ch in the string?}
   word: string[maxChars];   {word to guess}
   wrong: integer;           {number of wrong guesses}

begin {Play}
                             {pick a word}
word := words[RandomValue(maxWords)];
len := length(word);         {record the length of the word}
for i := 1 to len do         {no letters guessed, so far}
   found[i] := false;
done := false;               {the game is not over, yet}
wrong := 0;                  {no wrong guesses, yet}
ch := ' ';                   {initialize the character}
repeat
   writeln;                  {write the word}
   write('The word is: "');
   for i := 1 to len do
      if found[i] then
         write(word[i])
      else
         write('-');
   writeln('"');
   write('Guess a character:'); {get the player's choice}
   readln(ch);
   inString := false;        {see if ch is in the string}
   for i := 1 to len do
      if word[i] = ch then begin
         found[i] := true;
         inString := true;
         end; {if}
   if inString then          {handle a correct guess}
      writeln(ch, ' is in the string.')
```

```
   else begin                {handle an incorrect guess}
      writeln(ch, ' is not in the string.');
      wrong := wrong+1;      {one more wrong answer...}
      write('Your ');        {tell the player how they are doing}
      if wrong = 1 then
         write('head')
      else if wrong = 2 then
         write('body')
      else if wrong = 3 then
         write('left arm')
      else if wrong = 4 then
         write('right arm')
      else if wrong = 5 then
         write('left leg')
      else {if wrong = 6 then}
         write('right leg');
      writeln(' is now in the noose!');
      end; {else}

   if wrong = 6 then begin   {see if the player is hung}
      writeln('Sorry, Jack Ketch got you!');
      writeln('The word was ', word);
      done := true;
      end; {if}
   allFound := true;         {check for unknown characters}
   for i := 1 to len do
      if not found[i] then
         allFound := false;
   if allFound then begin    {see if the player got the word}
      writeln('You got it!  The word is ', word);
      done := true;
      end; {if}
until done;
end; {Play}


function PlayAgain: boolean;

{ See if the player wants to play another game.      }
{                                                    }
{ Returns:                                           }
{    True to play again, false to quit.              }

var
   ch: char;                 {player's response}
```

```
   begin {PlayAgain}
   ch := ' ';
   writeln;
   writeln;
   repeat
      write('Would you like to play again (y or n)?');
      readln(ch);
   until (ch = 'y') or (ch = 'Y') or (ch = 'n') or (ch = 'N');
   PlayAgain := (ch = 'y') or (ch = 'Y');
   end; {PlayAgain}


begin
FillArray;                      {fill the word array}
GetSeed;                        {initialize the random number generator}
repeat
   Play;                        {play a game}
until not PlayAgain;            {loop if he wants to play again}
end.
```

Solution to problem 6.7.

```
{ Draw a "flower" shape by rotating a cube.             }

program Flower;

uses Common, QuickDrawII;

const
   xScale = 86;                 {x conversion factor}
   yScale = 33;                 {y conversion factor}

   pi = 3.1415927;              {circumference of a circle}

var
   color: integer;              {color of the square}
   x,y: array[1..4] of real;    {points in the square}
   i: integer;                  {loop variable}


   procedure InitGraphics;

   { Standard graphics initialization           }

   begin {InitGraphics}
   SetPenMode(0);               {pen mode = copy}
   SetSolidPenPat(0);           {pen color = black}
   SetPenSize(3,1);             {use a square pen}
   end; {InitGraphics}
```

```
procedure Rotate (var x,y: real; angle,ox,oy: real);

{ Rotate the point x,y about ox,oy through   }
{ the angle given.                           }
{                                            }
{ Parameters:                                }
{    x,y - point to rotate                   }
{    angle - angle to rotate (in radians)    }
{    ox,oy - point to rotate around          }

var
   cosAngle,sinAngle: real;  {sin and cos of angle}
   nx: real;                 {new x}

begin {Rotate}
x := x-ox;                   {move the point}
y := y-oy;
cosAngle := cos(angle);      {this takes time - save the results}
sinAngle := sin(angle);
nx := x*cosAngle + y*sinAngle; {rotate the point}
y := y*cosAngle - x*sinAngle;
x := nx+ox;                  {move the point back}
y := y+oy;
end; {Rotate}


procedure RotateSquare;

{ Rotate the square 9 degrees                }
{                                            }
{ Variables:                                 }
{    oldX,oldY - coordinates of square       }

var
   i: integer;               {loop variable}

begin {RotateSquare}
for i := 1 to 4 do
   Rotate(x[i], y[i], pi/20.0, 1.25, 1.25);
end; {RotateSquare}
```

```
   procedure DrawSquare (color: integer);

   { Draw the square                              }
   {                                              }
   { Parameters:                                  }
   {    color - color to draw                     }

   begin {DrawSquare}
   SetSolidPenPat(color);       {set the pen color}
                                {draw the square}
   MoveTo(round(x[1]*xScale), round(y[1]*yScale));
   LineTo(round(x[2]*xScale), round(y[2]*yScale));
   LineTo(round(x[3]*xScale), round(y[3]*yScale));
   LineTo(round(x[4]*xScale), round(y[4]*yScale));
   LineTo(round(x[1]*xScale), round(y[1]*yScale));
   end; {DrawSquare}


begin
InitGraphics;                   {set up the graphics window}
x[1] := 1.0;   y[1] := 1.0;     {initialize the square}
x[2] := 2.0;   y[2] := 1.0;
x[3] := 2.0;   y[3] := 2.0;
x[4] := 1.0;   y[4] := 2.0;

color := 0;                     {start with black}
for i := 1 to 40 do begin
   RotateSquare;                {rotate}
   DrawSquare(color);           {erase the old square}
   color := color+1;            {get a new color}
   if color = 3 then
      color := 0;
   end; {for}
end.
```

Solution to problem 6.8.

```
   { Draw a circle.                               }

   program Circle;

   uses Common, QuickDrawII;

   const
      xScale = 86;              {x conversion factor}
      yScale = 33;              {y conversion factor}

      pi = 3.1415927;           {circumference of a circle}
```

```
procedure InitGraphics;

{ Standard graphics initialization                  }

begin {InitGraphics}
SetPenMode(0);                        {pen mode = copy}
SetSolidPenPat(0);                    {pen color = black}
SetPenSize(3,1);                      {use a square pen}
end; {InitGraphics}


procedure Rotate (var x,y: real; angle,ox,oy: real);

{ Rotate the point x,y about ox,oy through          }
{ the angle given.                                  }
{                                                   }
{ Parameters:                                       }
{    x,y - point to rotate                          }
{    angle - angle to rotate (in radians)           }
{    ox,oy - point to rotate around                 }

var
   cosAngle,sinAngle: real;           {sin and cos of angle}
   nx: real;                          {new x}

begin {Rotate}
x := x-ox;                            {move the point}
y := y-oy;
cosAngle := cos(angle);          {this takes time - save the results}
sinAngle := sin(angle);
nx := x*cosAngle + y*sinAngle;        {rotate the point}
y := y*cosAngle - x*sinAngle;
x := nx+ox;                           {move the point back}
y := y+oy;
end; {Rotate}


procedure DrawCircle (cx,cy,radius: real; color: integer);

{ Draw a circle                                     }
{                                                   }
{ Parameters:                                       }
{    color - color to draw                          }
{    cx,cy - center of the circle                   }
{    radius - radius of the circle                  }

var
   x,y: real;                         {point on the circle}
   i: integer;                        {loop variable}
```

```
   begin {DrawCircle}
   SetSolidPenPat(color);              {set the pen color}
   x := cx;                            {set the initial coordinates}
   y := cy+radius;
                                       {move to the initial point}
   MoveTo(round(x*xScale), round(y*yScale));
   for i := 1 to 40 do begin           {draw the circle}
      Rotate(x, y, pi/20.0, cx, cy);
      LineTo(round(x*xScale), round(y*yScale));
      end; {for}
   end; {DrawCircle}


begin
InitGraphics;                          {set up the graphics window}

DrawCircle(1.5, 1.5, 0.5, 2);          {draw the circle}
end.
```

Solution to problem 6.9.

```
{ Draw a star.                                          }

program Star;

uses Common, QuickDrawII;

const
   xScale = 86;                {x conversion factor}
   yScale = 33;                {y conversion factor}

   pi = 3.1415927;             {circumference of a circle}


   procedure InitGraphics;

   { Standard graphics initialization                   }

   begin {InitGraphics}
   SetPenMode(0);                      {pen mode = copy}
   SetSolidPenPat(0);                  {pen color = black}
   SetPenSize(3,1);                    {use a square pen}
   end; {InitGraphics}
```

```
procedure Rotate (var x,y: real; angle,ox,oy: real);

{ Rotate the point x,y about ox,oy through           }
{ the angle given.                                   }
{                                                    }
{ Parameters:                                        }
{    x,y - point to rotate                           }
{    angle - angle to rotate (in radians)            }
{    ox,oy - point to rotate around                  }

var
   cosAngle,sinAngle: real;          {sin and cos of angle}
   nx: real;                         {new x}

begin {Rotate}
x := x-ox;                           {move the point}
y := y-oy;
cosAngle := cos(angle);          {this takes time - save the results}
sinAngle := sin(angle);
nx := x*cosAngle + y*sinAngle;       {rotate the point}
y := y*cosAngle - x*sinAngle;
x := nx+ox;                          {move the point back}
y := y+oy;
end; {Rotate}


procedure DrawStar (cx,cy,radius: real; color: integer);

{ Draw a star                                        }
{                                                    }
{ Parameters:                                        }
{    color - color to draw                           }
{    cx,cy - center of the star                      }
{    radius - distance from the center to a point    }

var
   h,v: array[1..5] of integer;  {points of the star; QD coordinates}
   x,y: real;                        {point on the circle}
   i: integer;                       {loop variable}
```

```
   begin {DrawStar}
   SetSolidPenPat(color);              {set the pen color}
   x := cx;                            {set the top point}
   y := cy-radius;
   h[1] := round(x*xScale);          {convert it to screen coordinates}
   v[1] := round(y*yScale);
   for i := 2 to 5 do begin            {find the other 4 points}
      Rotate(x, y, pi*2.0/5.0, cx, cy);
      h[i] := round(x*xScale);
      v[i] := round(y*yScale);
      end; {for}
   MoveTo(h[3], v[3]);                 {draw the star}
   LineTo(h[1], v[1]);
   LineTo(h[4], v[4]);
   LineTo(h[2], v[2]);
   LineTo(h[5], v[5]);
   LineTo(h[3], v[3]);
   end; {DrawCircle}


begin
InitGraphics;                          {set up the graphics window}

DrawStar(1.5, 1.5, 0.5, 1);            {draw the star}
end.
```

Lesson Seven
Types

Types

You have already learned about several different types of variables. So far, we have used character, integer, real, and boolean variables. When you declare a variable, you tell Pascal what kind of variable it is by putting the name of its type right after the variable name. The general form for a variable declaration, then, is

```
variable-name: variable-type
```

Of course, there are several variations on this basic theme. You know that you can define more than one variable of the same type by separating the variable names with commas. Several of our programs do that; a frequent example is to declare two for loop index variables at the same time, as in this line, pilfered from an earlier sample.

```
{loop variables}
i,j: integer;
```

In the last lesson, though, you started doing something that, at first blush, just doesn't seem to fit the pattern of a variable definition. When you defined an array, you didn't give the name of the type; instead, you built up a complicated type for the variable as an array of some other type, like this:

```
{number of spots showing}
totals: array[2..12] of integer;
```

The reason that this works is that

```
array[2..12] of integer
```

is actually a new type in Pascal. What the definition of totals really does is to define a variable whose type is an array of integers, with the indices given.

The whole idea of allowing the programmer to define an entirely new type turns out to be so useful that Pascal carries all of this one step further. In Pascal, you can actually give your type a new name, and declare variables using the type name. Type declarations are placed right after constants, and right before the variable declarations. The totals array can be redefined like this:

```
type
   {number of spots}
   diceArray: array[2..12] of integer;

var
   {cumulative dice totals}
   totals: diceArray;
```

The reason that types are important, and how you should use them in your programs, is something you shouldn't expect to understand from reading a few paragraphs in a lesson. In fact, I won't even try to explain it yet. Instead, you are going to learn about types the same way you have learned about for loops, if statements, mod operators, and so forth: by seeing them in sample programs, and using them in your own programs. Gradually, over the next few lessons, I hope you will start to see just why types are so important, and why they truly make a language like Pascal, C, or Ada enormously more useful for real programming than the older, virtually typeless languages like FORTRAN and BASIC. It is the concept of types, and the ability to freely define them, along with the idea of records and pointers that makes Pascal such a good choice as a programming language.

Passing Arrays to a Subroutine

At the end of Lesson 6, we started exploring some very powerful concepts in graphics programming by learning to rotate a square in a graphics window. In our program, we used a procedure to handle the actual rotation. Here's the procedure we used:

```
procedure RotateSquare;

{ Rotate the square 9 degrees           }
{                                       }
{ Variables:                            }
{    oldX,oldY - coordinates of square  }

var
   i: integer;          {loop variable}

begin {RotateSquare}
for i := 1 to 4 do
   Rotate(oldX[i], oldY[i], pi/20.0, 1.5,
      1.5);
end; {RotateSquare}
```

Maybe you noticed something peculiar about this subroutine at the time. While it certainly showed very clearly that a procedure can access a global array, this procedure is very limited: it can only rotate the square defined by oldX and oldY. It would have made a lot more sense to pass the arrays to the procedure, so that the procedure could be used to rotate a number of different squares. The natural way to do that at the time would have been something like this:

```
procedure RotateSquare(var x,y:
   array[1..4] of real);

{ Rotate the square 9 degrees           }
{                                       }
{ Variables:                            }
{    x,y - coordinates of square        }

var
   i: integer;          {loop variable}

begin {RotateSquare}
for i := 1 to 4 do
   Rotate(x[i], y[i], pi/20.0, 1.5, 1.5);
end; {RotateSquare}
```

On the surface, this looks fine. The parameters x and y have the type

```
array[1..4] of real
```

This seems to be the same type we used to define the variables oldX and oldY in our program. Pascal, though, does no agree. In Pascal, two types are the same if they have the same type name. They are different if they have different type names, even if the underlying structure of the type is the same. While you didn't actually give the types you used to define the arrays a name, the fact that they appear in two different places causes Pascal to treat them as if they have different names. In fact, since the RotateSquare procedure shown could never be called due to type compatibility rules, the compiler flags an error when the procedure header is compiled. In Pascal, you can only use a type name in a parameter list.

You may have heard someone say that Pascal is a strongly typed language. This is one good example of what that means. Pascal is very picky about types. The language absolutely insists that types match, or it will flag an error. This may seem annoying at first; it seems annoying to most beginning Pascal programers I know. It seemed annoying to me, too. This strong typing, though, is part of what makes Pascal such a safe language. Because Pascal makes very strict checks on your program, it is harder to make mistakes. Once you learn to work within Pascal's rules, strong typing will seem natural. As your programs get larger and more complex, strong typing will also save you from many bugs and crashes that would occur in other programming languages that have weak typing, like C.

How, then, do you pass the array? The answer is that you define a new type, like we discussed at the start of this lesson, and use that type when you define oldX and oldY, and again when you are defining the parameters to the procedure. Now that you have a new type name, you can also split x and y onto different lines, making the program more natural. The type definition and global variable definitions look like this:

```
type
   {array of four reals}
   fourReals = array[1..4] of real;

var
   {points in the square}
   x,y: fourReals;
   {points in the last square drawn}
   oldX,oldY: fourReals;
```

The new procedure to rotate a square looks like this:

```
procedure RotateSquare(var x,y:
   fourReals);

{ Rotate the square 9 degrees           }
{                                       }
{ Variables:                            }
{    x,y - coordinates of square        }

var
   i: integer;          {loop variable}

begin {RotateSquare}
for i := 1 to 4 do
   Rotate(x[i], y[i], pi/20.0, 1.5, 1.5);
end; {RotateSquare}
```

At first glance, this example of types may make it seem like types get in the way, rather than help us out. The situation is a lot like a famous question used on the Peace Corps test that was designed to see if you are an optimist or a pessimist. The question showed a glass with water in it up to the half-way mark. The question was, is the glass half full or half empty? A pessimist would say half empty, because half of the water was gone, while an optimist would say half full, because half of the water was still there to be used.

Types have just forced us to learn a whole new technique of defining a named type just to pass an array to a procedure. This may seem like a pain. The Pascal programmer, though, sees that our program is a little easier to read. Because a new, named type was used, it is very clear that the RotateSquare procedure works on the same kind of data that we are using to define the square itself. In addition, in a long, complex program, we might use an array of four real numbers to represent something else, like a pair of points on a line. By using two different named types for the two different situations, Pascal would be able to check to make sure that we did not accidentally pass the wrong kind of data to a subroutine. This seems like a trivial example, but the same situation will probably arise if you start writing large programs.

Problem 7.1. Make the changes described in this section to the sample from Lesson 6.

Problem 7.2. Using the results of problem 7.1, create a program that rotates three squares simultaneously. Add a new parameter to RotateSquare so you can pass the angle of rotation, not just the array. Be sure and change the DrawSquare procedure at the same time, so it can be used to draw any of the three squares.

The first square should be black. Rotate it at an angle of pi/20.0 on each call to RotateSquare. The initial position should place the upper-left corner at 1.0, 1.0, and the lower right corner at 2.0, 2.0, just like the square in the sample.

The second square should be green. Rotate it at an angle of -pi/10.0 on each call to RotateSquare. The initial position should place the upper-left corner at 1.25, 1.25, and the lower right corner at 1.75, 1.75.

The third square should be purple. Rotate it at an angle of pi/5.0 on each call to RotateSquare. The initial position should place the upper-left corner at 1.4, 1.4, and the lower right corner at 1.6, 1.6.

Enumerations

In many older programming languages, like BASIC and FORTRAN, the only type you can define is an array. Arrays and the types that are built into the language are all you can use. This isn't a criticism of those two languages. They were designed to let scientists and engineers write programs to deal with mathematical equations, and they do that very well. Pascal, though, is designed for a broader range of programming jobs, so dealing with numbers and arrays of numbers just isn't enough. Pascal lets you define a wide range of complicated types to express your ideas in a more natural way. The first of these that we will look at is the enumeration.

An enumeration is a new type that consists of one or more named items. These names become the values that can be assigned to variables of the type. For example, you can define an enumeration of weekdays for a calendar program, like this:

```
type
   weekDay = (sunday, monday,
              teusday, wednesday,
              thursday, friday,
              saturday);
```

Using the new type, you can define variables that hold weekdays, like this:

```
var
   dayOff: weekday;
   thanksgiving: weekday;
```

Values can be assigned to the variables just like they are assigned to the variables you are familiar with.

```
thanksgiving := thursday;
```

To get a good grasp on how enumerations can be used, we will use a deck of cards as an example. There are a lot of new concepts in the program, shown in listing 7.1; these will be explained in detail in the next few sections.

Right at the top of the program, you see the types and variables that we are using to represent a deck of cards. Instead of using numbers to represent the cards, we can actually use the natural name of the value of the card and the suit of the card. There is a problem, though. As with the square in the last lesson, where each point on the square required two coordinates (x and y), each card in the deck of cards needs a value and a suit. We solve the problem the same way, by using two parallel arrays, one of which holds the suit of the card, while the other holds the face value of the card.

The next thing I want you to notice about this program is how, once again, we are using procedures to break a complicated task up into several simple tasks. If you are asked to shuffle a deck of cards and print the results, it might seem to be a difficult task. Stated that way, it is. To write the program, though, we just break this complicated task up into three simple tasks. First, we set up the arrays, filling in the array values with a

sorted deck of cards. This is initializing a deck of cards. It's the same thing you would do to play a game of cards in real life: open the box of cards. Next, we shuffle the cards. Finally, we deal them, printing the cards to the shell window. It is these three natural steps that you find in the body of the program:

```
{get a new (sorted) deck of cards}
InitializeDeck(suits, values);
{shuffle the deck}
Shuffle(suits, values);
{print the shuffled deck}
PrintDeck(suits, values);
```

From a programming standpoint, there is nothing

Listing 7.1

```
{ This program shuffles a deck of cards, then prints the       }
{ results.                                                     }
{                                                              }
{ The deck of cards is represented by a pair of arrays.  Each  }
{ array has one position for each of the 52 cards in a         }
{ standard deck of playing cards.  One array gives the value   }
{ of the card (see the value enumeration), while the other     }
{ gives the suit (see the suit enumeration).                   }

program Shuffle(output);

type
                                       {suits of cards}
   suit = (spades, diamonds, clubs, hearts);
                                       {face value of cards}
   value = (two, three, four, five, six, seven, eight, nine, ten,
            jack, queen, king, ace);

   suitDeck = array[1..52] of suit;    {these two arrays define}
   valueDeck = array[1..52] of value;  { a deck of cards       }

var
   suits: suitDeck;                    {our deck of cards}
   values: valueDeck;


   function Random(max: integer): integer;

   { Return a pseudo-random number in the range 1..max.          }
   {                                                             }
   { Parameters:                                                 }
   {    max - largest number to return                           }
   {    color - interior color of the rectangle                  }

   begin {Random}
   Random := (RandomInteger mod max) + 1;
   end; {Random}
```

(continued)

new in the InitializeDeck or Shuffle procedures. All of the statements and concepts have been dealt with before. The arithmetic in Shuffle is a bit involved, though, so we will look at the procedure in detail.

```
   begin {Shuffle}
   for i := 1 to 51 do begin
      j := Random(52 - (i-1)) + i-1;
      tvalue := values[i];
      values[i] := values[j];
      values[j] := tvalue;
      tsuit := suits[i];
      suits[i] := suits[j];
      suits[j] := tsuit;
      end; {for}
   end; {Shuffle}
```

(continuation of Listing 7.1)

```
   procedure InitializeDeck(var suits: suitDeck; var values: valueDeck);


   { Fills in the values to define a sorted deck of cards         }
   {                                                              }
   { Parameters:                                                  }
   {    suits - array of the card suits                           }
   {    values - array of the card values                         }

   var
      i: integer;                       {loop variable}

   begin {InitializeDeck}
   for i := 1 to 13 do begin            {initialize the suit array}
      suits[i] := spades;
      suits[i+13] := diamonds;
      suits[i+26] := clubs;
      suits[i+39] := hearts;
      end; {for}

   values[1] := two;                    {initialize the first suit}
   values[2] := three;
   values[3] := four;
   values[4] := five;
   values[5] := six;
   values[6] := seven;
   values[7] := eight;
   values[8] := nine;
   values[9] := ten;
   values[10] := jack;
   values[11] := queen;
   values[12] := king;
   values[13] := ace;
   for i := 14 to 52 do                 {copy the first suit to the}
      values[i] := values[i-13];        { remaining suits}
   end; {InitializeDeck}
```

(continued)

The idea is to shuffle the deck of cards, choosing each card randomly. To do this, we step through the deck of cards using the for loop. On each step, we want to pick one of the cards from the remaining ones, and place it in the current spot. Comparing this to a real deck of cards, what we are doing is picking a card at random from the deck, and placing it in a new pile. We then pick another card randomly from the deck and place it on top of the first card in the pile, and so forth.

To pick the card, we need to choose a card from the remaining ones. The first time, when i is 1, we want to pick from 52 cards. The next time, when i is 2, we want to pick from 52-1, or 52-(i-1) cards, and so forth. This explains the parameter to the Random call.

The card we pick must come from the remainder of the deck. When i is 1, Random will return a value from

(continuation of Listing 7.1)

```
      procedure Shuffle(var suits: suitDeck; var values: valueDeck);


      { Shuffles the deck of cards                                    }
      {                                                               }
      { Parameters:                                                   }
      {    suits - array of the card suits                            }
      {    values - array of the card values                          }

      var
         i: integer;                      {loop variable}
         j: integer;                      {card to swap with current card}
         tvalue: value;                   {temp value; for swap}
         tsuit: suit;                     {temp suit; for swap}

      begin {Shuffle}
      for i := 1 to 51 do begin
         j := Random(52 - (i-1)) + i-1;
         tvalue := values[i];
         values[i] := values[j];
         values[j] := tvalue;
         tsuit := suits[i];
         suits[i] := suits[j];
         suits[j] := tsuit;
         end; {for}
      end; {Shuffle}

      procedure PrintDeck(suits: suitDeck; values: valueDeck);


      { Prints the cards in order                                     }
      {                                                               }
      { Parameters:                                                   }
      {    suits - array of the card suits                            }
      {    values - array of the card values                          }

      var
         i: integer;                      {loop variable}
```

(continued)

1 to 52, which is exactly what we want. We then swap this card with the current card, and continue on. When i is 2, though, Random is returning a value from 1 to 51, but we want to pick a card from the ones that are left. The cards that are left are numbered 2 to 52, so we add i-1 to the value returned by Random.

Reading the explanation may not make much sense. To get a better grasp on the problem, we will trace through the first few loops by hand, writing down the values that can be returned by Random, and the values that can be assigned to j as a result. The table below shows the results for the first five iterations through the loop. This process, known as bench checking, is a little like playing the part of the debugger. It is a good way to see if your program is correct, and how it can be improved.

(continuation of Listing 7.1)

```
procedure PrintValue(v: value);

{ Print a value of a card                                    }
{                                                            }
{ Parameters:                                                }
{    v - value of the card                                   }

begin {PrintValue}
if v = two then
   write('two')
else if v = three then
   write('three')
else if v = four then
   write('four')
else if v = five then
   write('five')
else if v = six then
   write('six')
else if v = seven then
   write('seven')
else if v = eight then
   write('eight')
else if v = nine then
   write('nine')
else if v = ten then
   write('ten')
else if v = jack then
   write('jack')
else if v = queen then
   write('queen')
else if v = king then
   write('king')
else if v = ace then
   write('ace');
end; {PrintValue}
```

(continued)

i	Random(52 - (i-1))	j
1	1 to 52	1 to 52
2	1 to 51	2 to 52
3	1 to 50	3 to 52
4	1 to 49	4 to 52
5	1 to 48	5 to 52

Way back in lesson 4, I pointed out that a procedure and a function each bear a striking resemblance to a program. Each can have its own constant and variable section. They can also have their own type section. These types, constants and variables are only defined within the procedure or function; they cannot be used from outside of the procedure or function. I also pointed out that you could define other procedures and functions inside of a procedure or function, but our programs were so short at that point that a realistic example wasn't possible. The PrintDeck procedure in this program is the first example of how this is done. As you can see, there are no surprises. Printing a deck of cards is just a little involved, so, as with our main program, we use procedures to break the difficult task up into several smaller, simpler tasks. As with variables, the imbedded procedures cannot be used

(continuation of Listing 7.1)

```
        procedure PrintSuit(s: suit);

        { Print a suit of a card                                    }
        {                                                           }
        { Parameters:                                               }
        {    s - suit of the card                                   }

        begin {PrintSuit}
        if s = spades then
           write('spades')
        else if s = diamonds then
           write('diamonds')
        else if s = clubs then
           write('clubs')
        else if s = hearts then
           write('hearts');
        end; {PrintSuit}


     begin {PrintDeck}
     for i := 1 to 52 do begin
        PrintValue(values[i]);
        write(' of ');
        PrintSuit(suits[i]);
        writeln;
        end; {for}
     end; {PrintDeck}


  begin
  Seed(1234);                       {initialize the random number generator}
  InitializeDeck(suits, values);    {get a new (sorted) deck of cards}
  Shuffle(suits, values);           {shuffle the deck}
  PrintDeck(suits, values);         {print the shuffled deck}
  end.
```

from outside of PrintDeck.

Before moving on, I want to point out a few things about how enumerations actually work. Understanding how they work can help you to understand what you can and cannot do with enumerations.

Like character values and boolean values, an enumerated value has an ordinal value. In other words, internally, the compiler is using a specific integer for each enumeration. In an enumeration, the first enumeration name has an ordinal value of zero, and each succeeding name has an ordinal value one greater than the one before it. So, for the suit enumeration, the ordinal values are

name	ordinal value
spades	0
diamonds	1
clubs	2
hearts	3

Knowing how ordinal values are assigned to the enumerations is important for two reasons. First, you can use the ord function to get the ordinal value of the enumeration. This is often useful for programming tricks. The second, and more important reason, is that you can compare enumeration values, just like you can compare integer values, character values, and boolean values. It is perfectly legal to ask if the value of one card is less than another, like this:

```
if myCardValue < hisCardValue then
   writeln('I loose.')
```

As with character values, the result of the comparison is the same as the result of comparing the ordinal values. That also tells you why I put ace after the other cards in the value enumeration. By putting ace last, it has the highest ordinal value, and so it is greater than any other card value.

Problem 7.3. Acey Ducey is a card game played between two players. Your task is to write a program that will play the game.

The game is very simple. The computer starts by drawing two cards from the deck. You have to decide if the next card will have a value between the first two. A tie does not count. For example, if the two cards are the two of diamonds and the ten of clubs, the ten of hearts does not lie between the other two. If you think the next card will be between the first two, you place a bet. If not, you can pass by placing a bet of zero. If you are right, you get back double the bet. If you are wrong, you loose the bet.

Your program should start by giving you 50 dollars. It should ask for a random number seed, then initialize and shuffle the deck, as the sample program did.

You then draw the first two cards from the deck, using a variation on the PrintDeck procedure to print the values of the cards. Next, ask for a bet from the player. If the bet is negative, stop the game and print the amount of money the player has. Otherwise, draw another card and print it's value. Compare the new card to the first two. If its value lies between them, add the appropriate amount to the player's pile; otherwise, subtract the amount. Print the results and continue.

You should not allow the player to bet more money that he has.

If the player's pot goes to zero, stop the game.

Reshuffle the deck after 17 hands. Seventeen hands will use 51 cards from the deck, so another hand would use more cards than you have.

Records Store More than One Type

Programs are written to manipulate information of one sort or another. We have already found two places where the information we were dealing with consisted of more than one piece of information representing a single item. The most recent example was our card playing games, where a card had a suit and a value. We had to use two parallel arrays to keep track of a deck of cards. This, to say the least, is messy, confusing, and makes a program hard to read.

I wouldn't say things like that, of course, unless Pascal had an elegant way to solve the problem. What we need is a way to declare a variable that is made up of more than one thing. Unlike arrays, though, we don't want to force each thing to be the same type; we want to be able to put any type we want into the variable. In Pascal, we do this with records.

Like an array, a record is a type. The general formula for defining a record looks like this:

```
record-name = record
   variable-declarations
   end
```

You define variables with a record just like you would in the variable portion of a program or subroutine. They can be of any type, including arrays or other records. For example, we can define a record for a card like this:

```
type
   card = record
      s: suit;      {card's suit}
      v: value;     {card's value}
      end;
```

This new type can be put to use right away to describe a deck of cards as an array of cards. We don't have to use parallel arrays anymore!

```
   {deck of cards}
   deck = array[1..52] of card;
```

As with any other type, you can assign one record to another, pass a record to a procedure or function, declare arrays of records, or use records within another record. Like arrays, though, you cannot print a record or read one from the keyboard. You have to do that one element at a time.

That brings up a good point, though. How do you get at an individual variable within a record? You need a way to specify which element of the record you want to access, just as we specify which element of an array we want with an index. With records, we use the name of the record, followed by a dot, and the name of the variable within the record. If we define myCard as a card, like this:

```
var
   myCard: card;
```

we can assign values to the record, test the record, and so forth, using the methods we have already learned, and by specifying the individual element of the record using the dot operator. The examples below show how this is done.

```
myCard.s := diamonds;
myCard.v := ace;

if (myCard.v >= jack)
   and (myCard.v <= king) then
   writeln('face card');

PrintCard(myCard.s, myCard.v);
```

You can handle even more complicated situations by building expressions up from the basic rules you already know. To get at the value of a card in the deck of cards that we defined as an array of card records, you have to access an element of a record that is in an array of records. The first thing you have to name, then, is the array. We'll call it myDeck.

```
   myDeck
```

Next, you want a specific card within the array, so you name the card by giving an array index.

```
   myDeck[i]
```

Finally, you want a field within the record. We'll assume you are after the value field. You end up with this:

```
   myDeck[i].v
```

Since the resulting type is value, you can use this anywhere you could use a value variable. You can assign a value to it, assign it to another value, or pass it as a parameter, for example. So, from simple concepts you already know, you can build up very complicated expressions of a variable.

Problem 7.4. Modify the card shuffling sample to use records. Define a card and a deck of cards as we have done in this section. Change the various procedures so you can pass an individual deck of cards, rather than two separate arrays.

A First Look at Type Compatibility Rules

You haven't even seen all that you can do with types yet, but already it is getting confusing trying to determine exactly when one type is compatible with another one. While the examples have shown many of the ways types can be used, and the text has pointed out many ways that types cannot be used, there is no substitute at this point for a simple set of rules you can use to decide when you can use a variable of a certain type.

These type compatibility rules will tell you when you can assign one value to another. The same rules apply to passing a value as a value parameter to a procedure or function. To make it easier to talk about the rules, we will specify them as they apply to two arbitrary types, V1 and V2. We need to do this because there are a few cases where you can assign one variable to another, but the reverse is not true. For example, if

integerVariable is an integer variable, and realVariable is a real variable, the assignment

```
{legal assignment}
realVariable := integerVariable;
```

is perfectly legal, but the assignment

```
{illegal assignment}
integerVariable := realVariable;
```

is not legal. In our rules, we are assigning V2 to V1, or passing V2 as a parameter to a procedure or function expecting a value of type V1.

One problem with listing type compatibility rules at this point is that there is a lot more to know about types than you know now. As you learn more about types, these rules will be expanded upon. If you want a preview, you can read more about type compatibility in the ORCA/Pascal reference manual, or in any good Pascal reference book. Eventually, though, we'll get to the rest of the rules.

So, without further ado, the type compatibility rules say that V1 and V2 are assignment compatible if any of the following is true:

1. V1 and V2 are the same type. This means that the same type name was used to declare the two variables.
2. V1 is real and V2 is integer. (Real applies to both real and double in this context. Integer applies to integer, longint, and byte.)
3. V1 and V2 are both strings. In Standard Pascal, the strings must also have the same length, but ORCA/Pascal, like most microcomputer Pascals, does not enforce this restriction.
4. V1 is a string, and V2 is a character. Again, this is not allowed in Standard Pascal, but it is allowed in most microcomputer Pascals.

There is a little more to these rules than first meets the eye. Where do enumerations fit in, for example? The answer is that you can assign an enumeration value to a variable only if rule 1 is satisfied. In other words, they must be the same type. So, you can't mix integers and an enumerated value, and you can't use the math operations that work on integers on an enumeration. What about arrays? Unless the array is a string, rule 1 is again the only recourse. If the arrays are the same type, you can assign one array to the other or pass the array as a parameter. The same thing goes for records.

Type compatibility rules are stricter when you are trying to decide if a particular value can be passed as a var parameter. In that case, the types are only compatible if they are the same type.

Problem 7.5. Shown below are several examples of assignment statements and procedure and function calls. In each case, you need to decide if the assignment is legal, and if so, which of the rules applies. Here are the types being used (with procedures shown without the body):

```
type
   colors = (red, green, blue);
   hue = (redHue, yellowHue,
      blueHue);
   vector = array [1..3] of real;

var
   r: real;
   i: integer;
   c: colors;
   h: hue;
   c2: (red, green, blue);
   v: vector;
   v2: vector;
   v3: array[1..3] of real;
   s1: packed array[1..4] of char;
   s2: string[20];
   s3: packed array[2..4] of char;
   ch: char;

procedure test1(var r: real);
procedure test2(r: real);
procedure test3(c: colors);
procedure test4(a: vector);

a. r := 1.5;
b. r := 5;
c. r := red;
d. i := 45;
e. i := 1.5;
f. i := yellowHue;
g. i := ord(yellowHue);
h. c := red;
i. c := 0;
j. c := h;
k. c := c2;
l. test1(i);
m. test1(4.5);
n. test2(i);
o. test3(red);
p. test3(h);
```

```
   q. test4(v);
   r. test4(v3);
   s. v := v2;
   t. v := v3;
   u. s1 := s2;
   v. s1 := s3;
   w. s1 := 'Hello!';
   x. s1 := 'a';
   y. s1 := ch;
   z. ch := '';
```

Ordinal Types

As we delve deeper and deeper into the uses of types over the next few lessons, it will be very convenient to group several types we have learned into a group, so that we can talk about them at the same time. The group is referred to in Pascal books as the ordinal types. Basically, these are the types with specific, counting values, and no values in between. The ordinal types include integer, boolean, char, and enumerations.

These types all share some common properties, and that is why we need to group them together. All correspond to specific, integer values. The ord function can be used to convert from any of these types to the type integer. All of these types can also be compared to other values of the same type.

Ordinal types do not include variables that can take on non-counting values. For this reason, arrays, records, strings and reals are all absent from the ordinal types.

We have already glossed over one interesting property of the ordinal types. Any ordinal type can be used as a subscript to an array. For example, you can keep track of your social life with an array tied to days of the week, like this:

```
type
   weekDay = (sunday, monday,
              teusday, wednesday,
              thursday, friday,
              saturday);

var
   haveADate: array[sunday..saturday]
      of boolean;
   day: weekDay;
```

You can easily summarize the state of your social life, since any ordinal type can also be used in a for loop.

```
for day := sunday to saturday do
   haveADate[day] := false;
```

There are many other places where any ordinal value is legal in the Pascal language. Using the term ordinal type to apply to all of these types just makes it easier to talk about them. In fact, the very next section shows yet another place where any ordinal type will do.

The Case Statement

While this lesson has dealt with types, you may have noticed one place where our new fountain of types makes coding a bit more cumbersome. With all of these new types, there were a couple of placed where we had to evaluate a long series of if statements. A good example is the PrintValue subroutine, which we used to print the face value of a card.

```
procedure PrintValue(v: value);

{ Print a value of a card         }
{                                 }
{ Parameters:                     }
{    v - value of the card        }

begin {PrintValue}   if v = two then
   write('two')
else if v = three then
   write('three')
else if v = four then
   write('four')
else if v = five then
   write('five')
else if v = six then
   write('six')
else if v = seven then
   write('seven')
else if v = eight then
   write('eight')
else if v = nine then
   write('nine')
else if v = ten then
   write('ten')
else if v = jack then
   write('jack')
else if v = queen then
   write('queen')
else if v = king then
   write('king')
else if v = ace then
   write('ace');
end; {PrintValue}
```

It's pretty hard to read this subroutine. In particular, it is very hard to see at a glance if we left out a check for seven. Not only that, the compiler has to generate an enormous number of machine language instructions to do all of the if checks, making the program large and slow.

Pascal has a special statement, called the case statement, that is used in situations like this. The case statement is like a multiple branch. You give it an expression that can be any ordinal type, and then list the various values the expression can take on, followed by a colon, and the statement to execute. Using a case statement, the PrintValue procedure becomes

```
procedure PrintValue(v: value);

{ Print a value of a card          }
{                                  }
{ Parameters:                      }
{    v - value of the card         }

begin {PrintValue}
case v of
   two:     write('two');
   three:   write('three');
   four:    write('four');
   five:    write('five');
   six:     write('six');
   seven:   write('seven');
   eight:   write('eight');
   nine:    write('nine');
   ten:     write('ten');
   jack:    write('jack');
   queen:   write('queen');
   king:    write('king');
   ace:     write('ace');
   end; {case}
end; {PrintValue}
```

There is one peculiar thing about the case statement that stands out right away. There is an end to the case statement, but no begin was used. The reserved word case, in effect, serves as its own begin. You must have the end, though.

When the case statement executes, it starts by evaluating the expression that comes after case. In our example, the expression is a simple one, consisting of a single variable. The next statement executed is the one right after the constant value that corresponds to the value of the expression. After executing that statement, the program continues on with the statement after the reserved word end that marks the end of the list of case labels. In other words, the case statement works exactly like a series of if-else clauses. The case statement is just a bit easier to read, and gives you another way to organize your program.

There are many situations where you will want to use several different case labels for the same statement. To do this, separate the case labels with a comma, as the following example shows.

```
for i := 1 to 10 do
  case i of
    1,2,3,5,7: writeln(i:1, ' is prime');
    4,6,8,10:  writeln(i:1, ' is even');
    9:         writeln(i:1, ' is odd');
    end; {case}
```

Naturally, you can use compound statements to force the case label to apply to more than one statement, just like you would for any other Pascal construct.

There is one thing that you have to be very careful of with a case statement. In Pascal, it is an error to pass a value to a case statement when there is no corresponding case label. For example, if we make a small change in our for loop, the program is in error.

```
for i := 1 to 11 do
  case i of
    1,2,3,5,7: writeln(i:1, ' is prime');
    4,6,8,10:  writeln(i:1, ' is even');
    9:         writeln(i:1, ' is odd');
    end; {case}
```

The problem is that i will have a value of 11 the last time through the for loop, but there is no case label for the value 11. Technically, when this happens, it is a run-time error. The program should stop, and the error should be flagged. This is exactly what ORCA/Pascal will do. It is not uncommon, though, for Pascal compilers to be sloppy in this area; many microcomputer based implementations of Pascal will keep running in this situation. The most common thing for them to do is to skip to the end of the case statement. This makes sense in a way, but it isn't Pascal.

It does bring up a point, though. There are sometimes situations in a program where you want to handle a few cases, but not all. There are also many situations when it makes sense to have a default handler. Standard Pascal has no remedy, although careful use of sets, a variable type that will be covered later, does give you a way to use an if statement to get around the problem. A common extension to Pascal, though, is the otherwise clause. The statement after the otherwise label is executed whenever no other label is matched. It's a lot like an else at the end of a long series of if-else statements. The following example,

coded both as a series of if statements and as a case statement, shows how the otherwise label is used.

```
for i := 1 to 10 do
   case i of
      1: write('1st');
      2: write('2nd');
      3: write('3rd');
      otherwise: write(i:1, 'th');
      end; {case}

for i := 1 to 10 do
   if i = 1 then
      write('1st')
   else if i = 2 then
      write('2nd')
   else if i = 3 then
      write('3rd')
   else
      write(i:1, 'th');
```

Problem 7.6. Change the card shuffle sample to use case statements instead of repeated if statements. What is the difference in the size of the program? (See problem 5.2 if you have forgotten how to find the size of a program.)

Lesson Seven
Solutions to Problems

Solution to problem 7.1.

```
{ Rotate a cube in the graphics window.         }
{                                               }
{ This program makes use of two constants,      }
{ xScale and yScale, to decide how to convert   }
{ from the real numbers used to represent the   }
{ points of the cube into the integer           }
{ coordinates used by QuickDraw.  These values  }
{ will convert from inches to pixels in 640     }
{ mode on a 12" monitor.                        }

program RotateCube;

uses Common, QuickDrawII;

const
   xScale = 86;                 {x conversion factor}
   yScale = 33;                 {y conversion factor}

   pi = 3.1415927;              {circumference of a circle}

type
   fourReals = array[1..4] of real; {array of four reals}

var
   x,y: fourReals;              {points in the square}
   oldX,oldY: fourReals;        {points in the last square drawn}
   i: integer;                  {loop variable}


   procedure InitGraphics;

   { Standard graphics initialization           }

   begin {InitGraphics}
   SetPenMode(0);               {pen mode = copy}
   SetSolidPenPat(0);           {pen color = black}
   SetPenSize(3,1);             {use a square pen}
   end; {InitGraphics}
```

```
procedure Rotate (var x,y: real; angle,ox,oy: real);

{ Rotate the point x,y about ox,oy through   }
{ the angle given.                           }
{                                            }
{ Parameters:                                }
{    x,y - point to rotate                   }
{    angle - angle to rotate (in radians)    }
{    ox,oy - point to rotate around          }

var
   cosAngle,sinAngle: real;  {sin and cos of angle}
   nx: real;                 {new x}

begin {Rotate}
x := x-ox;                   {move the point}
y := y-oy;
cosAngle := cos(angle);      {this takes time - save the results}
sinAngle := sin(angle);
nx := x*cosAngle + y*sinAngle; {rotate the point}
y := y*cosAngle - x*sinAngle;
x := nx+ox;                  {move the point back}
y := y+oy;
end; {Rotate}


procedure RotateSquare(var x,y: fourReals);

{ Rotate the square 9 degrees                }
{                                            }
{ Parameters:                                }
{    x,y - coordinates of square             }

var
   i: integer;               {loop variable}

begin {RotateSquare}
for i := 1 to 4 do
   Rotate(x[i], y[i], pi/20.0, 1.5, 1.5);
end; {RotateSquare}
```

```
   procedure DrawSquare (color: integer);

   { Draw the square                              }
   {                                              }
   { Parameters:                                  }
   {    color - color to draw                     }

   begin {DrawSquare}
   SetSolidPenPat(color);       {set the pen color}
                                {draw the square}
   MoveTo(round(x[1]*xScale), round(y[1]*yScale));
   LineTo(round(x[2]*xScale), round(y[2]*yScale));
   LineTo(round(x[3]*xScale), round(y[3]*yScale));
   LineTo(round(x[4]*xScale), round(y[4]*yScale));
   LineTo(round(x[1]*xScale), round(y[1]*yScale));
   end; {DrawSquare}


begin
InitGraphics;                   {set up the graphics window}
x[1] := 1.0;   y[1] := 1.0;     {initialize the square}
x[2] := 2.0;   y[2] := 1.0;
x[3] := 2.0;   y[3] := 2.0;
x[4] := 1.0;   y[4] := 2.0;
DrawSquare(0);                  {draw the square}

for i := 1 to 10 do begin
   oldX := x;                   {save the current location}
   oldY := y;
   RotateSquare(oldX,oldY);     {rotate}
   DrawSquare(3);               {erase the old square}
   x :=oldX;                    {set the new location}
   y := oldY;
   DrawSquare(0);               {draw the square}
   end; {for}
end.
```

Solution to problem 7.2.

```
{ Rotate three cubes in the graphics window.    }
{                                               }
{ This program makes use of two constants,      }
{ xScale and yScale, to decide how to convert   }
{ from the real numbers used to represent the   }
{ points of the cube into the integer           }
{ coordinates used by QuickDraw.  These values  }
{ will convert from inches to pixels in 640     }
{ mode on a 12" monitor.                        }

program RotateCube;

uses Common, QuickDrawII;

const
   xScale = 86;                 {x conversion factor}
   yScale = 33;                 {y conversion factor}

   pi = 3.1415927;              {circumference of a circle}

type
   fourReals = array[1..4] of real; {array of four reals}

var
   bx,by: fourReals;            {points in the black square}
   gx,gy: fourReals;            {points in the green square}
   px,py: fourReals;            {points in the purple square}
   oldX,oldY: fourReals;        {points in the last square drawn}
   i: integer;                  {loop variable}


   procedure InitGraphics;

   { Standard graphics initialization           }

   begin {InitGraphics}
   SetPenMode(0);               {pen mode = copy}
   SetSolidPenPat(0);           {pen color = black}
   SetPenSize(3,1);             {use a square pen}
   end; {InitGraphics}
```

```
procedure Rotate (var x,y: real; angle,ox,oy: real);

{ Rotate the point x,y about ox,oy through   }
{ the angle given.                           }
{                                            }
{ Parameters:                                }
{    x,y - point to rotate                   }
{    angle - angle to rotate (in radians)    }
{    ox,oy - point to rotate around          }

var
   cosAngle,sinAngle: real;  {sin and cos of angle}
   nx: real;                 {new x}

begin {Rotate}
x := x-ox;                   {move the point}
y := y-oy;
cosAngle := cos(angle);      {this takes time - save the results}
sinAngle := sin(angle);
nx := x*cosAngle + y*sinAngle; {rotate the point}
y := y*cosAngle - x*sinAngle;
x := nx+ox;                  {move the point back}
y := y+oy;
end; {Rotate}


procedure RotateSquare(var x,y: fourReals; angle: real);

{ Rotate the square 9 degrees                }
{                                            }
{ Parameters:                                }
{    x,y - coordinates of square             }
{    angle - angle to rotate                 }

var
   i: integer;               {loop variable}

begin {RotateSquare}
for i := 1 to 4 do
   Rotate(x[i], y[i], angle, 1.5, 1.5);
end; {RotateSquare}
```

```
   procedure DrawSquare (x,y: fourReals; color: integer);

   { Draw the square                              }
   {                                              }
   { Parameters:                                  }
   {    x,y - coordinates of the square           }
   {    color - color to draw                     }

   begin {DrawSquare}
   SetSolidPenPat(color);       {set the pen color}
                                {draw the square}
   MoveTo(round(x[1]*xScale), round(y[1]*yScale));
   LineTo(round(x[2]*xScale), round(y[2]*yScale));
   LineTo(round(x[3]*xScale), round(y[3]*yScale));
   LineTo(round(x[4]*xScale), round(y[4]*yScale));
   LineTo(round(x[1]*xScale), round(y[1]*yScale));
   end; {DrawSquare}


begin
InitGraphics;                   {set up the graphics window}
bx[1] := 1.0;   by[1] := 1.0;   {initialize the black square}
bx[2] := 2.0;   by[2] := 1.0;
bx[3] := 2.0;   by[3] := 2.0;
bx[4] := 1.0;   by[4] := 2.0;
gx[1] := 1.25;  gy[1] := 1.25;  {initialize the green square}
gx[2] := 1.75;  gy[2] := 1.25;
gx[3] := 1.75;  gy[3] := 1.75;
gx[4] := 1.25;  gy[4] := 1.75;
px[1] := 1.4;   py[1] := 1.4;   {initialize the purple square}
px[2] := 1.6;   py[2] := 1.4;
px[3] := 1.6;   py[3] := 1.6;
px[4] := 1.4;   py[4] := 1.6;

DrawSquare(bx,by,0);            {draw the black square}
DrawSquare(gx,gy,2);            {draw the green square}
DrawSquare(px,py,1);            {draw the purple square}

for i := 1 to 10 do begin
                                {rotate the black square}
   oldX := bx;                  {save the current location}
   oldY := by;
   RotateSquare(bx,by,pi/20.0); {rotate}
   DrawSquare(oldX,oldY,3);     {erase the old square}
   DrawSquare(bx,by,0);         {draw the new square}
```

```
                                {rotate the green square}
   oldX := gx;                  {save the current location}
   oldY := gy;
   RotateSquare(gx,gy,-pi/10.0); {rotate}
   DrawSquare(oldX,oldY,3);     {erase the old square}
   DrawSquare(gx,gy,2);         {draw the new square}

                                {rotate the purple square}
   oldX := px;                  {save the current location}
   oldY := py;
   RotateSquare(px,py,pi/5.0);  {rotate}
   DrawSquare(oldX,oldY,3);     {erase the old square}
   DrawSquare(px,py,1);         {draw the new square}
   end; {for}
end.
```

Solution to problem 7.3.

```
{ This program plays Acey Ducey                                    }
{                                                                  }
{ Acey Ducey is a card game played, in this case, between the      }
{ computer and the human.  The computer draws and displays two     }
{ cards.  The player then decides how much to bet, and a third     }
{ card is drawn.  If it is between the first two, the player       }
{ wins, and gets back double the bet.  If it is not between        }
{ the two cards, the computer wins, and the player looses the      }
{ bet.  The game continues until the player looses all of his      }
{ money, or until the player signals the end of the game with      }
{ a negative bet.                                                  }
{                                                                  }
{ The deck of cards is represented by a pair of arrays.  Each      }
{ array has one position for each of the 52 cards in a             }
{ standard deck of playing cards.  One array gives the value       }
{ of the card (see the value enumeration), while the other         }
{ gives the suit (see the suit enumeration).                       }

program Shuffle(input,output);

type
                                        {suits of cards}
   suit = (spades, diamonds, clubs, hearts);
                                        {face value of cards}
   value = (two, three, four, five, six, seven, eight, nine, ten,
            jack, queen, king, ace);

   suitDeck = array[1..52] of suit;     {these two arrays define}
   valueDeck = array[1..52] of value;   { a deck of cards       }
```

```
var
   done: boolean;                        {is the game over?}
   hands: integer;                       {# of hands played from the deck}
   money: real;                          {amount of money left}
   nextCard: integer;                    {next card in the deck}
   suits: suitDeck;                      {our deck of cards}
   values: valueDeck;


   function Random(max: integer): integer;

   { Return a pseudo-random number in the range 1..max.          }
   {                                                             }
   { Parameters:                                                 }
   {    max - largest number to return                           }
   {    color - interior color of the rectangle                  }

   begin {Random}
   Random := (RandomInteger mod max) + 1;
   end; {Random}


   procedure InitializeDeck(var suits: suitDeck; var values: valueDeck);


   { Fills in the values to define a sorted deck of cards        }
   {                                                             }
   { Parameters:                                                 }
   {    suits - array of the card suits                          }
   {    values - array of the card values                        }

   var
      i: integer;                        {loop variable}

   begin {InitializeDeck}
   for i := 1 to 13 do begin             {initialize the suit array}
      suits[i] := spades;
      suits[i+13] := diamonds;
      suits[i+26] := clubs;
      suits[i+39] := hearts;
      end; {for}
```

```
values[1] := two;                    {initialize the first suit}
values[2] := three;
values[3] := four;
values[4] := five;
values[5] := six;
values[6] := seven;
values[7] := eight;
values[8] := nine;
values[9] := ten;
values[10] := jack;
values[11] := queen;
values[12] := king;
values[13] := ace;
for i := 14 to 52 do                 {copy the first suit to the}
   values[i] := values[i-13];        { remaining suits}
end; {InitializeDeck}


procedure Shuffle(var suits: suitDeck; var values: valueDeck);


{ Shuffles the deck of cards                                      }
{                                                                 }
{ Parameters:                                                     }
{    suits - array of the card suits                              }
{    values - array of the card values                            }

var
   i: integer;                       {loop variable}
   j: integer;                       {card to swap with current card}
   tvalue: value;                    {temp value; for swap}
   tsuit: suit;                      {temp suit; for swap}

begin {Shuffle}
for i := 1 to 51 do begin
   j := Random(52 - (i-1)) + i-1;
   tvalue := values[i];
   values[i] := values[j];
   values[j] := tvalue;
   tsuit := suits[i];
   suits[i] := suits[j];
   suits[j] := tsuit;
   end; {for}
end; {Shuffle}
```

```
procedure PrintCard(s: suit; v: value);

{ Prints the cards in order                                          }
{                                                                    }
{ Parameters:                                                        }
{    s - suit of the card                                            }
{    v - value of the card                                           }

var
   i: integer;                        {loop variable}


   procedure PrintValue(v: value);

   { Print a value of a card                                         }
   {                                                                 }
   { Parameters:                                                     }
   {    v - value of the card                                        }

   begin {PrintValue}
   if v = two then
      write('two')
   else if v = three then
      write('three')
   else if v = four then
      write('four')
   else if v = five then
      write('five')
   else if v = six then
      write('six')
   else if v = seven then
      write('seven')
   else if v = eight then
      write('eight')
   else if v = nine then
      write('nine')
   else if v = ten then
      write('ten')
   else if v = jack then
      write('jack')
   else if v = queen then
      write('queen')
   else if v = king then
      write('king')
   else if v = ace then
      write('ace');
   end; {PrintValue}
```

```
   procedure PrintSuit(s: suit);

   { Print a suit of a card                                    }
   {                                                          }
   { Parameters:                                              }
   {    s - suit of the card                                  }

   begin {PrintSuit}
   if s = spades then
      write('spades')
   else if s = diamonds then
      write('diamonds')
   else if s = clubs then
      write('clubs')
   else if s = hearts then
      write('hearts');
   end; {PrintSuit}


begin {PrintCard}
PrintValue(v);
write(' of ');
PrintSuit(s);
writeln;
end; {PrintCard}


procedure PlayHand;

{ Play one hand of Acey Ducey.                                }
{                                                             }
{ Variables:                                                  }
{    done - game over flag                                    }
{    money - amount of money the player has                   }
{    nextCard - next card to draw from the deck               }
{    suits,values - deck of cards                             }

var
   bet: real;                        {player's bet}
   v1,v2,v3: value;                  {value of the three cards}

begin {PlayHand}
writeln;
writeln('I draw:');
v1 := values[nextCard];              {draw the first card}
PrintCard(suits[nextCard], v1);
nextCard := nextCard+1;
v2 := values[nextCard];              {draw the second card}
PrintCard(suits[nextCard], v2);
nextCard := nextCard+1;
```

```
if v2 < v1 then begin                {sort the values}
   v3 := v2;
   v2 := v1;
   v1 := v3;
   end; {if}
repeat                               {get the bet}
   writeln('You have ', money:1:2, ' left.');
   write('Your bet:');
   readln(bet);
   if bet < 0.0 then
      done := true
   else if bet > money then
      writeln('Sorry, you don''t have that much.');
until bet <= money;
if not done then begin
   v3 := values[nextCard];           {draw the third card}
   write('Your card is:');
   PrintCard(suits[nextCard], v3);
   nextCard := nextCard+1;
   if (v1 < v3) and (v3 < v2) then begin
      money := money+bet;            {player wins}
      writeln('You win!');
      end {if}
   else begin
      money := money-bet;            {player looses}
      writeln('Sorry, you loose.');
      if money <= 0.0 then begin     {see if he's broke}
         writeln('You are out of money.  So long!');
         done := true;
         end; {if}
      end; {else}
   end; {if}
end; {PlayHand}


procedure GetSeed;

{ Initialize the random number generator                        }

var
   i: integer;                       {integer from keyboard}

begin {GetSeed}
writeln('Please enter a number from');
write  ('1000 to 30000:');
readln(i);
seed(i);
end; {GetSeed}
```

```
begin
money := 50.0;                    {player starts with $50}
GetSeed;                          {initialize the random number generator}

InitializeDeck(suits, values);    {get a new (sorted) deck of cards}
hands := 17;                      {this forces an immediate shuffle}
done := false;                    {not done, yet}
repeat
   if hands = 17 then begin       {reshuffle after 17 hands}
      Shuffle(suits, values);     {shuffle the deck}
      hands := 0;                 {no hands played from the deck}
      nextCard := 1;              {next card to draw}
      end; {if}
   PlayHand;                      {play one hand of Acey Ducey}
   hands := hands+1;              {update the # of hands played}
until done;
end.
```

Solution to problem 7.4.

```
{ This program shuffles a deck of cards, then prints the       }
{ results.                                                     }

program Shuffle(output);

type
                                       {suits of cards}
   suit = (spades, diamonds, clubs, hearts);
                                       {face value of cards}
   value = (two, three, four, five, six, seven, eight, nine, ten,
            jack, queen, king, ace);

   card = record                       {one card}
      s: suit;                         {card's suit}
      v: value;                        {card's value}
      end;
   deck = array[1..52] of card;        {deck of cards}

var
   cards: deck;                        {our deck of cards}
```

```
function Random(max: integer): integer;

{ Return a pseudo-random number in the range 1..max.           }
{                                                              }
{ Parameters:                                                  }
{    max - largest number to return                            }
{    color - interior color of the rectangle                   }

begin {Random}
Random := (RandomInteger mod max) + 1;
end; {Random}


procedure InitializeDeck(var c: deck);


{ Fills in the values to define a sorted deck of cards         }
{                                                              }
{ Parameters:                                                  }
{    c - deck of cards                                         }

var
   i: integer;                       {loop variable}

begin {InitializeDeck}
for i := 1 to 13 do begin            {initialize the suit array}
   c[i].s := spades;
   c[i+13].s := diamonds;
   c[i+26].s := clubs;
   c[i+39].s := hearts;
   end; {for}

c[1].v := two;                       {initialize the first suit}
c[2].v := three;
c[3].v := four;
c[4].v := five;
c[5].v := six;
c[6].v := seven;
c[7].v := eight;
c[8].v := nine;
c[9].v := ten;
c[10].v := jack;
c[11].v := queen;
c[12].v := king;
c[13].v := ace;
for i := 14 to 52 do                 {copy the first suit to the}
   c[i].v := c[i-13].v;              { remaining suits}
end; {InitializeDeck}
```

```
procedure Shuffle(var c: deck);


{ Shuffles the deck of cards                                    }
{                                                               }
{ Parameters:                                                   }
{    c - deck of cards                                          }

var
   i: integer;                      {loop variable}
   j: integer;                      {card to swap with current card}
   tvalue: value;                   {temp value; for swap}
   tsuit: suit;                     {temp suit; for swap}

begin {Shuffle}
for i := 1 to 51 do begin
   j := Random(52 - (i-1)) + i-1;
   tvalue := c[i].v;
   c[i].v := c[j].v;
   c[j].v := tvalue;
   tsuit := c[i].s;
   c[i].s := c[j].s;
   c[j].s := tsuit;
   end; {for}
end; {Shuffle}


procedure PrintDeck(c: deck);


{ Prints the cards in order                                     }
{                                                               }
{ Parameters:                                                   }
{    c - deck of cards                                          }

var
   i: integer;                      {loop variable}
```

```
procedure PrintValue(v: value);

{ Print a value of a card                                    }
{                                                            }
{ Parameters:                                                }
{    v - value of the card                                   }

begin {PrintValue}
if v = two then
   write('two')
else if v = three then
   write('three')
else if v = four then
   write('four')
else if v = five then
   write('five')
else if v = six then
   write('six')
else if v = seven then
   write('seven')
else if v = eight then
   write('eight')
else if v = nine then
   write('nine')
else if v = ten then
   write('ten')
else if v = jack then
   write('jack')
else if v = queen then
   write('queen')
else if v = king then
   write('king')
else if v = ace then
   write('ace');
end; {PrintValue}
```

```
      procedure PrintSuit(s: suit);

      { Print a suit of a card                                    }
      {                                                          }
      { Parameters:                                              }
      {    s - suit of the card                                  }

      begin {PrintSuit}
      if s = spades then
         write('spades')
      else if s = diamonds then
         write('diamonds')
      else if s = clubs then
         write('clubs')
      else if s = hearts then
         write('hearts');
      end; {PrintSuit}


   begin {PrintDeck}
   for i := 1 to 52 do begin
      PrintValue(c[i].v);
      write(' of ');
      PrintSuit(c[i].s);
      writeln;
      end; {for}
   end; {PrintDeck}


begin
Seed(1234);                      {initialize the random number generator}
InitializeDeck(cards);           {get a new (sorted) deck of cards}
Shuffle(cards);                  {shuffle the deck}
PrintDeck(cards);                {print the shuffled deck}
end.
```

Solution to problem 7.5.

a.	r := 1.5;	Rule 1.
b.	r := 5;	Rule 2.
c.	r := red;	Incompatible.
d.	i := 45;	Rule 1.
e.	i := 1.5;	Incompatible.
f.	i := yellowHue;	Incompatible.
g.	i := ord(yellowHue);	Rule 1.
h.	c := red;	Rule 1.
i.	c := 0;	Incompatible.
j.	c := h;	Incompatible.
k.	c := c2;	Incompatible. Types are not the same unless the same type name is used.
l.	test1(i);	Incompatible. Test1 uses a var parameter, so the types must be the same.
m.	test1(4.5);	Incompatible. You cannot pass a constant as a var parameter.
n.	test2(i);	Rule 2.
o.	test3(red);	Rule 1.
p.	test3(h);	Incompatible.
q.	test4(v);	Rule 1.
r.	test4(v3);	Incompatible.
s.	v := v2;	Rule 1.
t.	v := v3;	Incompatible.
u.	s1 := s2;	Rule 3.
v.	s1 := s3;	Incompatible. S3 is not a string! A packed array of char is only a string if the first index is 0 and the second between 1 and 255, or if the first index is 1 and the second greater than 1.
w.	s1 := 'Hello!';	Rule 3.
x.	s1 := 'a';	Rule 4.
y.	s1 := ch;	Rule 4.
z.	ch := '';	Incompatible. Strings cannot be assigned to characters. The empty string is still a string.

Solution to problem 7.6.

```
{ This program shuffles a deck of cards, then prints the       }
{ results.                                                     }

program Shuffle(output);

type
                                        {suits of cards}
   suit = (spades, diamonds, clubs, hearts);
                                        {face value of cards}
   value = (two, three, four, five, six, seven, eight, nine, ten,
            jack, queen, king, ace);

   card = record                        {one card}
      s: suit;                          {card's suit}
      v: value;                         {card's value}
      end;
   deck = array[1..52] of card;         {deck of cards}
```

```
var
   cards: deck;                          {our deck of cards}


   function Random(max: integer): integer;

   { Return a pseudo-random number in the range 1..max.          }
   {                                                             }
   { Parameters:                                                 }
   {    max - largest number to return                           }
   {    color - interior color of the rectangle                  }

   begin {Random}
   Random := (RandomInteger mod max) + 1;
   end; {Random}


   procedure InitializeDeck(var c: deck);


   { Fills in the values to define a sorted deck of cards        }
   {                                                             }
   { Parameters:                                                 }
   {    c - deck of cards                                        }

   var
      i: integer;                        {loop variable}

   begin {InitializeDeck}
   for i := 1 to 13 do begin             {initialize the suit array}
      c[i].s := spades;
      c[i+13].s := diamonds;
      c[i+26].s := clubs;
      c[i+39].s := hearts;
      end; {for}
```

```
c[1].v := two;                          {initialize the first suit}
c[2].v := three;
c[3].v := four;
c[4].v := five;
c[5].v := six;
c[6].v := seven;
c[7].v := eight;
c[8].v := nine;
c[9].v := ten;
c[10].v := jack;
c[11].v := queen;
c[12].v := king;
c[13].v := ace;
for i := 14 to 52 do                    {copy the first suit to the}
   c[i].v := c[i-13].v;                 { remaining suits}
end; {InitializeDeck}


procedure Shuffle(var c: deck);


{ Shuffles the deck of cards                                        }
{                                                                   }
{ Parameters:                                                       }
{    c - deck of cards                                              }

var
   i: integer;                          {loop variable}
   j: integer;                          {card to swap with current card}
   tvalue: value;                       {temp value; for swap}
   tsuit: suit;                         {temp suit; for swap}

begin {Shuffle}
for i := 1 to 51 do begin
   j := Random(52 - (i-1)) + i-1;
   tvalue := c[i].v;
   c[i].v := c[j].v;
   c[j].v := tvalue;
   tsuit := c[i].s;
   c[i].s := c[j].s;
   c[j].s := tsuit;
   end; {for}
end; {Shuffle}
```

```
procedure PrintDeck(c: deck);


{ Prints the cards in order                                          }
{                                                                    }
{ Parameters:                                                        }
{    c - deck of cards                                               }

var
   i: integer;                        {loop variable}


   procedure PrintValue(v: value);

   { Print a value of a card                                         }
   {                                                                 }
   { Parameters:                                                     }
   {    v - value of the card                                        }

   begin {PrintValue}
   case v of
      two:     write('two');
      three:   write('three');
      four:    write('four');
      five:    write('five');
      six:     write('six');
      seven:   write('seven');
      eight:   write('eight');
      nine:    write('nine');
      ten:     write('ten');
      jack:    write('jack');
      queen:   write('queen');
      king:    write('king');
      ace:     write('ace');
      end; {case}
   end; {PrintValue}
```

```
      procedure PrintSuit(s: suit);

      { Print a suit of a card                                    }
      {                                                           }
      { Parameters:                                               }
      {    s - suit of the card                                   }

      begin {PrintSuit}
      case s of
         spades:   write('spades');
         diamonds: write('diamonds');
         clubs:    write('clubs');
         hearts:   write('hearts');
         end; {case}
      end; {PrintSuit}


   begin {PrintDeck}
   for i := 1 to 52 do begin
      PrintValue(c[i].v);
      write(' of ');
      PrintSuit(c[i].s);
      writeln;
      end; {for}
   end; {PrintDeck}


begin
Seed(1234);                    {initialize the random number generator}
InitializeDeck(cards);         {get a new (sorted) deck of cards}
Shuffle(cards);                {shuffle the deck}
PrintDeck(cards);              {print the shuffled deck}
end.
```

The length of the original program is $14B3 in hexadecimal, and the length of the new program is $1442. Converted to decimal, these lengths are 5299 bytes and 5186 bytes, respectively. The savings in size is 113 bytes.

Lesson Eight
Pointers and Lists

What is a Pointer?

By now, you have used two very powerful techniques to organize information in Pascal. Arrays are use to handle a large amount of information when all of the pieces are the same type. Records are used to collect different kinds of information into a single variable.

While these types are very powerful, there is one situation they do not handle well. In many programs, you don't know in advance how many pieces of information you need to deal with. For example, a program to manage a mailing list may have a few hundred entries when one person uses it, but several thousand for another person. On solution is to allocate an array that will be big enough to hold some maximum number, and leave it at that. Of course, that presents a problem, too. If one person has a computer with 1.25M of memory, they may be able to handle a mailing list with 7000 or 8000 entries. Unfortunately, the program would be to large to run on a computer with 768K, and would not make effective use of all of the memory in a 2M machine.

Of course, you may not ever intend to write a commercial application. On your own machine, you know how much memory you have, right? Well, that could be true, but fixed size arrays present other problems. Many programs have to handle more than one kind of data at the same time. For example, an adventure game might need one array for handling the rooms in a castle, and another array for keeping track of the various inhabitants. You can try to make effective use of memory by guessing in advance how big each array needs to be, but if you guess wrong, you could overflow one array while there is still plenty of room in the other.

And, of course, all of this ignores the fact that the current implementations of Pascal on the Apple IIGS limit the maximum size of a single array to 64K.

In all of these situations, the problem is that you know there is a lot of memory out there, but you don't always know, in advance, how much memory is available or exactly what you will need to use it for when the program runs. The amount of memory used by an array or record is determined when the program is written, and cannot be changed without recompiling the program. What we need is a way to ask for a chunk of memory while the program is running. Programmers call this *dynamically allocated memory*. Since the compiler doesn't know where the memory will be when you compile the program, or even how much will be allocated, you need some way of keeping track of the memory. That, in a nutshell, is what pointers are for. A pointer *points to* a memory location. In terms of the Pascal program, a pointer points to a variable. The variable can be a simple variable, like an integer or a real number; a record; an array; or even another pointer. In short, a pointer can point to a variable of absolutely any type.

I don't want to scare you off, but pointers tend to give beginners a lot of trouble. I would like to talk for a moment about what kind of trouble people have so you can watch out for these issues as you read through the lesson. We will try to deal with each of the issues.

Part of the reason people have trouble with pointers is that the idea of dynamically allocated memory is foreign to those of you who cut your teeth on BASIC. If pointers are a new concept for you, you should expect it to take some time before you become comfortable with them. Another factor is that pointers have their own operator that you must learn to use. A lot of people get confused by this operator, which controls when you are dealing with a pointer, and when you are dealing with the thing it is pointing to. Finally, there is a bit of magic about pointers in a high-level language. The other data types we have dealt with were definite, fixed structures. You could get a handle on what they do, and how they work. From a language like Pascal, there are some mysteries to how pointers work, since the language takes care of a lot of details. It is only from assembly language that you really see how pointers work – and, if you ever learn enough assembly language to learn how pointers work, you will probably follow in the footsteps of the vast majority of programmers, and return to a language like Pascal that handles all of those mucky details for you!

A realistic example of how pointers are used in a real program is well beyond what you are likely to understand at this point, so some of the first few examples will seem very simplistic and contrived. You will look at them and wonder why we are using pointers at all, when you can easily see better ways to write the program without a pointer. Well, you are right, but we will use some simple programs to get used to the mechanics of pointers. By the end of the lesson, though, you will be dealing with data structures that you could not handle with arrays. In the next few

lessons, we will start doing things with pointers that are very difficult to do with arrays. In some cases, in Pascal at least, some of the things we will do can't be done any other way than by the use of pointers.

Pointers are Variables, Too!

The first thing we need to explore is how to define a pointer. Pointer declarations are a little strange compared to the other data types. For all of the other data types, we have built up type declarations from English-like words and phrases. It is easy to look at

```
v: integer;
```

and see that v is an integer variable. With a little practice, it's just as easy to look at

```
a: array [1..10] of real;
```

and see that you have an array of real numbers.

Like an array, which must be an array of something, a pointer must point to something specific. Unlike an array, we don't use words to describe the pointer variable. Instead, we define a pointer by putting a ^ character right before the type, like this:

```
ip: ^integer;
```

The ^ Character

The ^ character that is used to define and access pointers varies a bit from one computer to another. On many computers, it shows up as an up-arrow character. Actually, this makes a lot of sense, since the character indicates a pointer, after all. You will see the up-arrow character used in some Pascal books. Don't let it bother you: it's just another way of printing the ^ character.

The variable ip is a variable, just like any other. It just has an odd type. The type of ip is "pointer to integer." There are only two things that you can do with this variable in Standard Pascal: assign it to another pointer variable or compare it for equality or inequality with another pointer value. Of course, for either operation, the pointers must be type compatible. For example, the following program is legal in Pascal:

```
program AssignPointer;

var
   ip,jp: ^integer;

begin
ip := jp;
end;
```

The pointer is virtually worthless without the ^ operator. The ^ operator, appearing right after the pointer variable, gives us the value the pointer points to, rather than the pointer itself. For example, the assignments shown in the following program are legal, although the program itself has some problems. (Do not run this program!)

```
program Pointers(output);

var
   i,j: integer;
   ip: ^integer;

begin
j := 4;
ip^ := j;
i := ip^;
writeln(i);
end;
```

Let's step through the program, looking at what it is doing. First, we assign the value 4 to j. Nothing is new there; you've done that sort of thing dozens of times. The next line, though, assigns the integer j to the value pointed to by ip. Keep in mind that we are not assigning a value to the variable ip, we are assigning a value to the variable *pointed at* by the variable ip. That's what the ^ operator does for us; it tells the compiler that we want the value pointed at, not the pointer.

If that's confusing, think about how arrays work for a moment. If a and b are type compatible arrays of integer, then the assignment

```
a := b;
```

copies the contents of the array b into the array a. This is very, very different from the assignment

```
a[1] := 4;
```

which copies the value 4 into one position of the array. The [1] tells the compiler to use a specific value of the array, not the array itself. The ^ operator is doing something similar for a pointer variable. It tells the

compiler to copy the value into the variable pointed to by ip, not into the pointer ip itself.

The next line,

```
i := ip^;
```

uses the same idea to assign the value pointed to by ip to the variable i. Finally, the value of i is printed. The value should be 4.

Unfortunately, this program has a very, very serious flaw. In is a very common error in programs that use pointers. In fact, it is one of the most common causes of crashes on the Apple IIGS, in any kind of program. Did you catch the flaw? If you've never seen pointers before, probably not.

What does ip point to?

What if ip points to the location in memory that turns on your floppy disk drive? The disk drive would start to spin.

What if ip happens to point to memory allocated by the GS/OS operating system that holds a block of a data file? When you save the file, it will have some garbage information in it.

What if ip points into the middle of your program? Your program may crash.

What if ip points to the locations PRIZM is using to store the characters in your source file? You will see garbage in the file.

Worst of all, what if ip points to some memory that isn't being used for anything? You might think the program works, and pass it around to friends. It could then do all of these nasty things to *their* computer. This, of course, is not a good way to keep friends.

Allocating and Deallocating Memory

In short, pointers are no good without a way to get some memory for them to point to. Pascal gives us a procedure called new to get some new memory. When you are finished with the memory, the procedure dispose can be used to get rid of the memory. Both procedures need the name of the pointer for which you want to allocate or deallocate memory. We can change our program from the last section into a safe one using these procedures. This program is one you can run!

```
program Pointers(output);

var
   i,j: integer;
   ip: ^integer;
   begin
   new(ip);
   j := 4;
   ip^ := j;
   i := ip^;
   writeln(i);
   dispose(ip);
   end.
```

When this program runs, it starts by making a call to new. This procedure performs some advanced magic. The result is that, after the call, two bytes of memory have been obtained. The exact process

How New and Dispose Work

The process used to allocate and deallocate dynamic memory is a bit involved, and has nothing in particular to do with the way you write your Pascal program, but it is interesting.

One of the basic parts of the Apple IIGS operating system is the memory manager. The memory manager is responsible for finding free memory and giving it to the various programs in the computer. Even if your program is the only one you think is running, it turns out that many other programs are calling the memory manager to get memory, too. The GS/OS disk operating system calls the memory manager, as do many of the toolkits. PRIZM is calling the memory manager to get space for your program. Many desk accessories call the memory manager. Some of them may even install interrupt handlers, which can be running while your program is doing something else.

When you call new for the first time, the program makes a call to the memory manager to get a 4K block of memory. This memory is then subdivided into smaller and smaller pieces, dividing the block in half each time, until the program gets a chunk of memory of about the right size. In our program you need two bytes to hold the integer, and the library subroutine allocating the memory needs four bytes to keep track of all of the small pointers, so a total of eight bytes is actually taken from the 4K chunk of memory. (Remember, the number of bytes will be a power of two.) This method tends to waste a few bytes of memory now and then, but it turns out that it is very fast. It has some other technical advantages, too, that we won't go into here.

When you call dispose at the end of the program, the small block of memory is deallocated. Since it was the only piece of memory being used in the 4K block, the 4K block is also returned to the memory manager, where it can be reused by other programs.

involved in getting this memory is a bit involved, and not particularly important to you, the Pascal programmer. If you are curious, the sidebar gives an overview of the process. In any case, this memory is safe. It belongs to your program, and no other correctly written program will disturb it.

Just before the program ends, you see the dispose procedure. This procedure goes through a complicated mechanism that gets rid of the two bytes of memory. After calling dispose, the memory does not belong to your program anymore. It could be reused within 1/60th of a second by an interrupt routine, such as the software that controls the mouse and keyboard in PRIZM. Even if it isn't reused, because of the process used to allocate and deallocate memory, the location ip points to doesn't contain 4, anymore. In short, once you call dispose, the memory isn't yours anymore, and you should not access or change the value pointed at by ip.

This is a very short program, but it is worth typing it in and running it through the debugger. Start the program in step mode and bring up the variables window. Go ahead and enter i and j if you like, but be sure and enter ip. This will show you the value of the pointer itself. Like all variables declared at the program level in ORCA/Pascal, it starts with a value of zero. The value is displayed in hexadecimal notation, but it is still zero. Now add ip^ to the variables window. The debugger will display the value being pointed to by ip. Again, the important point is that ip is a variable, but the value you really want to see and make use of is what it points to. That, as you know, is ip^. Stepping through the program, you can see new allocate new memory, and you can see the assignment setting the value being pointed to. Dispose doesn't change the value of ip, but you will see the value of ip^ change.

Problem 8.1. A pointer can point to any variable type. Use that fact to change the program shown in this section to allocate a pointer to a real number. Assign the value 1.2 to the location pointed to by the pointer, and print the result. Do all of this without an intermediate real variable; in other words, assign the value directly to the value pointed at by the pointer, and use the pointer with the ^ operator in the writeln statement.

Problem 8.2. You can, of course, use ip^ anywhere that you could use an integer variable. Making use of that fact, write a program to add two numbers and print the result. The only variables you should define are three pointers, ip, jp, and kp. Be sure and allocate memory for all of them using new, then assign 4 to the first, and 6 to the second. Add the two values together and save them at kp^, then print the result. Be sure and follow your mother's advice, and clean up after yourself by calling dispose to deallocate the memory areas reserved by the calls to new.

Linked Lists

So far, all of our programs have used a pointer to a single variable. That's about as useful as your mother on a hot date. A single variable is easier to use, takes less space, produces a smaller program, the resulting program runs faster, and there is no chance of stepping on someone else's memory because you forgot to use new to allocate the memory. We used arrays to organize a fixed number of values into a data structure that was easier to use. The equivalent for a pointer is one of the many forms of a linked list.

Basically, a linked list is a series of connected records. Each of the records in the linked list contains, among other things, a pointer. The pointer points to another record in the list. A single pointer variable in the program points to the first record in the linked list.

For our first look at a linked list, we will create a list of integers. The record, then, must have a pointer to the next record, and an integer. It looks like this:

```
type
   listPointer = ^listRecord;
   listRecord = record
      next: listPointer;
      i: integer;
      end;

var
   list: listPointer;
   temp: listPointer;
```

If you look closely at these definitions, you will see something you have never seen before in a Pascal program. The definition of listPointer uses the type listRecord. That's not unusual; your other pointer definitions used a type, too. The unusual thing is that listRecord has not been defined when the definition of listPointer uses it! This is absolutely the only place in the Pascal language where you can make use of something before defining it. The type of variable that a pointer points to can be declared after the pointer itself is declared. The reason for this exception is that there would be no way to declare a linked list without it. The type of next must be a pointer to a listRecord, and so must list. But since types are only compatible if the names are the same, and are not compatible just

because the type has the same structure, the following declarations don't work.

```
{These don't work: list is not }
{type compatible with next.    }

type
   listRecord = record
      next: ^listRecord;
      i: integer;
      end;

var
   list: ^listRecord;
```

With these definitions, we can start to create a linked list. For each element in the list, we will need to call new to get space for a new record, and then place a value into the integer, like this:

```
new(temp);
temp^.i := 4;
```

Look carefully at the assignment that places a 4 in the record. The characters ^. may seem confusing at first, but they are the same simple ideas you are used to, combined to do something a bit more complicated. Temp, of course, is a pointer, so to put a value into temp, we need to use the ^ operator. Temp^ points to a record. To place a value into the variable i within a record, we add .i. The whole expression, temp^.i, then, refers to the integer variable i, located inside a record that is pointed to by the pointer temp. That's a complicated concept, but it is simple when you break it down into parts, reading the expression one symbol at a time from left to right, the way the compiler does.

At this point, we have a dynamically allocated record with an integer value in it. The pointer in the record still does not point to anything. The next step is to add this record to the list of records that the variable list points to.

```
temp^.next := list;
list := temp;
```

On the first line, we are assigning a value to the pointer in our new record. The value we are assigning is list; list points to the first element currently in the list. We really don't know how many things are in the list at this point. There may not be any, or there may be several thousand. The beauty of the linked list, though, is that we don't have to know! It doesn't matter at all how many things are already in the linked list.

The second line assigns temp to list. The first thing in the list, at this point, is our new record. Our record contains an integer variable with a value of 4, and a pointer to the rest of the list.

The next thing we need to learn is how to take something off of the list. Let's say that we want to remove the first item. Basically, then, we reverse the process of putting a record into the list, like this:

```
temp := list;
list := temp^.next;
```

There is one more detail that we need to deal with before we can use these ideas to write a program. So far, we have ignored the issue of the end of the list. How do we know when we get to the end of the list? We could keep a counter, but actually, there is a better way. It involves the use of a predefined pointer constant called nil. Nil is type compatible with any pointer type. You can set a pointer to nil or compare a pointer to nil. Like true, false and maxint, the constant nil is automatically defined in all implementations of Pascal. By convention, it is used to mean that the pointer doesn't point to anything, and that it how we mark the end of our list. By initializing list to nil at the start of the program, and checking to see if list is nil before removing an item from the list, we can tell when there is nothing in the list.

Stacks

Using what we now know about linked lists, we can create our first program, shown in listing 8.1.

We have already talked about all of the ideas in this program, this is just the first time you have seen them all in one place. Looking through the program, the first step is to get a list of numbers. GetList does this, reading numbers using familiar methods until you enter 0. For each number, GetList allocates a new record, saves the number in the record, and puts the record in the list.

PrintList loops for as long as there are entries left in the list. Each time through the loop, the top record in the list is removed from the list, the value is printed, and the memory used by the record is dumped. Notice how the PrintList procedure cleans up after itself. The memory used by every record is carefully disposed of after we are finished with the record. This is an important step in a program that uses dynamic memory. If you forget to dispose of some of the memory in a few places, the memory areas will eventually fill up, and there won't be any free memory for the new calls.

It is very important to understand exactly how this program works, since the ideas used in this program

form the basis for many of the fundamental techniques in modern programming practice. Stop now, and type in the program. Run the program with the following input:

Listing 8.1

```
{ This program reads in a list of integers, and then prints    }
{ them in reverse order.  The program stops when a zero value  }
{ is read.                                                     }

program Reverse(input, output);

type
   listPointer = ^listRecord;           {list pointer}
   listRecord = record                  {list element}
      next: listPointer;
      i: integer;
      end;

var
   list: listPointer;               {points to the top item in the list}


   procedure GetList (var list: listPointer);

   { Read a list from the keyboard.                            }
   {                                                           }
   { Parameters:                                               }
   {    list - pointer to the head of the list                 }

   var
      i: integer;                       {variable read from input}
      temp: listPointer;                {work pointer}

   begin {GetList}
   list := nil;                         {initialize the list pointer}

   repeat
      readln(i);                        {read a value}
      if i <> 0 then begin             {if not at the end of the list...}
         new(temp);                     {allocate a record}
         temp^.i := i;                  {place i in the record}
         temp^.next := list;            {put the record in the list}
         list := temp;
         end; {if}
   until i = 0;
   end; {GetList}

(continued)
```

```
1
2
3
4
0
```

The program responds with this:

```
4
3
2
1
```

This may not have been exactly what you expected. What happened is this: when the program creates the list, each new element is added on top of the old list. As the program retrieves records from the list, the last one added is removed first. This mechanism is called a stack. The common analogy is to think of it like a stack of plates. You pile the list elements up on top of one another. To get one back, you pull the top record off of the stack.

Just as a footnote, I should ward you about terminology buffs. Many high school teachers, a few college professors, and even an occasional book author figure that the way to become a good programmer is to learn a bunch of arcane words. It is true that you need some new words, like dynamically allocated memory, to describe new concepts, but these terminology buffs want you to know that a stack is called a LIFO data structure, for Last In, First Out. Let's face it, they write the tests, so you better know the term if you want to get a good grade in a class. Be warned, though: if you walk up to a group of programmers at a conference and start babbling about LIFO data structures, you will find a wide gap forming around you. A few people will glance at your shirt pocket, looking for the pencil holder, or examine the thickness of your glasses. In real life, these things are called stacks.

Stacks are a very flexible data structure. They are used in a wide variety of applications. A stack is appropriate any time you need to collect a large amount of information, especially if you don't particularly care in what order you use the information, or for the occasional case when you want to handle the most recent piece of information first. Stacks are also frequently used as a part of a more complicated data structure, like a hash table. We'll look at complex data structures like this later in the course. Stacks are used in such diverse applications as burglar alarms, data

(continuation of Listing 8.1)

```
      procedure PrintList (var list: listPointer);

      { Print a list.                                              }
      {                                                            }
      { Parameters:                                                }
      {    list - pointer to the head of the list                  }

      var
         temp: listPointer;               {work pointer}

      begin {PrintList}
      while list <> nil do begin
         temp := list;                    {remove an item from the list}
         list := temp^.next;
         writeln(temp^.i);                {write the value}
         dispose(temp);                   {free the memory}
         end; {while}
      end; {PrintList}

   begin
   GetList(list);                         {read a list}
   PrintList(list);                       {print a list}
   end.
```

bases, mailing lists, operating systems, and arcade games.

There are many variations on the basic ideas covered in this section. Some of these are explored in the problems. I highly recommend that you work both of these problems.

Problem 8.3. Many applications require you to process the information in a list from back to front. In some cases, you know this in advance, and a slightly different form of a list is used, called a queue. That situation is covered in the next section. In other cases, though, you may not know that the list needs to be reversed in advance, or you may need to process the list in both orders in different parts of the program. In a case like that, you need to be able to reverse the list.

Reversing a list is really quite easy. To do it, you use two lists. The new list starts out empty. You then loop through the old list, just like we do in the PrintList procedure, but instead of printing the value and disposing of the record, you add the record to the new list.

Write a procedure to reverse the order of a list. Use this procedure in the sample program so it prints the numbers in the same order they are read.

Problem 8.4. In some applications, we read in a list, then scan the list repeatedly, looking for records with certain characteristics. For example, in a burglar alarm, we might use one procedure to add new alarms to a list. Another might repeatedly scan the list, looking for fires. If no fires were found, the list could be rechecked for broken windows, and so on.

Implement this idea in our sample program by counting the number of times a particular number appears in the list. Use a for loop to loop from 1 to 5. For each value, scan the list, incrementing a counter if the number is found. Print a table of the results.

Try this program at least two times. The first time, enter zero immediately. The second time, use this data:

```
1
2
3
4
5
2
3
4
5
3
4
5
4
5
5
```

The results should be one one, two twos, and so forth.

Hint: To scan a list, set a pointer to the head of the list. Use a while loop to loop until this pointer is nil. At the end of the while loop, set the pointer to the next record, like this:

```
temp := temp^.next;
```

Queues

Another commonly used form of a list is the queue. A queue looks just like a stack, but it is formed differently. A queue is used when you want to process information in the same order it is read, so instead of adding new records to the beginning of the list, you want to add them to the end of the list. In a sense, the records are lined up, and processed on a first-come, first served basis. The terminology freaks call a queue a FIFO list, for First In, First Out, but again, don't embarrass yourself in a crowd by talking about stuff like that.

There are three basic ways to form a queue. If all of the information is read in first, then processed, you could just use the simple stack to read the data, then reverse the order of the list, like we did in problem 8.3. In many programming situations, though, you read some data, process a little bit, read some more, and so forth. In those cases, you need to build the list in the proper order.

One way to build a queue is to keep a second pointer, which we will call last. This pointer starts at nil, like the pointer that points to the first member of the list. When we add the first element to the list, the pointer last is set to the value of the new pointer. The next pointer in the new record is always set to nil.

From then on, we add a new record by setting the next pointer in the record pointed to by last to point to the new record, and then set last to point to the new record. In Pascal code, then, we set the list up like this:

```
list := nil;
last := nil;
```

To add a record to the end of the list, we check to see if the record is the first one in the list. If so, we set both last and list to point to the new record. If not, we chain the record to the end of the list.

```
if list = nil then begin
   list := temp;
   last := temp;
   end {if}
else begin
   last^.next := temp;
   last := temp;
   end; {if}
```

Of course, sense both branches of the if statement assign temp to last, we can make the program shorter, and still do the same thing, by pulling the assignment outside of the if statement, like this:

```
if list = nil then
   list := temp
else
   last^.next := temp;
last := temp;
```

We also don't actually make use of last before it is assigned a value for the first time, so setting it to nil when we initialize the list is also unnecessary.

Problem 8.5. You probably saw this one coming. Change the GetList procedure from the sample in the last section so it forms a queue instead of a stack. Use the mechanism described in this section to do it.

Running Out Of Memory

What happens if you ask for more memory, but none is available? If this happens, the program will stop with a run-time error, and tell you that you ran out of memory. Just for fun, the following program does this on purpose. Incidentally, when your program stops, all of the memory you allocated is disposed of, even if you missed some, so this program will not keep the memory after it quits.

```
program OutOfMemory;

type
   a = array[1..10000] of integer;

var
   p: ^a;

begin
repeat
   new(p);
until false;
end.
```

This program uses a repeat loop to form an intentional infinite loop. The program will keep going until you stop it or an error occurs. Each time through the loop, we allocate a new array of 10,000 integers, taking up 20,000 bytes of memory. It doesn't take long to run out at that rate. We are loosing track of the memory we allocate, of course, but since our purpose is to run out, it doesn't really matter.

Incidentally, there are occasional uses in programs for an infinite loop. They are rare, but they do exist. Looping until false is one easy way to do this. After all, the condition false will never be met!

Problem 8.6. Add a counter to the loop, and print its value each time through the loop. After the program fails, check the shell window to see how many 20,000 byte blocks of memory your program was able to allocate before it went belly up.

Lesson Eight
Solutions to Problems

Solution to problem 8.1.

```
program Pointers(output);

var
   rp: ^real;

begin
new(rp);
rp^ := 1.2;
writeln(rp^);
dispose(rp);
end.
```

Solution to problem 8.2.

```
program Pointers(output);

var
   ip,jp,kp: ^integer;

begin
new(ip);
new(jp);
new(kp);
ip^ := 4;
jp^ := 6;
kp^ := ip^ + jp^;
writeln(kp^);
dispose(ip);
dispose(jp);
dispose(kp);
end.
```

Solution to problem 8.3.

```
{ This program reads in a list of integers, and then prints    }
{ them in the order read.  The program stops when a zero value }
{ is read.                                                     }

program Echo(input, output);

type
   listPointer = ^listRecord;           {list pointer}
   listRecord = record                  {list element}
      next: listPointer;
      i: integer;
      end;

var
   list: listPointer;                {points to the top item in the list}


   procedure GetList (var list: listPointer);

   { Read a list from the keyboard.                            }
   {                                                           }
   { Parameters:                                               }
   {    list - pointer to the head of the list                 }

   var
      i: integer;                       {variable read from input}
      temp: listPointer;                {work pointer}

   begin {GetList}
   list := nil;                         {initialize the list pointer}

   repeat
      readln(i);                        {read a value}
      if i <> 0 then begin              {if not at the end of the list...}
         new(temp);                     {allocate a record}
         temp^.i := i;                  {place i in the record}
         temp^.next := list;            {put the record in the list}
         list := temp;
         end; {if}
   until i = 0;
   end; {GetList}
```

```
   procedure Reverse (var list: listPointer);

   { Reverse the order of a list.                                  }
   {                                                               }
   { Parameters:                                                   }
   {    list - pointer to the head of the list                     }

   var
      temp: listPointer;                {work pointer}
      list2: listPointer;               {pointer to the new list}

   begin {Reverse}
   list2 := nil;                        {nothing in list2, yet}
   while list <> nil do begin
      temp := list;                     {remove an item from the list}
      list := temp^.next;
      temp^.next := list2;              {add it to list2}
      list2 := temp;
      end; {while}
   list := list2;                       {return the new list}
   end; {Reverse}


   procedure PrintList (var list: listPointer);

   { Print a list.                                                 }
   {                                                               }
   { Parameters:                                                   }
   {    list - pointer to the head of the list                     }

   var
      temp: listPointer;                {work pointer}

   begin {PrintList}
   while list <> nil do begin
      temp := list;                     {remove an item from the list}
      list := temp^.next;
      writeln(temp^.i);                 {write the value}
      dispose(temp);                    {free the memory}
      end; {while}
   end; {PrintList}

begin
GetList(list);                          {read a list}
Reverse(list);                          {reverse the order of the list}
PrintList(list);                        {print a list}
end.
```

Solution to problem 8.4.

```
{ This program reads in a list of integers, then counts the    }
{ number of times each integer from 1 to 5 appears in the      }
{ list.                                                        }

program Count(input, output);

type
   listPointer = ^listRecord;           {list pointer}
   listRecord = record                  {list element}
      next: listPointer;
      i: integer;
      end;

var
   list: listPointer;                {points to the top item in the list}


   procedure GetList (var list: listPointer);

   { Read a list from the keyboard.                            }
   {                                                           }
   { Parameters:                                               }
   {    list - pointer to the head of the list                 }

   var
      i: integer;                       {variable read from input}
      temp: listPointer;                {work pointer}

   begin {GetList}
   list := nil;                         {initialize the list pointer}

   repeat
      readln(i);                        {read a value}
      if i <> 0 then begin              {if not at the end of the list...}
         new(temp);                     {allocate a record}
         temp^.i := i;                  {place i in the record}
         temp^.next := list;            {put the record in the list}
         list := temp;
         end; {if}
   until i = 0;
   end; {GetList}
```

```
   procedure CountList (list: listPointer);

   { Count the elements in the list.                          }
   {                                                          }
   { Parameters:                                              }
   {    list - pointer to the head of the list                }

   var
      temp: listPointer;                {work pointer}
      counts: array[1..5] of integer;   {array of frequency counts}
      i: integer;                       {loop counter}

   begin {CountList}
   for i := 1 to 5 do                   {set the counts to 0}
      counts[i] := 0;
   temp := list;                        {do the count}
   while temp <> nil do begin
      if (temp^.i >= 1) and (temp^.i <= 5) then
         counts[temp^.i] := counts[temp^.i] + 1;
      temp := temp^.next;
      end; {while}
   for i := 1 to 5 do                   {print the results}
      writeln(i, counts[i]);
   end; {CountList}

begin
GetList(list);                          {read a list}
CountList(list);                        {print the frequencies}
end.
```

Solution to problem 8.5.

```
{ This program reads in a list of integers, and then prints    }
{ them in the order read.  The program stops when a zero value }
{ is read.                                                     }

program Echo(input, output);

type
   listPointer = ^listRecord;           {list pointer}
   listRecord = record                  {list element}
      next: listPointer;
      i: integer;
      end;

var
   list: listPointer;                {points to the top item in the list}
```

```
procedure GetList (var list: listPointer);

{ Read a list from the keyboard.                                    }
{                                                                   }
{ Parameters:                                                       }
{    list - pointer to the head of the list                         }

var
   i: integer;                        {variable read from input}
   temp: listPointer;                 {work pointer}
   last: listPointer;           {points to the last record in the list}

begin {GetList}
list := nil;                          {initialize the list pointer}

repeat
   readln(i);                         {read a value}
   if i <> 0 then begin               {if not at the end of the list...}
      new(temp);                      {allocate a record}
      temp^.next := nil;              {no forwar link}
      temp^.i := i;                   {place i in the record}
      if list = nil then              {put the record in the list}
         list := temp
      else
         last^.next := temp;
      last := temp;
      end; {if}
until i = 0;
end; {GetList}


procedure PrintList (var list: listPointer);

{ Print a list.                                                     }
{                                                                   }
{ Parameters:                                                       }
{    list - pointer to the head of the list                         }

var
   temp: listPointer;                 {work pointer}

begin {PrintList}
while list <> nil do begin
   temp := list;                      {remove an item from the list}
   list := temp^.next;
   writeln(temp^.i);                  {write the value}
   dispose(temp);                     {free the memory}
   end; {while}
end; {PrintList}
```

```
begin
GetList(list);                          {read a list}
PrintList(list);                        {print a list}
end.
```

Solution to problem 8.6.

```
program OutOfMemory(output);

type
   a = array[1..10000] of integer;

var
   p: ^a;
   count: integer;

begin
count := 0;
repeat
   writeln(count);
   count := count+1;
   new(p);
until false;
end.
```

The number that is reported by this program will vary wildly, depending on how much memory you have, what version of the system disk you are using, how many drivers and desk accessories you have, what files are open on the desktop, and so forth.

Lesson Nine
Files

The Nature of Files in Pascal

A lot of fun and useful programs never save a file to disk or read from a disk file. Arcade games, some adventure games, many scientific and engineering calculations, and all of the programs you have written so far in this course all read data from the keyboard, or do calculations based on internal values. On the other hand, the vast majority of programs do read and write disk files. Spread sheets, word processors, data base programs, many games, ORCA/Pascal itself – all of these programs read and write files. This lesson introduces files as used in the Pascal language.

If you already know some other computer language, the way files are implemented in Pascal is almost certainly very different from what you are used to. That is because the Pascal language does not deal directly with the concept of disks, tape drives, networks, device drivers, or any of these other topics that languages like assembly language and C deal with in a fairly direct way. In Pascal, files are treated as a data structure. In fact, files are very similar to a queue. (Queues are a form of linked list, covered in the last lesson.) In Pascal, a file is a sequential access data structure. That means that you write the file starting with the first element, writing each successive element until you reach the end of the file. You read a file the same way, starting at the beginning.

File Variables

As with any other data structure, the first thing you must learn to do is to declare a file variable. A file variable definition looks like this:

```
variable_name: file of file_type;
```

There is nothing special about the variable name. You pick it just like you would for any other variable. There isn't much special about the file type, either. A file can be absolutely any type except another file type. This restriction also carries over to records and pointers used as file types. You can use any record type as a file type, so long as the record does not contain a file type. A file can also be a pointer to any type; again, so long as the pointer does not point to a file or something that contains a file. The reverse is allowed, though. You are allowed to include a file type in a record, as long as you don't use that record as the type of yet another file.

Pascal does have one other prohibition that applies to file variables. You can't assign one file variable to another, nor can you pass a file variable as a value parameter. You can, however, pass a file variable as a var parameter. These restrictions give you a safety net, preventing some really nasty bugs that can occur in languages like C.

Writing to a File

In theory, files can be of any length. Basically, that means that there is no fixed limit to the number of things you can put in a file. Of course, there's no free lunch. The information you stuff into a file has to be saved somewhere. In the case of the Apple IIGS, it is saved on one of the devices GS/OS handles. This is usually a floppy disk or hard disk, but it can also be a network, a tape drive, or anything else that GS/OS recognizes.

To write a value to a file, you need to initialize the file for output. This is done with the rewrite command. For a file variable f, the rewrite command looks like this:

```
rewrite(f);
```

The file variable, in this case f, is actually a pointer. It points to the next value to be written to the file. Pascal uses the put procedure to actually write a value to the file. Putting these ideas together, we can create a small program to write the integers 1 to 10 to disk, like this:

```
{ Write some numbers to a file. }
program WriteIt;

var
   f: file of integer;
   i: integer;

begin
rewrite(f);
for i := 1 to 10 do begin
   f^ := i;
   put(f);
   end; {for}
end.
```

Reading from a File

Reading a file is just as easy. To read from a file, you first have to initialize it for input. In terms used in other programming languages, you would open the file for input. The reset procedure does this; like rewrite, you pass the file variable as a parameter. Pascal's get procedure is used to read a value from the file. The value is stored in the location pointed at by the file variable; you access the value by dereferencing the pointer using the ^ operator, just like you did with pointer variables in the last lesson.

One thing that most people miss the first few times they use reset and get is that reset not only opens the file for input, it also does a get. After reset, the file variable already points to the first value that was stored in the file.

A file can't be open for input and output at the same time. You can't, for example, do a put, then immediately do a get. Instead, you use the reset procedure to open the file for input. The reset procedure closes the file if it was already open for output, then opens the file for input. If you use rewrite to reopen the file for output, all of the values that were in the old file are deleted. Naturally, you can use reset to reread the file as many times as you like.

Putting these ideas together, we can extend out program from the last section to read the file we wrote, printing the values to the shell window.

```
{ File I/O demo }
program IO(output);

var
   f: file of integer;
   i: integer;

begin
{open the file for output}
rewrite(f);

{write 10 values to the file}
for i := 1 to 10 do begin
   f^ := i;
   put(f);
   end; {for}

{open the same file for input}
reset(f);

{read and echo the values}
for i := 1 to 10 do begin
   writeln(f^);
   get(f);
   end; {for}
end.
```

Problem 9.1. The file variable can be any type. Demonstrate this by defining a file of strings. Writing the names of the months in the year to the file. Next, open the file for input, and read and write the strings, using the same mechanism shown in the sample.

Closing a File

If you know another programming language, you might be bothered a bit by the fact that the sample program doesn't close the file. We opened it for output with rewrite, then closed it and reopened it for input with reset, but we never closed the file at the end of the program.

Actually, we didn't have to. In fact, Standard Pascal doesn't even have a close command. A file gets closed automatically when the program stops. If the file variable is defined in a procedure, like it is in the following example, the file gets closed before the procedure returns to the main program. In short, the file gets closed when the procedure, function or program that declared the file finishes executing.

```
procedure WriteFile;

var
   f: file of integer;
   i: integer;

begin {WriteFile}
rewrite(f);
for i := 1 to 10 do begin
   f^ := i;
   put(f);
   end; {for}
{the file is closed here}
end; {WriteFile}
```

Files on Disk

Well, as Alice noted in her famous trip to Wonderland, things are getting curiouser and curiouser. Here we are, talking about disk files and file I/O, reading and writing to files, and not once have we mentioned a file name, or even where the file is physically written. After all, when you run a program like a word processor, the file that holds the text you type has a specific name, and is located in a specific place, on a specific disk. Our sample just wrote a file. If you are using a floppy disk, you probably noticed the disk spin as the file was written. It also read from the file. But where is the file? For all you know from reading the program, it might not even be on a disk. If it is, you don't have a clue yet where it actually is, or what the name is.

Well, that's how files work in Pascal. In fact, the compiler is free to implement the program you just wrote without even writing anything to disk. If the compiler chooses to, it can write the file to memory, and read the values back from memory.

The problem, of course, is that Pascal works on many different computers, and is designed to work on future computers that may have designs very different from the ones we use now. When Pascal was originally developed, most computers had punched card readers and line printers for I/O devices, not keyboards and CRT screens. I have worked on computers that didn't have disks at all. Pascal files have to be able to deal with disks, networks, punched cards, paper tape, modems, keyboards, mice, RAM disks, RAM itself, and even a few exotic devices that haven't been created yet. The way Pascal handles this is to define some basic file capabilities, like get and put, but to leave how a file is connected to a physical storage device completely up to the implementor of a particular compiler.

Fortunately, on today's desktop computers, there is a fairly common standard for assigning a name to a disk file. Both reset and rewrite accept an optional second parameter. This parameter is the name of the file you want to read from or write to, coded as a string. On the Apple IIGS, the name can be anything that the native file system will accept. GS/OS is a pretty flexible file system; it can be expanded to handle all sorts of disks. Currently, though, file names obey the following rules:

1. A file name starts with an alphabetic character.
2. The remainder of the file name is made up of alphabetic characters, numeric digits, and periods.
3. A file name must have at least one character, and no more than 15 characters.
4. GS/OS does not distinguish between uppercase characters and lowercase characters. In other words, the file names MYFILE, MyFile and myfile all refer to the same file on disk.

We can use this information to split our original sample program up into two programs. For variety, we'll change it to write the alphabet this time, then read it back in. First, then, the program to write the alphabet to a disk file.

```
{ Write the alphabet to a file. }
program WriteIt;

var
   f: file of char;
   ch: char;

begin
rewrite(f, 'Alphabet');
for ch := 'A' to 'Z' do begin
   f^ := ch;
   put(f);
   end; {for}
end.
```

And here is a program that will read this file and write the characters to the shell window.

```
{ Read the alphabet file. }
program ReadIt(output);

var
   f: file of char;
   i: integer;

begin
reset(f, 'Alphabet');
for i := 1 to 26 do begin
   write(f^);
   get(f);
   end; {for}
writeln;
end.
```

This is a file you can see on the disk. The Finder can see it, for example. You can also see it from ORCA/Pascal by clicking on the shell window and typing CAT, followed by pressing the RETURN key. Several files will be listed, but among them is a file called ALPHABET. This is the file you just created and read. If you know how to read a text catalog listing, you can also see that the file is a BIN file (binary file), even though we created a file of characters. We'll look at how to create TXT files (text files) in a moment.

Incidentally, if you don't know how to read a disk catalog, and would like to learn, you can refer to the ORCA/Pascal reference manual. Read the description of the CATALOG command, which you can find listed in the index.

We can now go full circle, returning to the original sample program that did not use a file name. You saw the disk being accessed; obviously the file went somewhere. When you don't give Pascal a file name, it creates one for itself. The name of the file is SYSPASxxxx, where xxxx is a number assigned by the program as you open files. In this case, the number is 1, so the complete file name is SYSPAS0001. The ALPHABET file was placed in the current directory, which is the same place where your program is. These temporary work files that are created for you are placed in another location, called the work prefix. The prefix number for the work prefix is 3. You can find the SYSPAS0001 file by typing

```
cat 3
```

from the shell window. This command catalogs prefix 3. The exact location of prefix 3 depends on how you have set up ORCA/Pascal.

Directories, Path Names and Folders

You may have bought your Apple IIGS because it has that wonderful, easy to use desktop interface that Apple is famous for, but unlike the Macintosh, you can own an Apple IIGS for less than the price of a car, and still get color. If so, good choice. You may not ever want to deal with icky text-based systems like those found on the hard-to-use IBM PC and its cousins. Good choice, again. Unfortunately, when you are programming, you still have to deal with files using names. You can't "point and click" inside of a program. This section gives you a brief overview of how those names work, in terms of the icons and folders you are used to in desktop programs. If you already know how files are named, what a path name is, and so forth, feel free to skip to the next section.

I will assume that you are already familiar enough with your computer to move around using a desktop program like the Finder. In the Finder, the first thing you see on the desktop is a list of the disks, lined up along the right-hand side of the screen. Below each disk is a name. To give the name of a disk on a Pascal program, you use exactly the same name, but you start it off with a slash character. For example, the disk where the Pascal compiler is located is called ORCA.PASCAL. In a file name, you would type

```
/ORCA.PASCAL
```

Double-click on the disk icon and the Finder will open a window showing the various files and folders. For example, one of the folders is called SAMPLES. If you want to look at a file in the samples folder, you add the name of the folder to the disk name, separating the two with another slash, like this:

```
/ORCA.PASCAL/SAMPLES
```

If the folder contains other folders, you can repeat this process, adding the new folder name to the name you have already accumulated.

Eventually, you will get to the right folder, and you will see the file you want to read. Let's assume that you want to read the file BULLSEYE.PAS from the samples folder. Once again, you tack the file name onto the names you already have, using a slash to separate the file name from the name of the disk and folder.

```
/ORCA.PASCAL/SAMPLES/BULLSEYE.PAS
```

The result is called a full path name. It specifies exactly what file you want to read or write.

In our alphabet example, we just gave a file name. Of course, the computer still writes to a specific place on the disk. When you leave off the name of the disk and any folders, the file name is added to a default directory called prefix 0. Prefix 0 is also called the default prefix. In a desktop program, you set the default prefix by using one of the file related commands, like open. When you click on the disk button, it changes the default prefix to the name of a new disk. Opening a folder on the disk adds the name of the folder to the default prefix. Closing a folder, of course, removes the name of the folder from the default prefix. The computer remembers this location, and uses it for all files that only have a file name. That's why, for example, your executable program usually shows up in the same folder as the source file. Since you probably just loaded or saved the source file, the default prefix is the folder where the source file is located, and the compiler saves the executable program in the same place. The alphabet file from the last sample showed up in the same folder, too.

Finally, if you want to get at a file in a folder that is located in the default prefix, you can use a partial path name. For example, if the default prefix is the ORCA.PASCAL disk, and you want to access the BULLSYS.PAS file, you can use

```
SAMPLES/BULLSEYE.PAS
```

The process of forming names for the reset and rewrite command, then, is fairly simple. To get at a file in the default prefix, just use the file name. If the file is in a folder in the default prefix, give the name of the folder, followed by the file name, using a slash to separate the two. If you need to give the name of the disk, too, start off with a slash and the name of the disk, then add the folders and file names, again separated by slashes.

If this is new to you, the best thing to do is to practice. The easiest way to practice is with the CAT, EDIT and PREFIX commands, which you can use from the shell window. The PREFIX command sets the default prefix. To set the default prefix to /ORCA.PASCAL, for example, you would use

```
prefix /orca.pascal
```

The CAT command catalogs the current prefix, showing you what files and folders are there. Folders are marked with a file type of DIR in the second column.

Finally, the EDIT command is another way to open a source window. The command

```
edit /orca.pascal/samples/bullseye.pas
```

will open a new window and read in the file. Unlike the Open command from the File menu, though, the EDIT command does not change the current prefix.

Problem 9.2. Move to the shell window, and use the PREFIX command to set the current prefix to /ORCA.PASCAL/SAMPLES. Verify that you did it right by using this command to open the BULLSEYE.PAS program:

```
edit bullseye.pas
```

Move to the libraries folder using the prefix command. Use the CAT command to make sure you are in the right spot. If you are, you will see the files PASLIB and SYSLIB, and a folder called ORCAPASCLDEFS. Close the BULLSEYE.PAS window, and, without changing the default prefix, edit the file again.

Read and Write

Get and put can be very efficient in many programming situations, and you should definitely know they exist and how to use them. After all, the file variable is just a pointer, so you can use get to get a value from the file, then dereference the pointer as many times as you like to use the value. On the other hand, using get and put and dereferencing the file variable is a little clumsy for those cases where you just want to read and write values to a file, as we have done so far in our samples. They can also be very clumsy when you want to read several values at a time.

When Nicholas Wirth designed Pascal, he apparently agreed. The read and write procedures are the result. So far, we have used read for reading from the keyboard, and write for writing to the shell window. Actually, we can use these procedures with any file variable. When you don't give the procedures some other file to work on, read assumes that you want to read from the file variable input, and write assumes that you want to write to the file variable output.

The general form of these procedures looks like this:

```
read(file_variable, V1, V2, V3);
write(file_variable, V1, V2, V3);
```

where V1, V2 and V3 are variables that are type compatible with the file type. Of course, while our example shows three variables, you can read and write any number of variables with a single read or write call.

You might wonder why there are two seemingly different ways to read and write files. Actually, there aren't. The read and write procedures are just short cuts. The call

```
read(f, V1, V2, V3);
```

is absolutely, 100% equivalent to the following statements:

```
V1 := f^;
get(f);
V2 := f^;
get(f);
V3 := f^;
get(f);
```

The read procedure just saves you some typing. Similarly, the call to write

```
write(f, V1, V2, V3);
```

is just a shortcut for

```
f^ := V1;
put(f);
f^ := V2;
put(f);
f^ := V3;
put(f);
```

Problem 9.3. Earlier in this chapter there are two sample programs that read and write characters to a file called ALPHABET. Revise these samples to use read and write rather than get and put.

Finding the End of a File

In real programs, it's rare to actually know how many values are in a file before you open the file and look. We had the same problem with linked lists in the last lesson. Back then, we solved the problem by using a pointer value of nil to indicate that there was nothing else in the list. When a program reads a file, it uses a function called eof to find the end of a file. The eof function takes a file variable as a parameter, and returns a boolean variable. The value returned is true if the variable f^ is past the end of the file, and false if there are more values to be processed.

You can't reset a file if the file does not exist. If you try, the program will stop with a run-time error. On the other hand, it is perfectly legal for a file to exist, but not have anything in it. You can create a file with no values by doing a rewrite, but never using put or write to write values to the file. In the case of an empty file, eof(f) is true right after you call reset. Technically, you aren't supposed to use the value f^ in this case; if you try, you will get the same sort of thing as you would get from using a variable before you assign a value to it. In other words, some value is there, but there is no reliable way to figure out what it will be beforehand.

There is one more thing you need to know about files before all of this can be put to use. You can't do a get(f) if eof(f) is true. If you try, the program will stop with a run-time error.

If this all seems sort of complicated, it is. On the other hand, actually reading a file isn't hard at all. Putting all of these rules together, we will change our sample program from a few sections back to read the ALPHABET file without knowing in advance how many characters are in the file.

```
{ Read the alphabet file. }
program ReadIt(output);

var
   f: file of char;
   i: integer;

begin
reset(f, 'Alphabet');
while not eof(f) do begin
   write(f^);
   get(f);
   end; {while}
writeln;
end.
```

It is very important to understand exactly how this program works, so we are going to step through it line by line. It would take a while to step through the whole alphabet, so we will assume that the ALPHABET file has been shortened to the characters 'A', 'B' and 'C'.

Reset, of course, opens the ALPHABET file for input, and reads the first character. The file is not empty, so eof(f) is false, and we start looping. If the ALPHABET file existed, but didn't have any characters, eof(f) would have been true right away, and we would have skipped the whole loop. Graphically, the situation looks like this:

```
          character    value of eof
f^ -->    'A'          false
          'B'          false
          'C'          false
          undefined    true
```

In words, the picture says there are three characters in the file, and the file pointer currently points to the first one.

In the while loop, the first thing that happens is f^ is written to the shell window, so an 'A' appears. Next, the get procedure advances the file pointer, giving us this:

```
          character    value of eof
          'A'          false
f^ -->    'B'          false
          'C'          false
          undefined    true
```

We go to the top of the while loop, where eof(f) is still false, drop through, and write the character 'B'. The get call advances the file pointer again.

```
          character    value of eof
          'A'          false
          'B'          false
f^ -->    'C'          false
          undefined    true
```

Once again, eof is false, so we go through the loop, writing the character and making a call to get. The situation now is this:

```
          character    value of eof
          'A'          false
          'B'          false
          'C'          false
f^ -->    undefined    true
```

At this point eof(f) is true, so we drop out of the while loop. The contents of the file variable are undefined, and you aren't supposed to access the value, even to read it. In ORCA/Pascal, the value happens to still be 'C', but there is absolutely no guarantee that this would be true in another implementation of Pascal, or even in future updates to ORCA/Pascal. In fact, if you access f^ after leaving the loop, it's perfectly legal for the program to stop with a run-time error.

The eof function causes a lot of people a world of grief. This is because so many books give an incorrect description of the function. Unfortunately, the ORCA/Pascal reference manual is one of them. Many books leave you with the impression that eof(f) is true when f^ is 'C'; that is, it would be true after two calls to get in our sample. My favorite Pascal reference manual, Standard Pascal, by Doug Cooper, for example, says on page 128,

"eof(f) The function call eof(f) yields the value true if the file is empty beyond the component that f^ currently represents or if f is empty. It is an error to call eof(f) if f is undefined."

If you read on, though, the same book gives an example using a while loop, just like the one we used in our sample program, and says flatly that the program reads and processes all values in the file, and that it correctly handles an empty file. As a matter of fact, if eof(f) were true when f^ pointed to the last value in a file, it would be impossible to write a program that could correctly handle an empty file and a file with exactly one value, since eof would be true after the call to reset in both cases. I'm not sure who first gave an incorrect definition of eof, but book authors have been quoting each other ever since, perpetuating the inaccuracy.

Problem 9.4. At this point, you have the tools to merge two files. The basic method is simple: you open one file for input using reset, and another for output, using rewrite. You read values from one file, writing them to the other, until you get to the end of the first file. Next, you use reset to open the first file, then repeat the process of reading and writing values. The only difference is that you don't use rewrite to reopen the output file.

Write a program that writes the integers 1 to 10 to a file called FILE1.

Write a second program to create a second file, called FILE2, that contains the integers 11 to 20.

Write a third program, merge, to read FILE1, writing it to a file called FILE3. It should then read FILE2, adding the contents of FILE2 to FILE3. The program should not depend on knowing the length of either file.

Check your work with yet another program that reads the values in FILE3 and writes them to the shell window.

The process of combining two files, or adding information to an existing file, is a very common one. It is used by text editors to combine files, data

bases to add records, and adventure games to add new characters.

Text Files

Pascal has a special, predefined file type called text that is used to read and write files of characters. In a lot of respects, text is exactly like the type

```
type
   text = file of char; {close...}
```

In fact, in the original implementation of Pascal, text was a file of char. In modern implementations of Pascal, though, there are several operations that you can perform on a text file that you cannot perform on a file that is a file of char. Two of these are readln and writeln. That's because a file of char isn't necessarily organized as a series of lines. In fact, a file of char doesn't deal with the concept of a line at all.

You probably don't realize it yet, but you are also familiar with two text files already. Input and output, the file variables you put in the program heading, are both of type text.

We can put all of this to use right away to write a program that can read one of our Pascal programs. Call this program me.pas, so the reset command will have a file name to work on. Also, be sure and save the program to disk before you run it. Your program won't look for the file as a window on the desktop; it will only be able to read the file if it has been saved to disk.

```
{ Write this program! }

program WriteMe(output);

var
   f: text;
   l: string[80];

begin
reset(f, 'me.pas');
while not eof(f) do begin
   readln(f, l);
   writeln(l);
   end; {while}
end.
```

When you run this program, it will print the disk file into the shell window.

Problem 9.5. Some folks like uppercase, and some like lowercase. Let's assume that, for some reason, you want to convert the source code for me.pas to uppercase characters. Change the sample program so it reads a line, converts all of the characters to uppercase, and then writes the line to a new text file called me.upper. Since the new file is a text file, you can open it with the open command, just like you would open a program, to see if your program worked.

Eoln Detects the End of a Line

If you try to read a text file character by character, you will find something very strange. When you get to the end of the first line, the value of the character is ' '. Clearly a space can't be used to represent the end of a line; something strange must be happening. It is. Strange, that is.

You see, exactly how lines are divided up varies enormously from computer to computer. On the Apple IIGS, text files and program source files use chr(13) to represent the end of a line. On computers that are using the UNIX operating system, chr(10) signals the end of a line. Other computers may even use both, with chr(10) appearing right before or after chr(13)! In short, you can't even count on a computer using a single value to represent the end of a line, let alone using a particular value.

Of course, that presents some obvious problems if you want to write a program that will read a text file, but run on more than one computer. Pascal solves this problem with a function called eoln. Whenever the file variable gets to the end of a line, its value is set to a space. That way, you know what the value will be, no matter what computer you are running on. In addition, eoln(f) is true. That's how you tell that a space is really the end of a line, not just another space character.

There is also one safety valve in text files that saves you a lot of checking. Pascal guarantees that there is an end of line right before the end of file.

Putting these facts together, we can write a program to scan a text file character by character.

```
{ Read a file char-by-char }

program Echo(output);

var
   f: text;

begin
reset(f, 'me.pas');
while not eof(f) do begin
   if eoln(f) then begin
      get(f);
      writeln;
      end {if}
   else begin
      write(f^);
      get(f);
      end; {else}
   end; {while}
end.
```

The trick, of course, is to be sure you look for the special case when you get to the end of a line, and handle that case appropriately. In this program, when an end of line is found, writeln is used to write a new end of line. Writing the character wouldn't do, since the character in f^ is a space. Also, you have to remember to skip over the end of line mark. In this sample, I used get to do it, but you could also use read, and read a single character, or readln, which does gets until eoln is true, then does one more get.

Problem 9.6. Convert the sample program to use readln to skip the end of line, and read to read characters, rather than get.

Problem 9.7. Some people like to program in uppercase, some in lowercase, and some in mixed case. Write a program that asks for two file names. The program should open the first file for input, and the second file for output. Both files should be text files. Read characters from one file, convert them to uppercase, and write them to the new file.

Be sure and handle ends of lines correctly. If you do, you should be able to edit the output file using the PRIZM editor.

Problem 9.8. One way publishers measure the size of an article or book is by counting the number of words. Of course, they count them by hand, right? Well, you can do it better.

Write a program that asks for the name of a text file. Scan the file, counting the words. For our purposes, a word is defined as any sequence of characters that starts with an uppercase or lowercase character. It includes all of the characters up to the next character that is not an uppercase or lowercase character or a digit. For example, all of the following are words:

```
word     stuff V1
```

On the other hand, a number, like 9.6, is not a word.

As an added bonus, keep track of the lengths of the words.

After scanning the file, print the number of words, the number of characters (not including line feeds), the average length of a word, and a table showing how many times a word of each length appeared in the file. Lump any words longer than 30 characters together into a single element in the count array.

Be sure to use long integers for your character counters. After all, an integer can only hold values up to 32767. Each of these lessons has 30,000 to 40,000 characters, not counting the solutions to the problems.

Note: Be careful! You can't divide the character count by the number of words to get the average word length, because the character count includes spaces, commas, periods, and so forth! You must either compute the average from the word length array or keep a separate character counter for characters that appeared in a word.

Test your program by typing the following text into a file and saving it to disk.

```
How, now, brown cow.
single
i c a b
thisisaverylongwordtotesttoseeiflo
ngwordsarecaught
```

Leaving out the histogram entries where there were no entries, the results should be:

```
83 characters.
10 words.
4 lines.
The average word length is 7.4.

  length     number
  ------     ------
       1          4
       3          3
       5          1
       6          1
      30          1
```

One final note of caution about this problem. In terms of the complexity of the logic involved, this is the hardest problem so far in this course. It's worth spending some time on it to test and develop your skills. If you get tangled up, though, don't hesitate to scrap your program and try another approach. There are relatively easy ways to make this program work, and very hard ways. Don't get stuck struggling with a hard way.

If you do get stuck, and you are using read and readln to read the file, stop and think about how these commands are implemented in terms of get calls.

Reading the Keyboard

You now know that reset gets a file ready for input, and rewrite gets a file ready for output. In the programs you have written so far, though, you have used the text files input and output without ever using reset to get the file input ready, or rewrite to get the file output ready. Actually, you can reset(input) and rewrite(output). The reason you don't have to is that Pascal does it for you.

This brings up two very detailed points about files in Pascal. For the most part, you can just ignore them, but there are occasionally places where a little detailed knowledge about the language can help you out.

The first problem is that reset reads the first character of the file as part of getting a file ready for input. To get the first character of keyboard input, the program would have to stop before it executed a single line of code and wait until the person using the program typed a key, giving input^ an initial value. That would be annoying, to put it mildly. The solution is to leave the initial value undefined until the first read. That way the program doesn't stop to wait for a key to be pressed until you actually are ready for the user to type something.

The other thing you should be aware of is how ORCA/Pascal actually handles the keyboard. Instead of reading the keystrokes a character at a time, the program waits for the user to type a complete line, finishing the line by pressing RETURN. That's why you can edit lines as you type. The DELETE key, for example, deletes characters so you can make corrections. If the program were to read a character as soon as it was typed, you would have to create a very large, sophisticated procedure to read a line from the keyboard and still give the user a chance to correct mistakes.

You can also use eof to detect an end of file condition from the keyboard. Of course, that means you need some way to signal an end of file condition by pressing keys, since the keyboard is always there and "open for input." You can do this by holding down the control key and typing an @ character.

For the most part, though, you should do as you have done in all of your program, and let Pascal handle all of the details.

Formatted Output

As you probably have guessed, Pascal does all of its normal type checking when you use a file variable. For example, if f is a file of real, and i is an integer, you can't do this:

```
i := f^;
```

Of course, you also know that write(f, v) is just a shortcut for

```
v := f^;
get(f);
```

But if text is a file of characters, and output is a text file, then why does write(1) work?

The answer is that text is not exactly equivalent to file of char. You already saw that readln, writeln and eoln could only be used on text files, and not on files of char. Text files also get special treatment when it comes to the write and read procedures. Technically, when you use an integer value, real value, boolean value or string as a parameter to write or writeln, and the file is a text file, Pascal converts the value you pass to a series of characters and writes the values to the character file. Read and readln do just the opposite, converting a series of characters into an integer, real, or string.

This rule doesn't just apply to the standard files input and output, though. You can use read and write to format values for any text file, even a disk file.

Random Access

Let's say you have a file with five numbers, 1, 2, 3, 3, and 5. Of course, we want a file with a 4 in the fourth spot. On a short file like this one, we could just read the entire file into an array or linked list, make any changes we want, and write the modified file. If you know you have enough memory to work on the file that way, it's a good choice in any language.

Of course, in real life, we may not have enough memory to handle a file. It isn't uncommon to work with a mailing list with several thousand entries, for example. A reasonable sized record for handling the entries would be about 100 bytes long. A 10,000 person mailing list, then, would take 1,000,000 bytes, which is more free memory than you are likely to find on most Apple IIGS computers. In a situation like that, the only mechanism Standard Pascal offers for dealing with the file is to open two files, one for input and one for output. You then read each and every value from the old file, modify the ones you need to change, and write them to the new file.

Let's face it, if you are using a database, you might be willing to wait when you open a file, and wait again when you save the changed file. Asking you to wait while the file is read and written for each change is a bit much, though.

The obvious solution is to open the file for input and output at the same time. You then scan through the file until you find the value than has to be changed, or, if you already know where the value is, jump right to it. You then read the old value, change it, and write the modified value back to the file.

Unfortunately, Standard Pascal can't do that. Fortunately, most microcomputer Pascals can. In ORCA/Pascal, you open a file for input and output at the same time using open. It works just like reset and rewrite. You still give open a file variable, and an optional file name. Unlike rewrite, though, the old contents of the file are not destroyed.

To jump to a particular location in the file, you use the seek procedure. Seek needs two parameters, the file variable and a record number. The record number is the number of records to skip, so you would use

```
seek(f, 0)
```

to jump to the beginning of a file.

Putting these two techniques together, it is a simple matter to change a file. The following program resets the fourth element of a file called NUMBERS to 4, making the change discussed above.

```
program Update;

var
   f: file of integer;

begin
open(f, 'NUMBERS');
seek(f, 3);
write(f, 4);
end.
```

Problem 9.9. Test the update program by writing a program to create a file with the five integers

```
1  2  3  3  5
```

Create a second program that can read this file, printing the values to the shell window. Run the new program once to make sure the file has the values you wrote. Next, run the sample program to update the file. Finally, run the update program again to check to make sure the file has been changed properly.

Lesson Nine
Solutions to Problems

Solution to problem 9.1.

```
{ String I/O demo }
program StringIO(output);

var
   f: file of string[10];
   i: integer;

begin
{open the file for output}
rewrite(f);

{write the months to the file}
f^ := 'January';        put(f);
f^ := 'February';       put(f);
f^ := 'March';          put(f);
f^ := 'April';          put(f);
f^ := 'May';            put(f);
f^ := 'June';           put(f);
f^ := 'July';           put(f);
f^ := 'August';         put(f);
f^ := 'September';      put(f);
f^ := 'October';        put(f);
f^ := 'November';       put(f);
f^ := 'December';       put(f);

{open the same file for input}
reset(f);

{read and echo the values}
for i := 1 to 12 do begin
   writeln(f^);
   get(f);
   end; {for}
end.
```

Solution to problem 9.2.

The commands needed are:

```
prefix /orca.pascal/samples
edit bullseye.pas
prefix /orca.pascal/libraries
cat
edit /orca.pascal/samples/bullseye.pas
```

Solution to problem 9.3.

```
{ Write the alphabet to a file. }
program WriteIt;

var
   f: file of char;
   ch: char;

begin
rewrite(f, 'Alphabet');
for ch := 'A' to 'Z' do
   write(f, ch);
end.


{ Read the alphabet file. }
program ReadIt(output);

var
   ch: char;
   f: file of char;
   i: integer;

begin
reset(f, 'Alphabet');
for i := 1 to 26 do begin
   read(f, ch);
   write(ch);
   end; {for}
writeln;
end.
```

Solution to problem 9.4.

```
{ Create File1 }
program MakeFile(output);

var
   f: file of integer;
   i: integer;

begin
{open the file for output}
rewrite(f, 'file1');

{write the file values}
for i := 1 to 10 do
   write(f, i);
end.



{ Create File2 }
program MakeFile(output);

var
   f: file of integer;
   i: integer;

begin
{open the file for output}
rewrite(f, 'file2');

{write the file values}
for i := 11 to 20 do
   write(f, i);
end.
```

```
{ Create File3 }
program Merge(output);

type
   str5 = string[5];

var
   outfile: file of integer;


   procedure Merge (name: str5);

   { Add the file to outfile                    }
   {                                            }
   { Parameters:                                }
   {    name - name of the file to add          }
   {                                            }
   { Variables:                                 }
   {    outfile - output file variable          }

   var
      i: integer;               {file value}
      infile: file of integer;  {file}

   begin {Merge}
   reset(infile, name);
   while not eof(infile) do begin
      read(infile, i);
      write(outfile, i);
      end; {while}
   end; {Merge}


begin
{open the file for output}
rewrite(outfile, 'file3');

{merge the two files}
Merge('file1');
Merge('file2');
end.
```

```
{ Read file3 }
program ReadIt(output);

var
   f: file of integer;
   i: integer;

begin
reset(f, 'file3');
while not eof(f) do begin
   read(f, i);
   writeln(i);
   end; {for}
end.
```

Solution to problem 9.5.

```
{ Write this program! }

program Upper(output);

var
   f: text;                    {file variable}
   i: integer;                 {loop variable}
   l: string[80];              {string from the file}


   function ToUpper(ch: char): char;

   { Convert a character to uppercase                }
   {                                                 }
   { Parameters:                                     }
   {    ch - character to convert                    }
   {                                                 }
   { Returns: Uppercase equivalent of ch             }

   begin {ToUpper}
   if (ch >= 'a') and (ch <= 'z') then
      ch := chr(ord(ch)-ord('a')+ord('A'));
   ToUpper := ch;
   end; {ToUpper}


begin
reset(f, 'me.pas');
while not eof(f) do begin
   readln(f, l);
   for i := 1 to length(l) do
      l[i] := ToUpper(l[i]);
   writeln(l);
   end; {while}
end.
```

Solution to problem 9.6.

```
{ Read a file char-by-char }

program Echo(output);

var
   f: text;
   ch: char;

begin
reset(f, 'me.pas');
while not eof(f) do begin
   if eoln(f) then begin
      readln(f);
      writeln;
      end {if}
   else begin
      read(f, ch);
      write(ch);
      end; {else}
   end; {while}
end.
```

Solution to problem 9.7.

```
{ Upper                                                  }
{                                                        }
{ This program converts a program to uppercase           }
{ characters.  You are prompted for the input and        }
{ output files.                                          }

program Upper(input,output);

var
   ch: char;                    {character from the file}
   inFile, outFile: text;       {input and output files}
   name: string[80];            {file name}
```

```
   function ToUpper(ch: char): char;

   { Convert a character to uppercase                  }
   {                                                   }
   { Parameters:                                       }
   {    ch - character to convert                      }
   {                                                   }
   { Returns: Uppercase equivalent of ch               }

   begin {ToUpper}
   if (ch >= 'a') and (ch <= 'z') then
      ch := chr(ord(ch)-ord('a')+ord('A'));
   ToUpper := ch;
   end; {ToUpper}


begin
write('Input file name:');      {open the input file}
readln(name);
reset(inFile, name);
write('Output file name:');     {open the output file}
readln(name);
rewrite(outFile, name);

while not eof(inFile) do begin  {translate the file}
   if eoln(inFile) then begin
      readln(inFile);
      writeln(outFile);
      end {if}
   else begin
      read(inFile, ch);
      write(outFile, ToUpper(ch));
      end; {else}
   end; {while}
end.
```

Solution to problem 9.8.

```
{ Count                                                    }
{                                                          }
{ This program counts the characters, lines, and words     }
{ in a file.  It also prints the average word length       }
{ and a histogram of word lengths.                         }

program Count(input,output);

const
   maxLen = 30;                {max length of a word}

var
   ch: char;                   {character from the file}
   f: text;                    {file to count}
   name: string[80];           {file name}

   i: integer;                 {loop variable}
   length: integer;            {length of a word}

   chars: longint;             {characters in the file}
   inWord: boolean;            {are we in a word?}
   lines: longint;             {lines in the file}
   wordChars: longint;         {characters that appear in words}
   wordLen: array[1..30] of longint; {word length histogram}
   words: longint;             {words in the file}


   function IsAlpha(ch: char): boolean;

   { Is the character alphabetic?                          }
   {                                                       }
   { Parameters:                                           }
   {    ch - character to test                             }
   {                                                       }
   { Returns: True if the character is alphabetic, else    }
   {    false.                                             }

   begin {IsAlpha}
   IsAlpha := ((ch >= 'a') and (ch <= 'z'))
      or ((ch >= 'A') and (ch <= 'Z'));
   end; {IsAlpha}
```

```
   function IsAlphaNum(ch: char): boolean;

   { Is the character alphabetic or numeric?           }
   {                                                   }
   { Parameters:                                       }
   {    ch - character to test                         }
   {                                                   }
   { Returns: True if the character is alphabetic or   }
   {    numeric, else false.                           }

   begin {IsAlphaNum}
   IsAlphaNum := ((ch >= '0') and (ch <= '9')) or IsAlpha(ch);
   end; {IsAlphaNum}


   procedure WordStats;

   { Add in the statistics for a new word.             }
   {                                                   }
   { Variables:                                        }
   {    length - length of the word                    }
   {    wordChars - number of characters in words      }
   {    wordLen - word length histogram                }

   begin {WordStats}
   wordChars :=                {update the word character count}
      wordChars+length;
   if length > maxLen then     {make sure we don't exceed the max length}
      length := maxLen;
   wordLen[length] :=          {update the histogram}
      wordLen[length]+1;
   end; {WordStats}


begin
write('File to count:');       {open the input file}
readln(name);
reset(f, name);

words := 0;                    {no words so far}
chars := 0;                    {no characters so far}
lines := 0;                    {no lines so far}
for i := 1 to maxLen do        {no words in the length array}
   wordLen[i] := 0;
wordChars := 0;                {no characters in a word, yet}
inWord := false;               {not doing a word}
```

```
while not eof(f) do begin       {process the file}
   if eoln(f) then begin
      readln(f);
      lines := lines+1;         {count an end of line}
      if inWord then begin
         WordStats;             {total the word statistics}
         inWord := false;       {not in a word}
         end; {if}
      end {if}
   else begin
      read(f, ch);              {get the character}
      chars := chars+1;         {count the character}
      if inWord then begin      {continue stats if in a word}
         if IsAlphaNum(ch) then
            length := length+1
         else begin
            inWord := false;
            WordStats;
            end; {else}
         end {if}
      else if IsAlpha(ch) then begin
         words := words+1;      {count the new word}
         inWord := true;        {note that we are in a word}
         length := 1;           {scan the characters in the word}
         end; {else if}
      end; {else}
   end; {while}

writeln(chars:1, ' characters.'); {write the statistics}
writeln(words:1, ' words.');
writeln(lines:1, ' lines.');
writeln('The average word length is ', wordChars/words:1:1, '.');
writeln;
writeln('  length     number');
writeln('  ------     ------');
for i := 1 to maxLen do
   writeln(i:8, wordLen[i]:11);
end.
```

Solution to problem 9.9.

```
{ Create the file to update }

program Create;

var
   f: file of integer;

begin
rewrite(f, 'NUMBERS');
write(f, 1);
write(f, 2);
write(f, 3);
write(f, 3);
write(f, 5);
end.



{ Write the NUMBERS file }

program WriteNumbers(output);

var
   f: file of integer;
   i: integer;

begin
reset(f, 'NUMBERS');
while not eof(f) do begin
   read(f, i);
   writeln(i);
   end; {while}
end.
```

Lesson Ten
Miscellaneous Useful Stuff

A Look at this Chapter

The first nine lessons of this course have taken you on a tour of the Pascal language. By this time, you have learned most of the mechanics of the language itself. Because the lessons have been developed using specific examples, though, a few topics have slipped through the cracks. This chapter covers those topics.

I don't want you to get the impression that these topics are unimportant. Quite the contrary: a great deal of the power of the Pascal language is tied up in the topics we will look at in this lesson. In particular, sets, variant records, and with statements are things you will see a lot of in the rest of the course. In our tour of the Pascal language, though, we have concentrated on the mechanics of writing short, simple programs. As we learn more about writing larger programs, programming efficiently, and organizing programs, the new techniques covered in this lesson will be put to use over and over.

Subranges

There are a lot of times when you are dealing with a variable when you know that the value will always lie in a certain range. Well, you expect the value to lie in a certain range, anyway. If it doesn't, your program isn't working right. In cases like these, Pascal has a way for you to signal the compiler, telling it what the possible values are. Pascal can actually check for you, making sure the value never gets out of the range you define! This is a very powerful tool for debugging your programs, and represents one of the ways that Pascal provides a safety net for programmers.

Lets look at a simple example to see how this would work. In many of our graphics programs, we have used a variable to hold the pen color. We know that pen colors must be 0, 1, 2 or 3. We also used x and y values, expecting them to stay in the ranges 0 to maxX or maxY. Instead of defining the variables as integers, we could specify the valid values, like this:

```
var
   color: 0..3;
   x: 0..maxX;
   y: 0..maxY;
```

There are several advantages to doing this. The first, and most obvious, is that the subrange serves as a kind of comment, reminding us what values the variable should hold. This is useful enough by itself, but there is an even more important advantage: if we ask it to, the compiler can actually check to make sure that the variables never have a value outside of the range we specify. If a bug in the program sets the variable to an unacceptable value, the program will stop with an error.

To see how this works, let's write a short program that violates the rule.

```
{$rangecheck+}
{$names+}

program goof;

var
   color: 0..3;

begin
color := 2;
color := color*4 mod 4;
color := color+6;
end.
```

The first directive, {$rangecheck+}, tells the compiler to check to make sure that values lie within the acceptable subranges. This takes time and space in your program, so the compiler only does it if you ask it to. The next directive, {$names+}, tells the compiler to print a trace back in the shell window if the program fails. The trace back shows you where the program fails, instead of simply telling you that it failed.

Tracing through the program, the first statement assigns 2 to color. That's OK; 2 is a legal value for color. The second statement multiplies this by 4, giving 8. Eight is not a legal value for color, but before the assignment takes place, we do a mod 4. That changes the value to 0. Even though the intermediate value was outside of the range 0..3, the compiler does not flag an error. You only get an error if you actually assign a value to the variable. That happens on the next line, where we set color to 0+6. That is where the program fails.

You may have noticed that subranges look a lot like an array subscript. They don't just look alike, they are identical. For example,

```
var
   a: array[1..10] of real;
```

and

```
type
   ten = 1..10;

var
   a: array[ten] of real
```

do exactly the same thing. In fact, we have already used the rangecheck directive, too. We used it in several examples to make sure that the subscript for an array was legal. What you were doing, without realizing it, was creating a subrange and checking to make sure that the array index was inside the allowed subrange.

Subranges can be of any scalar type. To review, the scalar types are integer, char, boolean, and enumerations. Scalar types also include subranges themselves.

When you use a subrange to specify the range of values for a variable, it does not create a new type; the variable is treated as if it were of exactly the same type as if no subrange was used. In other words, the only effect of a subrange is to document the program and allow range checking to be used. In our sample program, color is still type compatible with integer, since 0..3 is a subrange of integers.

Problem 10.1. The old Buck Rogers Magic Decoder Ring, and other such devices, use a simple cipher to encode text messages. A number is assigned to each character. The characters in the message are then encoded as numbers. The person receiving the message can decode it easily if the starting number is known, but has to do a bit of work to decode the message without the starting number.

For example, here is the code table for a starting number of 6.

```
 6  7  8  9 10 11 12 13 14 15 16
 A  B  C  D  E  F  G  H  I  J  K

17 18 19 20 21 22 23 24 25 26  1
 L  M  N  O  P  Q  R  S  T  U  V

 2  3  4  5
 W  X  Y  Z
```

To encode the word PASCAL, you just copy the numbers corresponding to each character, like this:

```
21 6 24 8 6 17
```

Write a program that decodes these cyphers. It should start by asking for the starting number. Using this starting number, fill an array of 26 characters with the correct characters. The array should be declared like this:

```
code: array[values] of alpha;
```

Declare appropriate subranges for values and alpha.

After creating the code array, the program should read integers from the keyboard until a zero is found. After reading each integer, it should with the character that corresponds to the integer. If you read a 27, print a space, without consulting the code array.

Turn range checking on to make sure you don't exceed any subranges.

Use your program with a starting value of 10 to decode this message:

```
10 1 14 27 6 14 27
17 10 5 18 23 16 27
15 4 23 27 8 14 3
```

Sets

When you look at the history of programming languages, a few languages stand out as major trend setters. In my opinion, FORTRAN was the first of these. FORTRAN was the first language to successfully deliver enough power and simplicity so that engineers and scientists could make effective use of a computer. That doesn't mean engineers and scientists can't learn assembly language, of course. It just recognizes that they have other things to do with their time. They needed a language that could deliver the power to evaluate expressions, but was still easy to learn and use. COBOL makes the list, too; it did the same thing for the business community. ALGOL, though, was the first commonly available language to offer a unified, structured approach to programming. In the late 60s and early 70s, an enormous number of the

books written by computer scientists used ALGOL to demonstrate concepts of programming. Pascal picked up where ALGOL left off, learning from mistakes made in that language. Pascal offered more organization and safety than any other language that preceded it. While other languages have since equaled Pascal, none have significantly improved on the safety of the language, and few have offered more organizational features.

Of the many advances that Pascal offered, several stand out. It pulled the concept of records and types together in a synthesis that had an enormous impact on language design. Today, so many languages do the same thing that it is easy to forget that Pascal blazed trails in this area. It also offered sets, a data type reminiscent of artificial intelligence languages like LISP. Sets are actually one of the most useful features in the Pascal language, but they are often overlooked, since records are so much more flashy. It takes a while to learn to use records well, and by the time most people do, they are beyond the beginner books that also told them about the marvels of sets. Sets are also hampered because many Pascal compilers don't handle them efficiently. ORCA/Pascal handles them quite well, though, so you shouldn't let efficiency be an issue in your Apple IIGS programs.

As with all of the other topics we have looked at, we'll look at sets by examining a lot of examples. You will learn what sets are gradually.

One way to think of the set is as a collection of little flags, each of which can be true or false. The flag is true if the element of the set the flag is tied to is in the set, and false if it is not.

Let's use a small set to look at this idea in more detail. We'll use a set of planets, defined by the enumeration

```
planet = (Mercury, Venus, Earth,
          Mars, Jupiter, Saturn,
          Uranus, Neptune, Pluto);
```

To make testing the set a bit easier, we will also define a few set variables. Set variables look a lot like file variables, in the sense that the type is "set of something," just like the type of a file is "file of something." In the case of sets, the "something" part has to be an ordinal type. To review, the ordinal types are char, integer, boolean, any enumeration, and any subrange of any of these types. So, for defining sets of planets, we would use a type like this one:

```
planetSet = set of planet;
```

Set variables are defined in the same way. Let's define a few to build a simple database that we can then use to test for various conditions.

```
gasGiant: planetSet;
rocky: planetSet;
hasMoons: planetSet;
visited: planetSet;
landed: planetSet;
hasAtmosphere: planetSet;
all: planetSet;
```

The first thing, of course, is to give some of these variables a value. To specify the values in a set, you inclose the members you want to be in the set between square brackets, like this:

```
landed := [Venus, Earth, Mars];
```

Listing the elements individually isn't the only way to specify the members of a set. There are a number of tricks and shortcuts that can be used. The first is to use a subrange, where you specify the first and last element, separated by a .. sequence. For example, this assignment does exactly the same thing as the last assignment:

```
landed := [Venus..Mars];
```

You can also mix subranges with individual elements, separating them with commas. The rocky planets gives us an opportunity to do that, since all of the planets from Mercury to Mars are considered rocky, but Pluto is also a rocky planet.

```
rocky := [Mercury..Mars, Pluto];
```

With these ideas in mind, we can prepare the database for our trip.

```
all := [Mercury..Pluto];
gasGiant := [Jupiter..Neptune];
moons := [Earth..Pluto];
atmosphere := [Venus..Neptune];
visited := [Venus..Neptune];
landed := [Venus..Mars];
```

Hopefully, you won't quibble too much. Pluto seems to have some sort of thin atmosphere, but not much. By visited, I mean the planets that have had at least a close flyby by a space probe, like Voyager. I also counted Earth in both the visited and landed category.

And, of course, I didn't give rocky a value. Looking at the list, we have one set, all, that includes all of the planets. Another, gasGiant, includes all of the huge planets that are made up mostly of gas. The rocky planets are the ones left over. To get them, we can use our first set operator. It's one you have already seen used on other types.

```
rocky := all - gasGiant;
```

The - operator, when applied to a set, removes all of the things in the second set from the first. It is OK if the first set doesn't have all of the things in the second set. For example, it would have been alright to subtract the gas giants from a set that didn't have Jupiter. The purpose is to eliminate anything that is there, not to complain about something that is missing. The table below shows what happens.

all	gasGiant	all - gasGiant
Mercury		Mercury
Venus		Venus
Earth		Earth
Mars		Mars
Jupiter	Jupiter	
Saturn	Saturn	
Uranus	Uranus	
Neptune	Neptune	
Pluto		Pluto

Let's say you want to plan a simple vacation. The first try might be to look for planets that have an atmosphere or a moon, since both give you something to look at once you get there. What you want, then, are all of the planets that are in hasAtmosphere and hasMoons. The + operator does this, returning a set that has all of the elements that were in either of the input sets. Assigning the result to vacationSpots, we have

```
vacationSpots :=
   hasAtmosphere + hasMoons;
```

Using our table, you can see how this works. It is a lot like the boolean or operator, where each element of the final set is in the set if it was in either one of the starting sets.

atmosphere	moons	vacationSpots
Venus		Venus
Earth	Earth	Earth
Mars	Mars	Mars
Jupiter	Jupiter	Jupiter
Saturn	Saturn	Saturn
Uranus	Uranus	Uranus
Neptune	Neptune	Neptune
	Pluto	Pluto

Well, that didn't narrow it down, much. Since there are so many lush spots to choose from, we can be a bit more picky. Instead of asking for places that have an atmosphere or a moon, we'll ask for places that have both. This is the intersection of the sets, or a set that is made up of members that appear in both sets. The * operator gives us the result we need, trimming our table down a little. It works like an and operation on boolean values.

```
vacationSpots := atmosphere * moons;
```

atmosphere	moons	vacationSpots
Venus		
Earth	Earth	Earth
Mars	Mars	Mars
Jupiter	Jupiter	Jupiter
Saturn	Saturn	Saturn
Uranus	Uranus	Uranus
Neptune	Neptune	Neptune
	Pluto	

To some extent, you can compare sets, too. The = and <> operators ask if two sets are identical. For two sets to be identical, they both have to have exactly the same members. You can compare to the null set, [], for example, to see if there are any members at all. The <= and >= operators can also be used with sets. This is a little trickier to understand. What the comparison operator does is to see if all of the elements in one set are in another. For example,

```
gasGiant <= atmosphere
```

asks if all of the members of the set gasGiant are also in the set atmosphere. Since all of the gas giants do, in fact, have an atmosphere, the result is true. On the other hand,

```
gasGiant >= atmosphere
```

asks if all of the elements of the set atmosphere are in the set gasGiant. In terms of our database, we are asking if all of the planets with an atmosphere are also gas giants. Venus, Earth and Mars are all in the set atmosphere, but do not appear in the set gasGiant, so the result is false.

With some of the basic ideas behind us, it's time to start looking at some programming examples. For our first example, let's take another look at a subroutine you have seen many times, ToUpper. This subroutine takes a character and returns its uppercase equivalent.

```
function ToUpper(ch: char): char;

{ Convert a character to        }
{ uppercase                     }
{                               }
{ Parameters:                   }
{    ch - character to convert  }
{                               }
{ Returns:                      }
{    Uppercase equivalent of ch }

begin {ToUpper}
if (ch >= 'a') and (ch <= 'z')
   then
   ch := chr(ord(ch) -
      ord('a') + ord('A'));
ToUpper := ch;
end; {ToUpper}
```

The problem that must be solved in this subroutine is a simple one that comes up over and over in programming. To convert from lowercase to uppercase, we need to subtract

```
ord('a') - ord('A')
```

from the character value. Of course, if we do that to, say, the '{' character, which falls right after 'z' in the ASCII character set, we would convert the character to an '[' character. Obviously, we need a way to make sure that the operation is only done on lowercase characters. The if statement does the job, but sets can be used to express the idea a little better.

```
function ToUpper(ch: char): char;

{ Convert a character to        }
{ uppercase                     }
{                               }
{ Parameters:                   }
{    ch - character to convert  }
{                               }
{ Returns:                      }
{    Uppercase equivalent of ch }

begin {ToUpper}
if ch in ['a'..'z'] then
   ch := chr(ord(ch) -
      ord('a') + ord('A'));
ToUpper := ch;
end; {ToUpper}
```

The function doesn't look that much different. The only thing that has changed is the if statement, which uses the condition

```
ch in ['a'..'z']
```

A lot of the power and usefulness of sets are tied up in that simple test. It asks the question, "Is the value of the character variable ch the same as one of the values from 'a' to 'z'?" In this case, 'a'..'z' represents a subrange of characters, so, for example, 'm' is in the range of characters. If ch is in the set of characters listed, the result of the expression is true. If it is not, the result is false.

So far, all of the set values we have created are made up of constants or subranges. You can also use variables, functions, or even general expressions to fill a set. One common example of this is when you want to add an element to an existing set, but the value is in a variable. One simple example that shows how to do this is a program that finds all of the characters that appear in a file. We can represent all of the members of the type char as a set, using the declaration

```
type
   charSet = set of char;
```

We need a set to hold the characters we find, of course. We'll call that set found.

```
var
   found: charSet;
```

When we start, we haven't found any characters at all, so we set found to the null set. The null set is a set with nothing in it. It's sort of like zero for numbers, or nil for a pointer variable.

```
found := [];
```

We will read characters into the variable ch. Ch is a variable, not a set, but we still want to add the character to the members of the set. The + operator can add two sets, and since we can use a variable in a set value, we can add ch to the set of found characters like this:

```
found := found + [ch];
```

Putting these ideas together, we get the program in listing 10.1. It reads a file, builds a set of the characters in the file, and then prints the characters.

Back when we first looked at the case statement, I pointed out that it is not legal to supply a value as the case value that was not handled. For example, this case statement handles numbers from 1 to 10:

```
case i of
   1,3,5,7,9: write('odd');
   2,4,6,8,10: write('even');
   end; {case}
```

It is illegal to execute this code if i has a value of 20, since there is no case label to handle the situation. In this case, an if statement can be used to get around the restriction, like this:

```
if (i >= 1) and (i <= 10) then
   case i of
      1,3,5,7,9: write('odd');
      2,4,6,8,10: write('even');
      end; {case}
```

While it worked in this particular situation, testing to see if the value is valid can get very cumbersome in a real hurry. What if we are writing a program that handles the vowels (a, e, i, o, u) and the letters that are

Listing 10.1

```
{ Find all of the characters that appear in a file }

program FindChars(input, output);

type
   charSet = set of char;

var
   found: charSet;                  {set of chars found}
   f: text;                         {file to read from}


   procedure OpenFile;

   { Open the file f. }

   var
      fname: string[64];            {name of the file}

   begin {OpenFile}
   write('File to scan:');
   readln(fname);
   reset(f,fname);
   end; {OpenFile}
```

(continued)

"sometimes vowels," as my second grade teacher always put it (y, w). It would be inconvenient, to say the least, to test for each letter before entering the case statement. ORCA/Pascal has an extension, the otherwise label, to handle this situation, and this is what you were shown in Lesson 7. This is a very common extension, but you may have wondered why Nicholas Wirth didn't put it into the original Pascal language. The reason is sets: with sets, the if check becomes easy again.

```
if ch in
   ['a','e','i','o','u','y','w']
   then
   case ch of
      'a','e','i','o','u':
         Vowel(ch);
      'y','w':
         MaybeVowel(ch);
      end; {case}
```

You will find this sort of check peppered

(continuation of Listing 10.0)

```
      procedure FindChars;

      { Create a set containing all of the characters from }
      { the file f.                                        }

      var
         ch: char;                     {char from the file}

      begin {FindChars}
      found := [];                     {nothing in the set}
      while not eof(f) do begin        {for each char in the file...}
         read(f,ch);
         found := found + [ch];        {...add it to the set}
         end; {while}
      end; {FindChars}


      procedure PrintChars;

      { Print the printable characters from the set found }

      var
         ch: char;                     {loop variable}

      begin {PrintChars}
      for ch := ' ' to '~' do
         if ch in found then
            write(ch);
      end; {PrintChars}


   begin
   OpenFile;                           {open the file}
   FindChars;                          {find the chars}
   PrintChars;                         {print the results}
   end.
```

throughout the code of an experienced Pascal programmer.

Throughout all of this, I have ignored one interesting point. How big is a set? They must be fairly big, since you can create a set of char, and there are 128 characters in the ASCII character set used on the Apple IIGS. All Pascal compilers have some upper limit on the size of a set. The original Pascal standard did not give any guidance in this area, and many old compilers limit the number of elements that can be in a set to a fairly small number. Because sets are so useful for characters, though, the ISO Pascal standard (the international standard) and the ANSI Pascal standard (the American standard) both require sets to be big enough to create a set of char. In ORCA/Pascal, sets can be considerably larger. In fact, they can have up to 2048 elements.

That's quite a few elements, and it is very natural to worry about how much memory these sets take up. Fortunately, sets are variable size. When you specify the type of a set, the compiler knows how many elements there could be. In the case of a set of char, the compiler knows that there can never be more than 128 elements in the set, so it only reserves enough space to hold 128 elements. Sets are also stored very efficiently; only one bit us used for each set element. Since there are eight bits in a byte, a set of char requires 16 bytes – twice the memory of a double precision floating point number, and only 1/16th of the memory that would be required to create a boolean array with one element for each character. In short, sets are a very efficient way to store information.

Problem 10.2. As you have no doubt gathered, hexadecimal numbers are often used when dealing with computers. Converting from one base to the other is pretty easy, but a bit tedious. You are probably aware of that by now, since you had to do it for two different programs. In this problem, you will make life easier for yourself by writing a program to convert these hexadecimal numbers to decimal numbers.

The idea is really very simple. Start by setting a value to zero; the value should be declared as a long integer, which will let you handle hexadecimal numbers up to 8 digits long. Next, read a string from the keyboard. Loop over each character in the string. For each character, do the following:

1. Make sure the character is a digit ('0'..'9') or a hexadecimal character ('A'..'F' or 'a'..'f'). Use sets for this check. If the character is not valid, flag an error.

2. Convert the character to a value. To do this, call a function that accepts a hexadecimal character as input, and returns its integer value. This subroutine is broken down into these steps:

    1. Convert the character to uppercase to make the rest of the program easier. Use ToUpper, from the text of this lesson (the version that used sets).

    2. Form an integer value by subtracting ord('0') from the ordinal value of the character.

    3. If the result is larger than 9, subtract 7.

    4. Return the result.

3. Multiply the old value by 16, then add the value returned by the function described in step 2.

Naturally, the program should prompt the user for a number and write the result. You may also want to allow more than one number to be converted. I will leave the details up to you – by now, you have written enough programs like this one to start forming opinions of your own about how you want the program to work!

Revisiting the For Loop

Once upon a time, in a lesson long, long ago, you learned about the for loop. Since then, we have expanded on your knowledge of this loop, so that you now know that you can loop over any ordinal type. When for loops were first introduced, though, you didn't know enough about Pascal to understand some of the features and restrictions that apply to for loops. In this section, we will take a more detailed look at for loops to fill in some minor gaps in your knowledge.

Pascal's for loop is specifically designed for situations when you want to loop over a range of ordinal values. The biggest implication of restricting the for loop to ordinal values is that you can't use real numbers as a for loop index. The for loop is also designed to loop over all of the values, sequentially, without skipping any. These may seem like harsh restrictions, but Pascal has the while and repeat loops,

too, and these can be used to fill in the gaps left by these restrictions.

In languages older than Pascal, many programming bugs occurred because a programmer changed the value of the for loop variable, either accidentally or on purpose, and the compiler generated code that could not handle the new value. For example, if you tell the compiler that you want to loop from 1 to 100, like this:

```
for i := 1 to 100 do
```

but then put the assignment

```
i := 200;
```

in the loop, problems can result. Some compilers might generate code that assumed i would always be less than 101, causing a crash or infinite loop. Other compilers might be careful enough not to assume a particular value of i, but the resulting for loop would take longer to execute, slowing down the program.

To prevent this kind of problem, and to allow the compiler to generate the best possible code, Pascal has several restrictions on the for loop variable. Basically, they all boil down to one simple idea: you cannot change the for loop variable inside of the loop. The specific restrictions are:

1. You cannot assign a value to the for loop variable inside the for loop. You can assign a value to the for loop variable if the assignment is not in the for loop, though.
2. You cannot use the same loop variable for two nested for loops. Two loops that appear sequentially can use the same for loop variable, but if one loop appears inside the other, they must each have their own loop variable.
3. The for loop variable cannot be passed as a var parameter to a function or procedure, since the function or procedure can change a var parameter. You can pass a for loop variable as a value parameter, though, since the function or procedure cannot change the local copy of the variable.
4. The for loop variable must be defined locally, in the same function, procedure or program where the for loop appears.

These restrictions make for loops in Pascal very efficient compared to other languages. In fact, for loops in ORCA/Pascal are often more efficient than for loops in ORCA/C that do the same thing, even though both compilers use the same code generation techniques. The difference is that C doesn't have these restrictions, so the compiler has to generate more complicated machine code to do the loop tests.

Once a for loop finishes, you might be tempted to use the value. That's a bad idea. After this loop, for example, what is the value of i?

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

The honest answer is that you don't know. That's a good answer to remember, because a Pascal compiler isn't required to end the loop with i set to any particular value. The value of a loop variable, technically, is undefined after a for loop finishes. Any program that depends on the value can fail if you move it to another Pascal compiler that happens to leave a different value in the loop variable. In fact, it might fail if you compile the program under two different versions of the same compiler! The lesson to learn is simple, though: don't use the for loop variable after leaving a for loop unless you assign a value to it first.

That's enough restrictions! There are also some capabilities the for loop has that we haven't talked about in detail, too. The first you have seen by example, but it is a good idea to spell it out. You can use any valid Pascal expression to decide what the start and stop value for the loop should be. For example, you can loop a random number of times using the results of the random number function we have used in so many simulations:

```
for i := 1 to Random do ;
```

This brings up a couple of interesting points, though. First, what if the value returned is less than 1? The for loop can handle that. If the stop value is smaller than the start value, the body of the loop never gets executed. A clearer example may help you to understand this point.

```
for i := 10 to 1 do
   writeln(i);
```

What will this loop write? The answer is, "nothing." Since the starting value of i is already larger than 1, the writeln statement never gets executed. The same check prevents problems if Random returns a value smaller than 1 in our original example.

You might be justifiably concerned about another issue, too. What would happen if Random were called every time the condition were tested? The answer, of course, is that the stop value would change each time through the loop! Pascal evaluates the stop condition one time, though, and saves the value. Even if the stop

condition doesn't change, you might be worried about the efficiency of your program. The fact that Pascal computes the stop value before the loop starts, and saves the value, means that even a very complex expression for the stop value won't slow down the loop itself.

So far, all of our for loops have started with a small value and looped up towards a larger one. That isn't the only way to loop. You can start with a large value, and loop down to a smaller one. The difference is that you use downto instead of to, like this:

```
for i := 10 downto 1 do
   writeln(i:1, '...');
writeln('Blast-off!');
```

Problem 10.3. One way to reverse a sequence of characters is to loop backwards, starting at the last character in the string, and looping towards the first. Write a program that uses this idea to reverse the characters in a string.

Your program should prompt for a string. Next, print the string in reverse order, using downto and looping from length(str) down to 1.

Continue processing strings until the user enters a null string (one with a length of 0).

Problem 10.4. Which of the following code fragments are legal in Pascal? In each case, assume that the loop variables are declared locally, and are integers.

```
a. for i := 1 to 10 do
      ;

b. for i := 10 to 1 do
      ;

c. for i := 10 downto 1 do
      ;

d. for i := 1 to 10 do
      for j := 1 to 10 do
         ;

e. for i := 1 to 10 do
      for i := 1 to 10 do
         ;

f. for i := 1 to 10 do
      i := 4;

g. for i := 1 to 10 do
      ;
   i := 4;

h. for i := 1 to 10 do
      ;
   writeln(i);

i. for i := 1 to 10 do
      ;
   for i := i to 10 do
      ;
```

Gotos Are Legal In Pascal, Too!

Goto statements are the last kind of statement involved in the flow of control in Pascal. Basically, a goto statement is a jump. The program moves to the destination of the goto, and starts executing with that statement. The following program gives a very simple example of this idea.

```
program DoAGoto(output);

label 3;

begin
goto 3;
writeln('This gets skipped.');
3: writeln('This gets printed.');
end.
```

As you can see, there isn't much to a goto statement. In fact, all there is is the reserved word goto, followed by a number called a label. The number tells the compiler where to go to; a corresponding number must appear somewhere in the program, followed by a colon. The number also must be declared as a label in a label area, as shown. If you need more than one label, they are separated by commas, like this:

```
label 1,99,102;
```

The label declaration area comes before anything else in the declaration area. Constants come after it, followed by types, variables, and finally procedures or functions.

Give the program a try, especially if you aren't sure exactly what will happen. This is a great chance to use the step-and-trace debugger, again.

The goto statement has an interesting history. In a sense, it is a good example of how an idea can be misapplied, abused, and eventually twisted into something the person who came up with the idea did not intend. What I am referring to, of course, is the idea that goto statements are bad. In fact, many people group structured programming, Pascal, and so-called "goto-less programming" together, treating them as synonymous. Some early Pascal compilers even left the goto statement out, or made it an optional feature that was only available if you asked for it. In many classes in Pascal, students are still taught that the goto statement is always bad. Nothing could be farther from the truth.

In a sense, ignoring the goto statement while you learn Pascal is a good idea, up to a point. This is especially true if you learned to program in BASIC or FORTRAN before taking up Pascal. The reason is that you can do anything with the goto statement and an if statement that you can do with repeat loops, while loops, case statements, and if-then-else statements. Experience has shown, though, that most programs written using these statements are easier to read, more efficient, and have fewer bugs that programs written with gotos. So, while you learn the structured statements, and how to use them to organize programs logically, it is a good idea to forget that the goto statement exists.

So the reason we haven't used the goto statement isn't because it is bad, or has no use. The reason we haven't used the goto statement is because it isn't needed as much in Pascal as it is in languages that do not have if-then-else statements, while loops, repeat loops, and case statements. There are two places, though, where the goto statement is very useful, easy to understand, and will make your program much more efficient. These two places are an error exit and an early exit from a loop.

A good example of an early exit from a loop is when you are searching a linked list for a particular item. As a simple example, let's assume that you want to scan a list of names to see if a particular name exists. This problem is a very common one in programming: the list could be a list of names in a customer database, a list of commands that an adventure game recognizes, a dictionary in a spelling checker, or a list of variables in a Pascal program. If the name is in the list, you want to print true. If the name is not in the list, you want to print false.

The most obvious way to write a test of this sort is to loop over the list, testing each element, like this:

```
ptr := names;
found := nil;
while ptr <> nil do begin
   if ptr^.name = name then
      found := name;
   ptr := ptr^.next;
   end; {while}
if found = nil then
   writeln('false')
else
   writeln('true: "', name, '"');
```

This will work, all right, but it takes way too much time. This algorithm will look at each and every element of the list, even if it has already found the one we need. On average, though, we only need to look at half of the elements in the list before we find the right one. If there are ten names from a test program's list of inputs, the speed won't be an issue, but if you are searching through a list of 30,000 words in a dictionary, the difference is very important.

A goto statement, though, can give us the performance we need by leaving the loop as soon as a matching name is found. It also gets rid of the need for the found variable.

```
ptr := names;
while ptr <> nil do begin
   if ptr^.name = name then
      goto 1;
   ptr := ptr^.next;
   end; {while}
1:
if ptr = nil then
   writeln('false')
else
   writeln('true: "', name, '"');
```

By eliminating, on average, half of the passes through the loop, you double the speed of the search. You could eliminate the final test, too, by reorganizing the loop a bit.

```
ptr := names;
while ptr <> nil do begin
   if ptr^.name = name then begin
      writeln('true: "', name, '"');
      goto 1;
      end; {if}
   ptr := ptr^.next;
   end; {while}
writeln('false');
1:
```

It is worth pointing out that you can, in fact, exit the loop statement early without using a goto statement. The idea is to test ptr and found in the while statement, instead of just testing ptr. Compared to the simplicity and efficiency of the goto statement, though, this solution leaves a lot to be desired. There are also many cases in practical programming situations where you need to exit from several nested loops, or from a for loop. In these cases, adding more and more extra tests quickly makes the program far less efficient and harder to read than using a single, simple goto statement.

Listing 10.2 is a complete sample program that uses a goto statement to exit a loop early. It reads a list of names, stopping when you enter a blank string. You can then ask if a name is in the list. The program stops if you hit the RETURN key right away. The sample program also shows a very unusual goto used as an error exit; we'll discuss this in detail in a moment.

The goto 1 to exit the loop is exactly what we were showing with the small code fragments. There is nothing new there, but you do get a chance to see the idea in a complete program. More surprising, though, is the goto 99. This is a goto being used as an error exit. In this case, we don't really have an error, just a situation where the program is finished and we want to quit. In a simple program like this one, it wouldn't be terribly difficult to use a test at the end of the loop for the same purpose, and normally that is what I would have done. In a very large, complex program, though, this loop might have been buried inside other loops, or even nested inside of several other procedures or functions. In that case, adding tests to all of the loops and subroutines to handle an early exit would be

Listing 10.2

```
{ This program reads a list of names from the keyboard,         }
{ stopping when you hit the RETURN key right away.  These       }
{ names are stored in the linked list names.  The program then  }
{ asks you to enter a name, and scans the list.  If the name is }
{ in the list, the program prints the name.  If not, false is   }
{ printed.                                                      }

program ListOfNames(input, output);

label 99;                                {used for program exit}

const
   nameLength = 20;                      {max length of a name}

type
   namePtr = ^nameType;                  {name pointer}
   nameType = record                     {element of the name list}
      next: namePtr;
      name: string[nameLength];
      end;

var
   names: namePtr;                       {list of names}
```

(continued)

difficult, inefficient, and very error prone.

Of course, it is a bit surprising that the goto statement appears inside of the Test procedure, but the label is in the body of the program. This illustrates on of the more unusual capabilities of the goto statement in Pascal. Since the goto statement is primarily used for leaving loops early or error exits, it is legal to jump right out of a procedure or function. You can even jump right out of a function called by another procedure, and so forth.

There are several limitations on the goto statement that you need to keep in mind.

1. You can't jump into another structure. For example, this goto is not legal:

```
goto 1;
repeat
   1: {illegal!}
until true;
```

2. You can jump out of a procedure or function, but you can't jump into a procedure or function.

3. You can't jump from the if part of an if-then-else to the else part, or vice versa. For example, these gotos are not legal:

```
{illegal!}
if condition then begin
   1: goto 2;
   end {if}
else begin
   2: goto 1;
   end; {else}
```

4. If you declare a label in a label statement, the label must be used in the procedure, function or program where the label was declared.

5. The same label number cannot be used on two different statements in the same procedure.

6. Labels must be integers in the range 1..9999.

(continuation of Listing 10.2)

```
procedure ReadList;

{ Read a list of names from the keyboard }

var
   ptr: namePtr;                    {new name record}
   name: string[nameLength];        {name read from keyboard}

begin {ReadList}
names := nil;                       {no names so far}

repeat
   write('Name:');                  {get a name}
   readln(name);
   if length(name) <> 0 then begin
      new(ptr);                     {get a new name record}
      ptr^.next := names;           {add the record to the list}
      names := ptr;
      ptr^.name := name;            {put the name in the record}
      end; {if}
until length(name) = 0;
end; {ReadList}
```

(continued)

Most of these rules make a great deal of sense, if you just stop and think about what it would mean to violate the rule. Jumping into a for loop, for example, wouldn't give the program a chance to set up the loop index variable or calculate the stop condition; that is the reason for rule 1. The only arbitrary rule is rule 6: labels must be numbers, and they must be in the range 1 to 9999. Why the limit of 9999? I have no idea. This limit certainly leaves enough labels, but the arbitrary limit does seem a little odd.

It is a good idea to avoid a goto statement unless it really improves the basic algorithm. The reason is that the goto statement actually blocks some optimizations in a modern, optimizing compiler. By trying to squeeze just a bit more speed from your program by using a goto instead of a test at the end of a loop, you could actually slow your program down.

(continuation of Listing 10.2)

```
      procedure Test;

      { Test to see if names are in the list }

      label 1;

      var
         ptr: namePtr;                    {used to trace the list}
         name: string[nameLength];        {name read from keyboard}

      begin {Test}
      repeat
         write('Name to find:');          {get a name}
         readln(name);
         if length(name) = 0 then         {quit if no name is given}
            goto 99;
         ptr := names;                    {scan for the name}
         while ptr <> nil do begin
            if ptr^.name = name then begin {name found -> write it}
               writeln('true: "', name, '"');
               goto 1;
               end; {if}
            ptr := ptr^.next;
            end; {while}
         writeln('false');                {name not found}
   1:
      until false;                        {loop forever}
      end; {Test}


   begin
   ReadList;
   writeln;
   Test;
   99:
   end.
```

Problem 10.5. In the text, I said that the loop

```
ptr := names;
found := nil;
while ptr <> nil do begin
   if ptr^.name = name then
      found := name;
   ptr := ptr^.next;
   end; {while}
if found = nil then
   writeln('false')
else
   writeln('true: "', name, '"');
```

looped all the way through the list each time. That's fairly easy to see. Computer scientists would say that the algorithm has a run-time of order n, where n is the length of the list. They write this as a big O, meaning order, then the order in parenthesis, like this: O(n).

I also said that the search with a goto statement,

```
ptr := names;
while ptr <> nil do begin
   if ptr^.name = name then begin
      writeln('true: "',
         name, '"');
      goto 1;
      end; {if}
   ptr := ptr^.next;
   end; {while}
writeln('false');
1:
```

only had to scan half of the list, on average. Computer scientists would say that this algorithm has a typical run-time of order n/2, written O(n/2).

The reason computer scientists invented this strange way of talking is that you can see at a glance that the second version of the search works twice as fast as the first. Or does it? If you have a knack for mathematics, you might want to try to prove it. While we technically won't prove it, in this problem you will write a program that will demonstrate the idea with a simulation.

Your program should be based on the sample program from the text. Start by replacing ReadList with a procedure that builds a list of characters, one for each letter in the alphabet. To do this, you will need to change the nameType record so it has a pointer and a character, rather than a pointer and a string. Test your program after doing this to make sure you have everything correct up to this point.

Next, change the Test procedure so that it uses RandomValue to create random characters, instead of asking for a character from the keyboard. Use a for loop to look for a character count times, where count is a constant defined at the top of the program. For test purposes, set count to 5.

The last step in developing the program is to change the Test procedure one last time. Instead of printing whether or not the name is found, use a counter to see how many compares you have to do before finding the correct element of the list. This counter should be declared globally, and it will need to be a long integer, rather than a standard integer. After calling ReadList and Test, the main program should print the average number of searches (the total number of searches divided by count).

With the program in place and debugged, change count to 10,000 to get a fair sample size, turn debug off so you won't have to wait all day, and run your simulation.

Theoretically, the average number of compares per search should be 13.5. How close were you?

Problem 10.6. Some of the goto statements shown below are legal, and some are not. Identify the illegal ones, and be sure you understand why they are illegal.

Note: This problem serves as an example of the legal and illegal uses of the goto statement. Even the legal uses often represent bad programming style. In general, you should only use a goto statement for the reasons stated in the text!

```
a. if condition then begin
      DoIt;
      goto 4;
      end; {if}
   DoThat;
   4:
```

```
b. goto 4;
   DoThat;
   if condition then begin
      4:
      DoIt;
      end; {if}

c. procedure foobar;

   label 3;

      procedure gorp;

      begin
      goto 3;
      end;

   begin
   gorp;
   3:
   end;

d. procedure foobar;

      procedure gorp;

      label 3;

      begin
      3:
      end;

   begin
   goto 3;
   end;

e. procedure Burglary;

   begin
   while alarm do begin
      if policeArrived then
         goto 3;
      RingBell;
      end; {while}
   3:
   end;

f. procedure Burglary;

   label 3,9999;

   begin
   while alarm do begin
      if policeArrived then
         goto 3;
      RingBell;
      end; {while}
   3:
   end;

g. procedure Burglary;

   label 10000;

   begin
   while alarm do begin
      if policeArrived then
         goto 10000;
      RingBell;
      end; {while}
   10000:
   end;
```

Variant Records

We have already seen how records can be used to organize information in our program, grouping any type of variable together into a record about a particular thing. For example, we could use a record to record a person's name, address, and state (all strings), zip code and phone number (possibly integers), and sex (perhaps as an enumeration). All of these facts about a person can be collected into a single variable, so they can be kept together.

What if we need to keep different information about different groups of people, though? For example, a pet store might want to list whether a fish is a salt-water fish or fresh-water fish, but they certainly wouldn't need to waste space on the same information about a dog. For the dog, they might want to list if it has been spayed or neutered, but the same information hardly applies to the fish. Rather than waste space by including all of this information when it isn't needed, a variant record can be used.

In a variant record, you use a tag variable to keep track of what the record is for. For the pet store, for example, the variant record might look like this:

```
type
   animals =
      (bird, fish, dog, cat);
   sexType = (male, female);

   animalPtr = ^animalType;

   animalType = record
      next: animalPtr;
      inStock: 0..maxint;
      case kind: animals of
         bird: ();
         fish: (
            fSex: sexType;
            freshWater: boolean;
            );
         dog,cat: (
            dSex: sexType;
            spayed: boolean;
            );
      end;
```

There is a wealth of information in this record, so we will take a few moments to study it in detail. The first two variables in the record are next and inStock. Up to this point, the record looks exactly like any other record, and it is. These two variables are needed no matter what kind of animal we are dealing with, and they will appear in every record of type animalType.

The case statement is what makes this record a variant record. The case statement looks vaguely like a case statement in a program, but there are differences. In the variant record, the case condition is a new variable, so we have to give it a type. Next comes the case labels, each of which is followed by a list of variables, enclosed in parenthesis. In a case statement in the body of a program, you have to include a case label for each and every situation that could arise, and the same is true here. Since the tag variable, kind, is of type animals, we have to have a label for each and every animal type. For that reason, even though bird doesn't have any variables, we still have to list it; leaving it out would be an error. Of course, you can only list each of the tag constants one time. It wouldn't make sense to define two different sets of variables for the tag constant dog.

In this example, we need to record the same information for dogs and cats, so we list them together. We also decided to record the sex of fish, dogs and cats. Variables in the record must have unique names, even if they appear in different parts of a variant record, so we can't use sex as the name of both variables. To avoid a conflict, we append a unique letter to the start of the variable names, creating fSex for the sex of a fish, and dSex for the sex of a dog or cat. There are other ways to handle the problem of duplicate names, but appending a unique prefix to the variable name is a common solution.

Let's take a look at how the same information would be stored in a standard record, and compare the standard record to the variant record. The standard record would look like this:

```
animalType = record
   next: animalPtr;
   inStock: 0..maxint;
   kind: animals;
   sex: sexType;
   freshWater: boolean;
   spayed: boolean;
   end;
```

This record requires 14 bytes of memory: 4 bytes for the pointer (next), and two bytes for each of the other fields. It also has a freshWater field for birds, dogs and cats, which is not the sort of thing that promotes clarity. The variant record, on the other hand, has a variable size, depending on what kind of animal we are dealing with. In all cases, the size is less than 14 bytes. In the case of a bird, the record has three variables, next, inStock and kind. These variables use 8 bytes of memory. Fish records have five variables, next, inStock, kind, fSex and freshWater, for a total of 12 bytes. Dogs and cats also use 12 bytes of memory for their variables, which are next, inStock, kind, dSex and spayed.

Reserving memory, setting up a record, and accessing information from the record works just like it does in a standard record. The only difference is that you need to set the tag variable (kind, in our example) before setting any of the values in the variant part of the record. For example, this code sets up a variant record that contains information about a dog. We set the tag variable, kind, before setting the variables in the variant part of the record. The variables next and inStock, which are not in the variant part of the record, can be set before or after kind.

```
new(rec);
rec^.next := nil;
rec^.inStock := 4;
rec^.kind := dog;
rec^.dSex := female;
rec^.spayed := false;
```

The incorrect way to assign the variables looks like this:

```
new(rec);
rec^.next := nil;
rec^.inStock := 4;
{incorrect: set kind first!}
rec^.dSex := female;
rec^.spayed := false;
rec^.kind := dog;
```

ORCA/Pascal doesn't check to be sure you set the tag variable before setting values in the variant part of the record, but some compilers might. One simple way to make sure you are doing it right is to assign values to the variables in the same order that they appear in the record. This also helps prevent you from skipping any variables, and makes it easier to check for skipped variables later.

Listing 10.3 shows one use of variant records. In this example, we create and then animate 10 shapes. The shapes can be squares, triangles, or stars. Each of the shapes does a random walk across the screen, moving one pixel in a random direction on each cycle through the program.

To animate the shapes, we need to keep track of what kind of a shape it is, and the coordinates for the shape. Since each shape has a different number of points, we use a variant record. All of the shapes have a color, so that is stored in a non-variant part of the record. Animation is pretty slow with debug code on, so once you get the program to work, try it with debug off.

When your allocate a variant record using new, you can allocate space for it exactly like you would for any other record. As an example, let's consider a variant record that can hold either a real number or an integer:

```
type
   number = record
      case isReal: boolean of
         true: (r: real);
         false: (i: integer);
      end;

var
   ptr = ^number;
```

For an integer, this record is four bytes long; two bytes for the boolean tag variable, and two bytes for the integer. If the record is holding a real number, though, it is six bytes long; two for the tag variable, and four for the real number. When you allocate the record like this:

```
new(ptr);
```

the compiler has no way of knowing whether you want to store a real number or an integer. In fact, you may store a real number now, then convert it to an integer later. The compiler reserves six bytes of space, enough for the largest of the variant records. In many cases, this is exactly what you want the compiler to do, but if you know for sure that you will never store anything but an integer in the record, the extra two bytes are wasted. Our example is of a very small record; the loss of two bytes is probably no big deal. In real programming situations, though, you could easily end up wasting dozens, or even hundreds of bytes. In those situations, Pascal gives you another option. You can list the tag value when you call new, like this:

```
new(ptr,false);
```

Here, you are telling the compiler that the record will be one where isReal is set to false. In effect, you are also promising never to change the value of isReal. The compiler dutifully allocates only four bytes, rather than six. When you dispose of the memory, you should specify the same variant record tags you used in new. In this example, the proper way to dispose of ptr is like this:

```
dispose(ptr,false);
```

Allocating exactly the right amount of space for a variant record is a powerful, useful technique, but it is also dangerous if misused. For example, if you allocate space far an integer, then store a real number in the record, you will wipe out the two bytes right after the record! This will damage something, but you have no way of predicting in advance what it will be. This is a very nasty sort of bug. It doesn't always show up, since you may wipe out something harmless one time, and something valuable on another occasion. It is also a very hard type of bug to find, since it is rarely obvious from seeing what the bug does that the bug was caused by overwriting more bytes than you allocated.

Technically, a Pascal compiler is allowed to check for this kind of an error. The way it does this is to make sure that the tag variable is appropriate for the variable you are setting. For example,

```
new(ptr);
ptr^.isReal := true;
ptr^.i := 3; {error!}
```

Listing 10.3

```
{ Do a random walk with 10 random shapes                          }

program RandomWalk;

uses Common, QuickDrawII;

const
   numShapes = 10;              {# of shapes to animate}
   walkLength = 100;            {# of "steps" in the walk}

   maxX = 316;                  {size of the graphics screen}
   maxY = 83;

type
   shapeKind = (triangle, square, star);

   shapeType = record           {information about one shape}
      color: 0..3;
      case kind: shapeKind of
         triangle: (
            tx1,tx2,tx3: integer;
            ty1,ty2,ty3: integer;
            );
         square: (
            sx1,sx2,sx3,sx4: integer;
            sy1,sy2,sy3,sy4: integer;
            );
         star: (
            px1,px2,px3,px4,px5: integer;
            py1,py2,py3,py4,py5: integer;
            );
      end;

var
   i,j: integer;                 {loop variables}
                                 {shapes: current array of shapes}
                                 {oldShapes: shapes in last position}
   shapes,oldShapes: array[1..numShapes] of shapeType;

(continued)
```

```
(continuation of Listing 10.3)

      procedure InitGraphics;

      { Standard graphics initialization.                              }

      begin {InitGraphics}
      SetPenMode(2);               {pen mode = xor}
      SetPenSize(3,1);             {use a square pen}
      end; {InitGraphics}


      function RandomValue(max: integer): integer;

      { Return a pseudo-random number in the range 1..max.             }
      {                                                                }
      { Parameters:                                                    }
      {    max - largest number to return                              }

      begin {RandomValue}
      RandomValue := (RandomInteger mod max) + 1;
      end; {RandomValue}


      procedure DrawShape(s: shapeType);

      { This subroutine draws one of the shapes on the screen.         }
      {                                                                }
      { Parameters:                                                    }
      {    s - shape to draw                                           }

      begin {DrawShape}
      SetSolidPenPat(s.color);     {set the pen color for the shape}

      case s.kind of

         triangle: begin           {draw a triangle}
            MoveTo(s.tx1,s.ty1);
            LineTo(s.tx2,s.ty2);
            LineTo(s.tx3,s.ty3);
            LineTo(s.tx1,s.ty1);
            end;

   (continued)
```

```
   (continuation of Listing 10.3)

         square: begin              {draw a square}
            MoveTo(s.sx1,s.sy1);
            LineTo(s.sx2,s.sy2);
            LineTo(s.sx4,s.sy4);
            LineTo(s.sx3,s.sy3);
            LineTo(s.sx1,s.sy1);
            end;

         star: begin                {draw a star}
            MoveTo(s.px1,s.py1);
            LineTo(s.px2,s.py2);
            LineTo(s.px3,s.py3);
            LineTo(s.px4,s.py4);
            LineTo(s.px5,s.py5);
            LineTo(s.px1,s.py1);
            end;
         end; {case}
      end; {DrawShape}


      procedure CreateShape(var s: shapeType);

      { This subroutine creates a shape.  The color and initial      }
      { position of the shape are chosen randomly.  The size of the  }
      { shape is based on pre-computed values.                       }
      {                                                              }
      { Parameters:                                                  }
      {    s - shape to create                                       }

      var
         cx,cy: integer;            {center point for the shape}

      begin {CreateShape}
      s.color := RandomValue(3);   {get a color}
      cx := RandomValue(279)+19;   {get the center position, picking the}
      cy := RandomValue(73)+8;    {points so the shape is in the window}

(continued)
```

```
(continuation of Listing 10.3)

      case RandomValue(3) of       {set the initial positions}
         1: begin                  {set up a triangle}
            s.kind := triangle;
            s.tx1 := cx-19;
            s.ty1 := cy+4;
            s.tx2 := cx;
            s.ty2 := cy-8;
            s.tx3 := cx+19;
            s.ty3 := cy+4;
            end;

         2: begin                  {set up a square}
            s.kind := square;
            s.sx1 := cx-15;
            s.sy1 := cy-6;
            s.sx2 := cx+15;
            s.sy2 := cy-6;
            s.sx3 := cx-15;
            s.sy3 := cy+6;
            s.sx4 := cx+15;
            s.sy4 := cy+6;
            end;

         3: begin                  {set up a star}
            s.kind := star;
            s.px1 := cx-13;
            s.py1 := cy+7;
            s.px2 := cx;
            s.py2 := cy-8;
            s.px3 := cx+13;
            s.py3 := cy+7;
            s.px4 := cx-21;
            s.py4 := cy-3;
            s.px5 := cx+21;
            s.py5 := cy-3;
            end;
         end; {case}
      end; {CreateShape}

(continued)
```

```
(continuation of Listing 10.3)

      procedure UpdateShape(var s: shapeType);

      { This subroutine moves a shape across the screen in a random  }
      { walk.                                                        }
      {                                                              }
      { Parameters:                                                  }
      {    s - shape to update                                       }

      var
         dx,dy: integer;           {movement direction}

      begin {UpdateShape}
      dx := RandomValue(3)-2;     {get the walk direction}
      dy := RandomValue(3)-2;

      case s.kind of              {make sure we don't walk off of the}
                                   { screen, then update the position }
         triangle: begin           {check a triangle}
            if dx = -1 then
               if s.tx1 < 1 then
                  dx := 0;
            if dx = 1 then
               if s.tx3 >= maxX then
                  dx := 0;
            if dy = -1 then
               if s.ty2 < 1 then
                  dy := 0;
            if dy = 1 then
               if s.ty3 >= maxY then
                  dy := 0;
            s.tx1 := s.tx1+dx;     {update a triangle}
            s.ty1 := s.ty1+dy;
            s.tx2 := s.tx2+dx;
            s.ty2 := s.ty2+dy;
            s.tx3 := s.tx3+dx;
            s.ty3 := s.ty3+dy;
            end;

(continued)
```

```
(continuation of Listing 10.3)

         square: begin             {check a square}
            if dx = -1 then
               if s.sx1 < 1 then
                  dx := 0;
            if dx = 1 then
               if s.sx2 >= maxX then
                  dx := 0;
            if dy = -1 then
               if s.sy1 < 1 then
                  dy := 0;
            if dy = 1 then
               if s.sy3 >= maxY then
                  dy := 0;
            s.sx1 := s.sx1+dx;     {update a square}
            s.sy1 := s.sy1+dy;
            s.sx2 := s.sx2+dx;
            s.sy2 := s.sy2+dy;
            s.sx3 := s.sx3+dx;
            s.sy3 := s.sy3+dy;
            s.sx4 := s.sx4+dx;
            s.sy4 := s.sy4+dy;
            end;

         star: begin               {check a star}
            if dx = -1 then
               if s.px4 < 1 then
                  dx := 0;
            if dx = 1 then
               if s.px5 >= maxX then
                  dx := 0;
            if dy = -1 then
               if s.py2 < 1 then
                  dy := 0;
            if dy = 1 then
               if s.py1 >= maxY then
                  dy := 0;

(continued)
```

```
(continuation of Listing 10.3)

            s.px1 := s.px1+dx;     {update a star}
            s.py1 := s.py1+dy;
            s.px2 := s.px2+dx;
            s.py2 := s.py2+dy;
            s.px3 := s.px3+dx;
            s.py3 := s.py3+dy;
            s.px4 := s.px4+dx;
            s.py4 := s.py4+dy;
            s.px5 := s.px5+dx;
            s.py5 := s.py5+dy;
            end;
         end; {case}
      end; {UpdateShape}


   begin
   InitGraphics;                  {set up the graphics window}
   Seed(6289);                    {initialize the random number generator}

   for i := 1 to numShapes do     {set up and draw the initial shapes}
      begin
      CreateShape(shapes[i]);
      DrawShape(shapes[i]);
      end; {for}

   for i := 1 to walkLength do    {do the random walk}
      begin
      oldShapes := shapes;        {move the shapes}
      for j := 1 to numShapes do
         UpdateShape(shapes[j]);
      for j := 1 to numShapes do  {redraw the shapes}
         begin
         DrawShape(shapes[j]);
         DrawShape(oldShapes[j]);
         end; {for}
      end; {for}
   end.
```

is an error, since the record is supposed to hold a real number, but you have assigned an integer. Making this kind of check is very expensive in terms of code size and execution speed, though, so very few compilers actually make the check.

There are many cases where you want to use a variant record, but the value of the tag variable is not needed, because the kind of the record is known already, or doesn't matter. In that case, you can create a variant record with a tag field type, but no tag field variable. A good example of this is the rect record in the Common.Intf file, an interface file that defines types for the toolkit. (The source code for this interface file is in the file COMMON.INTF, located in the interface folder on the samples disk that comes with ORCA/Pascal.) The record, along with the types point and rectKinds, which are used in the record, looks like this:

```
point = record
    v: integer;
    h: integer;
    end;

rectKinds = (normal, mac, points);

rect = record
  case rectKinds of
    normal:  (v1: integer;
              h1: integer;
              v2: integer;
              h2: integer);

    mac:     (top:    integer;
              left:   integer;
              bottom: integer;
              right:  integer);

    points:  (topLeft:  point;
              botRight: point);
    end;
```

In this case, what was needed was three different ways to refer to the points in a rectangle. The reason three ways were needed is mostly historical. The Apple IIGS toolbox reference manual uses v1, v2, h1 and h2 as the names of the points that define a rectangle. On the Macintosh, top, bottom, left and right are used, and some of the early Apple IIGS compilers used the Macintosh names instead of the ones in the toolbox manuals. The Macintosh toolbox also defined topLeft and botRight, so you could refer to the corners of the rectangle as points.

In each case, the same information is in the record: four coordinates that identify the size of a rectangle. By using a variant record, these three versions of the record lay overtop of each other in memory. So, if you define a rect variable called r,

```
r.h1 := 4;
writeln(r.left);
```

is perfectly legal. A four would be written to the screen. The effect is to give you three ways to access the information.

Leaving out the tag variable has one other effect on the record. In ORCA/Pascal, and in most other Pascals, if you leave out the tag variable, the compiler does not leave room for the variable in the record. This, of course, makes the record a little shorter. In the rect record, shown above, the length of the record is eight bytes (the record has four integers, each of which is two bytes long).

Problem 10.7. One common use of variant records takes advantage of the fact that the variables in the variant part overlap. This fact can be used to examine the values of a complicated variable type. You have already seen one example, the rect structure used in the toolbox. This variant record overlays three versions of a rectangle, allowing you to refer to the points that define the rectangle as either two points or four values. Of course, programs that do this are not portable.

One thing that happens over and over in the toolbox is to extract the least significant 16 bits from a long integer, or the most significant 16 bits. You can do this with math operations if you are very careful, but it is much easier and faster to do it with a variant record.

Define a variant record with a boolean tag type and no tag variable. For the true case, define a long integer. For the false case, define two integers, lsw and msw, in that order. This record puts the two integers in the same memory as the long integer, so that you can save a long value and then extract the integer parts.

Write a program that reads long integers from the keyboard, looping until a 0 is entered. Save this value in the record, then write the two integers.

Experiment with this program a bit. What you should find is that for values up to maxint, the program prints the same value you entered for the least significant integer (the first one), then a zero for the most significant integer (the second one). As the numbers get larger, you start to fill in the sign bit, so the first integer is written as a negative number. Finally, when the numbers exceed 65535, values start to show up in the second integer.

Problem 10.8. The rect you saw in the text is one of the basic QuickDraw structures. It can be used to do a variety of wonderful things. The rect record is used to define the basic drawing area for arcs, ovals, rectangles, and rounded rectangles.

Write a program that draws these four basic shapes in the graphics window. Each shape should be a little less that half of the window's width and height, with one shape in the upper left, one in the upper right, one in the lower left, and one in the lower right of the window.

The calls to paint these shapes look like this:

```
PaintArc(r, start, arc);
PaintOval(r);
```

```
PaintRect(r);
PaintRRect(r, ovalWidth,
   ovalHeight);
```

In each case, the rectangle (in this case, r) defines the top, bottom, left and right edge for the object. In the case of a rectangle or oval, that's all you need.

An arc, in QuickDraw, is a portion of an oval. Start and arc are two angles, measured clockwise from the vertical. The angles are given in degrees. To draw an arc in the lower-right corner of the rectangle, for example, you would use 90 for start, and 90 for arc. Use these measurements for your program.

A rounded rectangle is a rectangle with rounded corners. The two parameters, ovalWidth and ovalHeight, define the size of an oval drawn at the corner of the rectangle. For our sample, try 36 for ovalWidth, and 12 for ovalHeight.

For more information about shapes, see volume 2 of the Toolbox Reference manual. Shapes are discussed int he introduction to QuickDraw, right at the front of the book.

Problem 10.9. So far, all of our programs that had any knowledge of the size of the graphics window used constants, like maxX and maxY. As you probably know, though, it is easy enough to change the size and location of the graphics window; it can be dragged and sized just like any other window.

You can read the size of the window from your program, though. The QuickDraw call GetPortRect fills in the window size as a rectangle. The top left corner is zero, while the bottom right corner gives the size of the drawing window. Here is an example of a call to GetPortRect, where r is defined as a rect record in your program:

```
GetPortRect(r);
```

Write a program that finds the size of the graphics window, and draws a large X across the window. The X should be drawn from the top-left corner to the bottom-right corner, and from the bottom-left corner to the top-right corner. Try your program several times, moving and resizing the graphics window between each run.

The With Statement

Using records can get tedious. For example, in the sample program that animates shapes, we had several assignments that looked like this:

```
s.px1 := s.px1+dx;
s.py1 := s.py1+dy;
s.px2 := s.px2+dx;
s.py2 := s.py2+dy;
s.px3 := s.px3+dx;
s.py3 := s.py3+dy;
s.px4 := s.px4+dx;
s.py4 := s.py4+dy;
s.px5 := s.px5+dx;
s.py5 := s.py5+dy;
```

Typing all of those s.'s can get to be boring. If we were dealing with an array of pointers to records, with somewhat longer names, the assignments might look more like this:

```
shapeArray[i]^.px1 :=
   shapeArray[i]^.px1+dx;
shapeArray[i]^.py1 :=
   shapeArray[i]^.py1+dy;
shapeArray[i]^.px2 :=
   shapeArray[i]^.px2+dx;
shapeArray[i]^.py2 :=
   shapeArray[i]^.py2+dy;
shapeArray[i]^.px3 :=
   shapeArray[i]^.px3+dx;
shapeArray[i]^.py3 :=
   shapeArray[i]^.py3+dy;
shapeArray[i]^.px4 :=
   shapeArray[i]^.px4+dx;
shapeArray[i]^.py4 :=
   shapeArray[i]^.py4+dy;
shapeArray[i]^.px5 :=
   shapeArray[i]^.px5+dx;
shapeArray[i]^.py5 :=
   shapeArray[i]^.py5+dy;
```

This is beyond boring. It is absolutely tedious to do all of that typing. It also makes the program harder to read. There is a solution, though. The solution is the with statement.

The with statement tells the compiler to append some prefix to any variable that needs it. The best way to understand what this means is with an example.

Using a with statement, the first set of assignments looks like this:

```
with s do begin
   px1 := px1+dx;
   py1 := py1+dy;
   px2 := px2+dx;
   py2 := py2+dy;
   px3 := px3+dx;
   py3 := py3+dy;
   px4 := px4+dx;
   py4 := py4+dy;
   px5 := px5+dx;
   py5 := py5+dy;
   end; {with}
```

When it is processing this with statement, the Pascal compiler looks at the variables that appear, and checks to see if the variable is in the record s. If so, the compiler treats the variable as if there was an s. at the beginning of the variable. For example, px1 appears in the record, so the compiler treats it as if you had typed s.px1. The variable dx does not appear in the record, though, so the compiler leaves it alone. Judicious use of the with statement makes you program easier to read and modify.

You can use a with statement for any record, including an array of pointers to records, as in our second example. The savings in typing and clarity is much greater here. That monstrous second set of assignments shortens to this:

```
with shapeArray[i]^ do begin
   px1 := px1+dx;
   py1 := py1+dy;
   px2 := px2+dx;
   py2 := py2+dy;
   px3 := px3+dx;
   py3 := py3+dy;
   px4 := px4+dx;
   py4 := py4+dy;
   px5 := px5+dx;
   py5 := py5+dy;
   end; {with}
```

There is another advantage to the with statement that is far less obvious. When a compiler generates code for the statement

```
shapeArray[i]^.py4 :=
   shapeArray[i]^.py4+dy;
```

it creates the machine code to go i units past the start of the array shapeArray and dereference the pointer two times. That takes a lot of machine code instructions. In our example, we had 10 assignments like that one, so the compiler would create 20 identical sequences of machine code. If you use a with statement, the compiler calculates the address of the record one time and saves it, instead of calculating the address 20 times. In some programs, the savings in both program size and execution speed can mount up very quickly.

You can nest with statements just like you would nest any other statement. If there is a conflict between two records, the most recent with statement is used.

```
type
   r1 = record
      r: real;
      i: integer;
      end;

   r2 = record
      r: real;
      l: longint;
      end;
var
   v1: r1;
   v2: r2;

...

with r1 do
   with r2 do begin
      r := 3.1;
      i := 4;
      l := 100000;
      end;
```

The values for i and l are pretty clear, but there is a variable called r in both v1 and v2. Since r2 is in the most recent with statement, it is r2.r that gets set. The compiler treats the statements like this:

```
r2.r := 3.1;
r1.i := 4;
r2.l := 100000;
```

You can also specify more than one record in the with statement, separating the records with commas. Our nested with statements could be replaced with

```
with r1, r2 do begin
```

The compiler treats this exactly as if you typed two nested with statements. It's just a short cut to save some typing.

Problem 10.10. Modify the sample program RandomWalk to use the with statement, rather than repeatedly using s. to refer to the records.

Lesson Ten
Solutions to Problems

Solution to problem 10.1.

```
{ This program decodes cyphers that associate letters with      }
{ sequential numbers.                                           }

{$rangecheck+}

program Cypher(input, output);

type
   values = 0..27;                      {legal code numbers}
   alpha = 'A'..'Z';                    {alphabetic characters}

var
   base: values;                        {base number}
   code: array[values] of alpha;        {decode array}
   done: boolean;                       {used for loop termination}
   num: integer;                        {number from keyboard}

   indexCh: 'A'..'[';                {character; used to fill code array}
   index: values;                       {index while filling code array}

begin
write('Starting number:');              {get the starting number}
readln(base);
indexCh := 'A';                         {fill the decode array}
index := base-1;
repeat
   code[index+1] := indexCh;
   indexCh := succ(indexCh);
   index := (index + 1) mod 26;
until indexCh = succ('Z');
```

```
done := false;                          {decode the message}
repeat
   read(num);
   if num = 0 then
      done := true
   else if (num >= 1) and (num <= 26) then
      write(code[num])
   else if num = 27 then
      write(' ')
   else begin
      writeln;
      writeln(num, ' is not a valid code number.');
      done := true;
      end; {else}
until done;
end.
```

Solution to problem 10.2.

```
{ Hex to decimal converter }

program HextoDec (input, output);

const
   lineLength = 20;                     {max length of a line}

var
   i: 1..lineLength;                    {loop index}
   line: string[lineLength];            {input line}
   val: longint;                        {decimal value}


   function ToUpper(ch: char): char;

   { Convert a character to uppercase           }
   {                                            }
   { Parameters:                                }
   {    ch - character to convert               }
   {                                            }
   { Returns:                                   }
   {    Uppercase equivalent of ch              }

   begin {ToUpper}
   if ch in ['a'..'z'] then
      ch := chr(ord(ch) - ord('a') + ord('A'));
   ToUpper := ch;
   end; {ToUpper}
```

```
   function Value(ch: char): integer;

   { Return the value of a hexadecimal digit    }
   {                                            }
   { Parameters:                                }
   {    ch - character to convert               }
   {                                            }
   { Returns:                                   }
   {    Numeric value of the hexadecimal digit  }

   var
      val: integer;                     {value of the character}

   begin {Value}
   ch := ToUpper(ch);
   val := ord(ch) - ord('0');
   if val > 9 then
      val := val-7;
   Value := val;
   end; {Value}


begin
repeat
   write('Hex number:');
   readln(line);
   if length(line) <> 0 then begin
      val := 0;
      for i := 1 to length(line) do
         if line[i] in ['0'..'9','A'..'F','a'..'f'] then
            val := val*16 + Value(line[i])
         else
            writeln(line[i], ' is not a valid hexadecimal digit.');
      writeln('Decimal number: ', val:1);
      end; {if}
until length(line) = 0;
end.
```

Solution to problem 10.3.

```
{ Reverse the characters in a string }

program Reverse (input, output);

const
   lineLength = 80;                    {max length of a line}

var
   i: 1..lineLength;                   {loop index}
   line: string[lineLength];           {input line}

begin
repeat
   write('String:');
   readln(line);
   for i := length(line) downto 1 do
      write(line[i]);
   writeln;
   writeln;
until length(line) = 0;
end.
```

Solution to problem 10.4.

a. Legal.
b. Legal. The body of the for loop is skipped, but the loop is still legal.
c. Legal.
d. Legal.
e. Illegal. You cannot use the same loop variable in nested for loops.
f. Illegal. You cannot assign a value to the loop variable in the body of the loop.
g. Legal.
h. Illegal. While many compilers, including ORCA/Pascal, will not complain about this for loop, the value of the loop variable once you leave the for loop is not predictable.
i. Legal.

Solution to problem 10.5.

With the seed value used in this program, my simulation gave a result of 13.401. This is very close to the theoretical value of 13.5.

```
{ This program tests the hypothesis that the loop with a goto   }
{ exit does half the number of compares, on average, as the     }
{ loop that does not use a goto statement.                      }

program TestLoopEfficiency (output);

const
   count = 10000;                        {number of times to search}

type
   charPtr = ^charType;                  {char record pointer}
   charType = record                     {element of the char list}
      next: charPtr;
      ch: char;
      end;

var
   chars: charPtr;                       {list of chars}
   compares: longint;                    {# of compares}


   procedure MakeList;

   { Create the list of characters }

   var
      ch: char;                          {loop variable}
      ptr: charPtr;                      {new char record}

   begin {MakeList}
   chars := nil;
   for ch := 'a' to 'z' do begin
      new(ptr);
      ptr^.next := chars;
      chars := ptr;
      ptr^.ch := ch;
      end; {for}
   end; {MakeList}
```

```
   function RandomValue(max: integer): integer;

   { Return a pseudo-random number in the range 1..max.          }
   {                                                             }
   { Parameters:                                                 }
   {    max - largest number to return                           }

   begin {RandomValue}
   RandomValue := (RandomInteger mod max) + 1;
   end; {RandomValue}


   procedure Test;

   { Test to see if characters are in the list }

   label 1;

   var
      i: 1..count;                    {loop counter}
      ptr: charPtr;                   {used to trace the list}
      ch: char;                       {char to find}

   begin {Test}
   for i := 1 to count do begin
      ch := chr(RandomValue(26)-1+ord('a')); {get a char}
      ptr := chars;                   {scan for the char}
      while ptr <> nil do begin
         compares := compares+1;
         if ptr^.ch = ch then         {char found -> exit loop}
            goto 1;
         ptr := ptr^.next;
         end; {while}
1:
      end; {for}
   end; {Test}


begin
Seed(6289);
compares := 0;
MakeList;
Test;
writeln('Average # of compares per search = ', compares/count:1:3);
end.
```

Solution to problem 10.6.

a. Legal.
b. Illegal. You cannot jump into the body of an if statement.
c. Legal.
d. Illegal. You cannot jump into a procedure.
e. Illegal. The label 3 is not defined in a label statement.
f. Illegal. The label 9999 does not appear on a statement.
g. Illegal. The largest valid label number is 9999.

Solution to problem 10.7.

```
{ Extract the words that make up a long integer }

program Extract(input, output);

var
   value: record
      case boolean of
         true:  (l: longint);
         false: (lsw,msw: integer);
      end;

begin
repeat
   write('long integer:');
   readln(value.l);
   writeln('least significant word: ', value.lsw:1);
   writeln('most significant word : ', value.msw:1);
   writeln;
until value.l = 0;
end.
```

Solution to problem 10.8.

```
{ Draw the basic rectangular shapes }

program Shapes;

uses Common, QuickDrawII;

var
   r: rect;                              {shape rectangle}

begin
SetPenMode(0);                           {set up QuickDraw}
SetPenSize(3,1);
SetSolidPenPat(1);
r.h1 := 6;                               {draw an arc}
r.h2 := 152;
r.v1 := 2;
r.v2 := 39;
PaintArc(r, 90, 90);
r.h1 := 164;                             {draw an oval}
r.h2 := 310;
PaintOval(r);
r.v1 := 43;                              {draw a rounded rectangle}
r.v2 := 81;
PaintRRect(r, 36, 12);
r.h1 := 6;                               {draw a rectangle}
r.h2 := 152;
PaintRect(r);
end.
```

Solution to problem 10.9.

```
{ Draw an X across the graphics window }

program Shapes;

uses Common, QuickDrawII;

var
   r: rect;                              {shape rectangle}

begin
SetPenMode(0);                           {set up QuickDraw}
SetPenSize(3,1);
SetSolidPenPat(2);
GetPortRect(r);                          {get the size of the window}
MoveTo(r.h1,r.v1);                       {draw the X}
LineTo(r.h2,r.v2);
MoveTo(r.h1,r.v2);
LineTo(r.h2,r.v1);
end.
```

Solution to problem 10.10.

```
{ Do a random walk with 10 random shapes                          }

program RandomWalk;

uses Common, QuickDrawII;

const
   numShapes = 10;              {# of shapes to animate}
   walkLength = 100;            {# of "steps" in the walk}

   maxX = 316;                  {size of the graphics screen}
   maxY = 83;
```

```
type
   shapeKind = (triangle, square, star);

   shapeType = record           {information about one shape}
      color: 0..3;
      case kind: shapeKind of
         triangle: (
            tx1,tx2,tx3: integer;
            ty1,ty2,ty3: integer;
            );
         square: (
            sx1,sx2,sx3,sx4: integer;
            sy1,sy2,sy3,sy4: integer;
            );
         star: (
            px1,px2,px3,px4,px5: integer;
            py1,py2,py3,py4,py5: integer;
            );
      end;

var
   i,j: integer;               {loop variables}
                               {shapes: current array of shapes}
                               {oldShapes: shapes in last position}
   shapes,oldShapes: array[1..numShapes] of shapeType;


   procedure InitGraphics;

   { Standard graphics initialization.                          }

   begin {InitGraphics}
   SetPenMode(2);              {pen mode = xor}
   SetPenSize(3,1);            {use a square pen}
   end; {InitGraphics}


   function RandomValue(max: integer): integer;

   { Return a pseudo-random number in the range 1..max.         }
   {                                                            }
   { Parameters:                                                }
   {    max - largest number to return                          }

   begin {RandomValue}
   RandomValue := (RandomInteger mod max) + 1;
   end; {RandomValue}
```

```
procedure DrawShape(s: shapeType);

{ This subroutine draws one of the shapes on the screen.       }
{                                                              }
{ Parameters:                                                  }
{    s - shape to draw                                         }

begin {DrawShape}
SetSolidPenPat(s.color);     {set the pen color for the shape}

with s do
   case kind of

      triangle: begin        {draw a triangle}
         MoveTo(tx1,ty1);
         LineTo(tx2,ty2);
         LineTo(tx3,ty3);
         LineTo(tx1,ty1);
         end;

      square: begin          {draw a square}
         MoveTo(sx1,sy1);
         LineTo(sx2,sy2);
         LineTo(sx4,sy4);
         LineTo(sx3,sy3);
         LineTo(sx1,sy1);
         end;

      star: begin            {draw a star}
         MoveTo(px1,py1);
         LineTo(px2,py2);
         LineTo(px3,py3);
         LineTo(px4,py4);
         LineTo(px5,py5);
         LineTo(px1,py1);
         end;
      end; {case}
end; {DrawShape}
```

```
procedure CreateShape(var s: shapeType);

{ This subroutine creates a shape.  The color and initial      }
{ position of the shape are chosen randomly.  The size of the  }
{ shape is based on pre-computed values.                       }
{                                                              }
{ Parameters:                                                  }
{    s - shape to create                                       }

var
   cx,cy: integer;           {center point for the shape}

begin {CreateShape}
s.color := RandomValue(3);   {get a color}
cx := RandomValue(279)+19;   {get the center position, picking the}
cy := RandomValue(73)+8;     {points so the shape is in the window}

with s do
   case RandomValue(3) of    {set the initial positions}
      1: begin               {set up a triangle}
         kind := triangle;
         tx1 := cx-19;
         ty1 := cy+4;
         tx2 := cx;
         ty2 := cy-8;
         tx3 := cx+19;
         ty3 := cy+4;
         end;

      2: begin               {set up a square}
         kind := square;
         sx1 := cx-15;
         sy1 := cy-6;
         sx2 := cx+15;
         sy2 := cy-6;
         sx3 := cx-15;
         sy3 := cy+6;
         sx4 := cx+15;
         sy4 := cy+6;
         end;
```

```
      3: begin               {set up a star}
         kind := star;
         px1 := cx-13;
         py1 := cy+7;
         px2 := cx;
         py2 := cy-8;
         px3 := cx+13;
         py3 := cy+7;
         px4 := cx-21;
         py4 := cy-3;
         px5 := cx+21;
         py5 := cy-3;
         end;
      end; {case}
end; {CreateShape}


procedure UpdateShape(var s: shapeType);

{ This subroutine moves a shape across the screen in a random  }
{ walk.                                                        }
{                                                              }
{ Parameters:                                                  }
{    s - shape to update                                       }

var
   dx,dy: integer;           {movement direction}

begin {UpdateShape}
dx := RandomValue(3)-2;      {get the walk direction}
dy := RandomValue(3)-2;
```

```
with s do
   case kind of              {make sure we don't walk off of the}
                             { screen, then update the position }
      triangle: begin        {check a triangle}
         if dx = -1 then
            if tx1 < 1 then
               dx := 0;
         if dx = 1 then
            if tx3 >= maxX then
               dx := 0;
         if dy = -1 then
            if ty2 < 1 then
               dy := 0;
         if dy = 1 then
            if ty3 >= maxY then
               dy := 0;
         tx1 := tx1+dx;      {update a triangle}
         ty1 := ty1+dy;
         tx2 := tx2+dx;
         ty2 := ty2+dy;
         tx3 := tx3+dx;
         ty3 := ty3+dy;
         end;

      square: begin          {check a square}
         if dx = -1 then
            if sx1 < 1 then
               dx := 0;
         if dx = 1 then
            if sx2 >= maxX then
               dx := 0;
         if dy = -1 then
            if sy1 < 1 then
               dy := 0;
         if dy = 1 then
            if sy3 >= maxY then
               dy := 0;
         sx1 := sx1+dx;      {update a square}
         sy1 := sy1+dy;
         sx2 := sx2+dx;
         sy2 := sy2+dy;
         sx3 := sx3+dx;
         sy3 := sy3+dy;
         sx4 := sx4+dx;
         sy4 := sy4+dy;
         end;
```

```
         star: begin            {check a star}
            if dx = -1 then
               if px4 < 1 then
                  dx := 0;
            if dx = 1 then
               if px5 >= maxX then
                  dx := 0;
            if dy = -1 then
               if py2 < 1 then
                  dy := 0;
            if dy = 1 then
               if py1 >= maxY then
                  dy := 0;
            px1 := px1+dx;      {update a star}
            py1 := py1+dy;
            px2 := px2+dx;
            py2 := py2+dy;
            px3 := px3+dx;
            py3 := py3+dy;
            px4 := px4+dx;
            py4 := py4+dy;
            px5 := px5+dx;
            py5 := py5+dy;
            end;
         end; {case}
   end; {UpdateShape}


begin
InitGraphics;                  {set up the graphics window}
Seed(6289);                    {initialize the random number generator}

for i := 1 to numShapes do     {set up and draw the initial shapes}
   begin
   CreateShape(shapes[i]);
   DrawShape(shapes[i]);
   end; {for}

for i := 1 to walkLength do    {do the random walk}
   begin
   oldShapes := shapes;        {move the shapes}
   for j := 1 to numShapes do
      UpdateShape(shapes[j]);
   for j := 1 to numShapes do  {redraw the shapes}
      begin
      DrawShape(shapes[j]);
      DrawShape(oldShapes[j]);
      end; {for}
   end; {for}
end.
```

Lesson Eleven
Stand-Alone Programs

What is a Stand-Alone Program?

So far, all of your programs have been executed from the PRIZM desktop programming environment. That's fine for developing and testing programs, but once the program is finished, you probably want to run it from the Finder, or some other program launcher, just like you run other programs. If you have tried to do that with one of the programs you have written, you found that the Finder doesn't think your program is really a program.

The reason for this is that there are two basic kinds of executable files on the Apple IIGS. (There are also several special forms of executable files.) The two kinds of executable files are S16 files and EXE files. S16 files are the kind that the Finder and other program launchers recognize; these are the ones we call stand-alone programs. You can also run S16 programs from ORCA/Pascal, but there is a very important difference between the way an S16 program is handled, and the way an EXE program is handled. When ORCA/Pascal runs an S16 program, it runs it the same way the Finder does: ORCA/Pascal shuts itself down, then runs the program. Once your program finishes, the Apple IIGS's operating system returns to ORCA/Pascal. When ORCA/Pascal runs an EXE program, it does not shut down. Because ORCA/Pascal is still there, you can see your source file, use the debugger, and take advantage of the fact that ORCA/Pascal has already started all of the tools you usually need. That makes it a lot easier to write simple programs, because you have less to worry about. It also makes the development process faster, since you don't have to wait for ORCA/Pascal to shut itself down before running the program, and start back up when your program is finished. That's basically why we have two file types. S16 files can be executed from any environment, like the Finder, but EXE files can only be executed from the safety of a programming environment.

Using StartGraph and EndGraph

The simplest kind of stand-alone program that you can write using ORCA/Pascal is a graphics program. In many ways, stand-alone graphics programs are written just like the graphics programs you have already written that run from the graphics window. The first, and most important difference, is that a stand-alone program must initialize QuickDraw before making any graphics calls. Initializing tools from Pascal is not particularly easy, so ORCA/Pascal has a procedure that will initialize QuickDraw for you. The procedure is called StartGraph. When you start a tool, you need to shut it down, too. ORCA/Pascal has a procedure called EndGraph to do that.

The StartGraph procedure can actually do one thing you haven't been able to do from within the programming environment. The Apple IIGS has two different graphics resolutions. The PRIZM desktop development environment that you have been using runs in a graphics mode that gives you a screen 640 pixels wide, and 200 pixels high. Each of the pixels can be one of four different colors. This is a convenient mode for programs like PRIZM that need to display text. With 640 pixels to play with, PRIZM can display nearly 80 characters on each line. If it weren't for the scroll bar, it would actually display a full 80 characters.

The other graphics mode has half the number of pixels across the width of the text screen; the screen is 320 pixels by 200 pixels. This just isn't enough room to display text files, but it is great for pictures, because each of the pixels can be any one of 16 colors.

StartGraph can set up QuickDraw for either of these two graphics modes. When you call StartGraph, you pass a single integer, either 320 or 640. The number tells the subroutine which graphics mode to use when it starts QuickDraw. You can use StartGraph and EndGraph from PRIZM while you are testing your program. Since PRIZM must use the 640 by 200 graphics mode, it prevents StartGraph from kicking in the 320 by 200 graphics mode. Your program will work, but everything will be squashed to half its normal width, and only four colors will be displayed.

StartGraph does a couple of other things that are a bit different than what you are used to. StartGraph clears the graphics screen to black, sets the pen color to white, and sets the pen mode to OR. The reason for this is that StartGraph was designed for engineers who wanted to write stand-alone graphics programs without dealing with the desktop programming environment. The black screen, white pen, and drawing mode match what they are used to on graphics terminals. Since that isn't what you are used to, you need to know how to switch everything back. Our first sample program, shown in Listing 11.1, shows you how. Go ahead and run this like a normal program first; we will talk about how to make it a stand-alone program in a moment.

Listing 11.1

```
{ Draw an X across the screen }

program X (input);

uses Common, QuickDrawII;

var
   r: rect;                    {size of the graphics area}

   procedure InitGraphics;

   { Standard graphics initialization.                          }

   var
      r: rect;                 {screen size}

   begin {InitGraphics}
   StartGraph(640);            {initialize QuickDraw}
   SetPenMode(0);              {pen mode = copy}
   SetPenSize(3,1);            {use a square pen}
   SetSolidPenPat(3);          {paint the screen white}
   GetPortRect(r);
   PaintRect(r);
   SetSolidPenPat(0);          {use a black pen}
   end; {InitGraphics}

begin
InitGraphics;                  {set up the graphics screen}
GetPortRect(r);                {draw the X}
MoveTo(r.h1,r.v1);
LineTo(r.h2,r.v2);
MoveTo(r.h1,r.v2);
LineTo(r.h2,r.v1);
readln;                        {wait for the user to press RETURN}
EndGraph;                      {shut down QuickDraw}
end.
```

There are two steps involved in making this a stand-alone program. Both are very important. The first is to turn off the debugger. The source-level debugger is a great help when you are trying to debug a program, but it works by imbedding something called a COP vector throughout your program. These COP vectors work like subroutine calls, telling the debugger what to do as your program runs. If you forget to turn debug code off, and leave these COP vectors in your program, the program will crash when you try to run it from the Finder. In the past, you have turned debug code off to get more speed from your program. Forgetting to do it now does a bit more than just making your program slow!

Of course, eventually you will forget. No real harm is done when the program crashes; you just have to reboot and start again.

The second step is to tell ORCA/Pascal to produce an S16 file, instead of the EXE file it normally produces. You should only do this after your program has been written and debugged. To set the file type, pull down the Run menu and select the Link command.

Listing 11.2

```
{ Exploring text mixed with graphics. }

program TextAndGraphics(output);

uses Common, QuickDrawII;

   procedure InitGraphics;

   { Standard graphics initialization.                          }

   var
      r: rect;                  {screen size}

   begin {InitGraphics}
   StartGraph(640);             {initialize QuickDraw}
   SetPenMode(0);               {pen mode = copy}
   SetPenSize(3,1);             {use a square pen}
   SetSolidPenPat(3);           {paint the screen white}
   GetPortRect(r);
   PaintRect(r);
   SetSolidPenPat(0);           {use a black pen}
   end; {InitGraphics}

begin
InitGraphics;                   {set up the graphics screen}
SetForeColor(0);                {write some text}
SetBackColor(3);
MoveTo(10,10);
writeln('You can get');
write  ('some ');
SetForeColor(3);
SetBackColor(0);
writeln('interesting effects');
SetForeColor(2);
SetBackColor(3);
writeln('by mixing text');
SetForeColor(1);
write  ('with graphics!');
EndGraph;                       {shut down QuickDraw}
end.
```

The link dialog will show up. In the dialog, you will see a row of four radio buttons, labeled EXE, S16, CDA and NDA. At the moment, EXE is selected; push S16 to change the file type. I would also recommend turning off the "Execute after linking" option. With a stand-alone program, you are better off running it from the Finder. It takes less time to do it that way. Once you have made your selections, press Set Options.

If you were wondering, the other two file type radio buttons help you create classic desk accessories (CDA) and new desk accessories (NDA). We won't go into these in this course, but your Pascal reference manual does have a tutorial introduction to writing desk

accessories. You have enough background now to write CDAs; you might want to give it a try. Writing NDAs assumes a pretty good knowledge of the toolbox, though, so you should probably stay away from them until you have written a few standard desktop programs.

With all of the changes made, you are ready to create your first stand-alone program. Use Compile to Memory, as you always do. Since you turned off the Execute after Linking option, though, you only create the program; it does not run. Get into the Finder (or whatever program launcher you usually use to run commercial programs) and give your program a try!

Problem 11.1. In the last lesson, there was a fairly long sample program called RandomWalk. Convert that program to a stand-alone program. While you are at it, switch it to 320 mode. You will need to cut the X coordinates for all of the shapes in half. In addition, change the colors so the program can select any color from 1 to 15 for the shapes, instead of 1 to 3.

Mixing Text and Graphics

When you are running a program from the development environment, you have a very clear idea of where things go. When you use QuickDraw to draw on the screen, the drawings show up in the graphics window. When you use writeln or readln to deal with text, things happen in the shell window.

When you write a stand-alone program, though, things aren't quite that well defined. There isn't a development system to bring up two separate windows, one for text and one for graphics, and keep track of what goes where for you. If you look through the tool calls that are available from QuickDraw, the font manager, and so forth, you will find a new set of commands to draw text on the screen. So what happens to writeln and readln?

The answer, fortunately, is that ORCA/Pascal still handles them in stand-alone graphics programs. There is some set-up involved, though. It's done by StartGraph, which sets things up for your program so that text output calls are converted to calls to the tools that draw text on the screen. You can read text from the screen, too. In fact, the only thing you loose is automatic scrolling.

Text is rerouted to the graphics screen anytime you use StartGraph to start QuickDraw. In the last section, you saw that we can use StartGraph in a program without making it a stand-alone program. The last section probably convinced you that you want to stay in the development environment while you are working on a program, too, since the development cycle goes much faster that way. We'll make use of this to try mixing text and graphics without creating a stand-alone program, at least not yet.

There is one interesting thing you have to know about QuickDraw before we write our first program that puts text on the graphics screen. When you draw text, QuickDraw can draw it in any of the screen colors. Naturally, that means you need a way to tell QuickDraw what colors to use. For the drawing commands, like LineTo, we use SetSolidPenPat to tell QuickDraw what color to use. For text, there are two colors: that color of the characters themselves, and the color of the background that the characters appear on. There are two separate commands to set these colors, SetBackColor, and SetForeColor. The color numbers are the same as the ones you are used to.

Listing 11.2 gives an example of some of the effects you can get with these colors. Give it a try as an EXE program.

Problem 11.2. Make this a stand-alone program, and run it. The program will return to the Finder too quick to read the screen if you don't put in some sort of pause. Putting a readln right before the call to EndGraph is a good way to handle this situation. That way, the program doesn't stop until you press the return key.

Stand-Alone Text Programs

Other than starting and stopping the text tools, there is no real trick to writing stand-alone text programs. Just as with the graphics programs, you have to turn off debug code and set the file type to S16, instead of EXE. The mechanism used to start the text tools is a bit odd, and involves issues with the toolbox that are beyond the scope of this course. Rather than try to explain what tools are, what the tool locater is, and so forth, just so you can initialize and shut down one tool, I will simply give you text equivalents for StartGraph and EndGraph, called StartText and EndText. For a stand-alone text program, call StartText at the very beginning of your program, and EndText at the end of your program.

Because of the way tools work, you should not call StartText and StartGraph in the same program. For a stand-alone program, you need to pick one screen or the other. For the same reasons, you must not call StartText from a program that is not stand-alone. Debug your program without the calls to StartText and EndText, then add the calls when you switch the file type to S16. You will also need a

```
uses Common, TextToolSet;
```

at the start of your program.

Problem 11.3. Pick some text program you have written in the course, and make it a stand-alone program. The solution to this problem uses hangman, a program you wrote as the solution to problem 6.6. If you like your solution, or if you typed in the one we provided, use that program.

Listing 11.3

```
procedure StartText;

{ Start the text tools for a stand-alone text program.     }

begin {StartText}
TextStartUp;
SetErrGlobals(127,0);
SetErrorDevice(1,3);
SetInGlobals(127,0);
SetInputDevice(1,3);
SetOutGlobals(127,0);
SetOutputDevice(1,3);
end; {StartText}

procedure EndText;

{ Shut down the text tools for a stand-alone text program. }

begin {EndText}
TextShutDown;
end; {EndText}
```

Lesson Eleven
Solutions to Problems

Solution to problem 11.1.

```
{ Do a random walk with 10 random shapes                          }

program RandomWalk;

uses Common, QuickDrawII;

const
   numShapes = 10;              {# of shapes to animate}
   walkLength = 100;            {# of "steps" in the walk}

   maxX = 320;                  {size of the graphics screen}
   maxY = 200;

type
   shapeKind = (triangle, square, star);

   shapeType = record           {information about one shape}
      color: 0..3;
      case kind: shapeKind of
         triangle: (
            tx1,tx2,tx3: integer;
            ty1,ty2,ty3: integer;
            );
         square: (
            sx1,sx2,sx3,sx4: integer;
            sy1,sy2,sy3,sy4: integer;
            );
         star: (
            px1,px2,px3,px4,px5: integer;
            py1,py2,py3,py4,py5: integer;
            );
      end;

var
   i,j: integer;                 {loop variables}
                                 {shapes: current array of shapes}
                                 {oldShapes: shapes in last position}
   shapes,oldShapes: array[1..numShapes] of shapeType;
```

```
procedure InitGraphics;

{ Standard graphics initialization.                          }

var
   r: rect;                  {screen size}

begin {InitGraphics}
StartGraph(320);             {initialize QuickDraw}
SetPenMode(2);               {pen mode = xor}
SetPenSize(1,1);             {use a square pen}
SetSolidPenPat(15);          {paint the screen white}
GetPortRect(r);
PaintRect(r);
SetSolidPenPat(0);           {use a black pen}
end; {InitGraphics}


function RandomValue(max: integer): integer;

{ Return a pseudo-random number in the range 1..max.         }
{                                                            }
{ Parameters:                                                }
{    max - largest number to return                          }

begin {RandomValue}
RandomValue := (RandomInteger mod max) + 1;
end; {RandomValue}


procedure DrawShape(s: shapeType);

{ This subroutine draws one of the shapes on the screen.     }
{                                                            }
{ Parameters:                                                }
{    s - shape to draw                                       }

begin {DrawShape}
SetSolidPenPat(s.color);     {set the pen color for the shape}

case s.kind of

   triangle: begin           {draw a triangle}
      MoveTo(s.tx1,s.ty1);
      LineTo(s.tx2,s.ty2);
      LineTo(s.tx3,s.ty3);
      LineTo(s.tx1,s.ty1);
      end;
```

```
   square: begin              {draw a square}
      MoveTo(s.sx1,s.sy1);
      LineTo(s.sx2,s.sy2);
      LineTo(s.sx4,s.sy4);
      LineTo(s.sx3,s.sy3);
      LineTo(s.sx1,s.sy1);
      end;

   star: begin                {draw a star}
      MoveTo(s.px1,s.py1);
      LineTo(s.px2,s.py2);
      LineTo(s.px3,s.py3);
      LineTo(s.px4,s.py4);
      LineTo(s.px5,s.py5);
      LineTo(s.px1,s.py1);
      end;
   end; {case}
end; {DrawShape}


procedure CreateShape(var s: shapeType);

{ This subroutine creates a shape.  The color and initial      }
{ position of the shape are chosen randomly.  The size of the  }
{ shape is based on pre-computed values.                       }
{                                                              }
{ Parameters:                                                  }
{    s - shape to create                                       }

var
   cx,cy: integer;            {center point for the shape}

begin {CreateShape}
s.color := RandomValue(15);  {get a color}
cx := RandomValue(300)+10;   {get the center position, picking the}
cy := RandomValue(184)+8;    {points so the shape is in the window}

case RandomValue(3) of       {set the initial positions}
   1: begin                  {set up a triangle}
      s.kind := triangle;
      s.tx1 := cx-10;
      s.ty1 := cy+4;
      s.tx2 := cx;
      s.ty2 := cy-8;
      s.tx3 := cx+10;
      s.ty3 := cy+4;
      end;
```

```
   2: begin                  {set up a square}
      s.kind := square;
      s.sx1 := cx-8;
      s.sy1 := cy-6;
      s.sx2 := cx+8;
      s.sy2 := cy-6;
      s.sx3 := cx-8;
      s.sy3 := cy+6;
      s.sx4 := cx+8;
      s.sy4 := cy+6;
      end;

   3: begin                  {set up a star}
      s.kind := star;
      s.px1 := cx-7;
      s.py1 := cy+7;
      s.px2 := cx;
      s.py2 := cy-8;
      s.px3 := cx+7;
      s.py3 := cy+7;
      s.px4 := cx-11;
      s.py4 := cy-3;
      s.px5 := cx+11;
      s.py5 := cy-3;
      end;
   end; {case}
end; {CreateShape}


procedure UpdateShape(var s: shapeType);

{ This subroutine moves a shape across the screen in a random  }
{ walk.                                                        }
{                                                              }
{ Parameters:                                                  }
{    s - shape to update                                       }

var
   dx,dy: integer;           {movement direction}

begin {UpdateShape}
dx := RandomValue(3)-2;     {get the walk direction}
dy := RandomValue(3)-2;
```

```
case s.kind of              {make sure we don't walk off of the}
                             { screen, then update the position }
   triangle: begin           {check a triangle}
      if dx = -1 then
         if s.tx1 < 1 then
            dx := 0;
      if dx = 1 then
         if s.tx3 >= maxX then
            dx := 0;
      if dy = -1 then
         if s.ty2 < 1 then
            dy := 0;
      if dy = 1 then
         if s.ty3 >= maxY then
            dy := 0;
      s.tx1 := s.tx1+dx;     {update a triangle}
      s.ty1 := s.ty1+dy;
      s.tx2 := s.tx2+dx;
      s.ty2 := s.ty2+dy;
      s.tx3 := s.tx3+dx;
      s.ty3 := s.ty3+dy;
      end;

   square: begin             {check a square}
      if dx = -1 then
         if s.sx1 < 1 then
            dx := 0;
      if dx = 1 then
         if s.sx2 >= maxX then
            dx := 0;
      if dy = -1 then
         if s.sy1 < 1 then
            dy := 0;
      if dy = 1 then
         if s.sy3 >= maxY then
            dy := 0;
      s.sx1 := s.sx1+dx;     {update a square}
      s.sy1 := s.sy1+dy;
      s.sx2 := s.sx2+dx;
      s.sy2 := s.sy2+dy;
      s.sx3 := s.sx3+dx;
      s.sy3 := s.sy3+dy;
      s.sx4 := s.sx4+dx;
      s.sy4 := s.sy4+dy;
      end;
```

```
   star: begin               {check a star}
      if dx = -1 then
         if s.px4 < 1 then
            dx := 0;
      if dx = 1 then
         if s.px5 >= maxX then
            dx := 0;
      if dy = -1 then
         if s.py2 < 1 then
            dy := 0;
      if dy = 1 then
         if s.py1 >= maxY then
            dy := 0;
      s.px1 := s.px1+dx;     {update a star}
      s.py1 := s.py1+dy;
      s.px2 := s.px2+dx;
      s.py2 := s.py2+dy;
      s.px3 := s.px3+dx;
      s.py3 := s.py3+dy;
      s.px4 := s.px4+dx;
      s.py4 := s.py4+dy;
      s.px5 := s.px5+dx;
      s.py5 := s.py5+dy;
      end;
   end; {case}
end; {UpdateShape}
```

```
begin
InitGraphics;                    {set up the graphics window}
Seed(6289);                      {initialize the random number generator}

for i := 1 to numShapes do       {set up and draw the initial shapes}
   begin
   CreateShape(shapes[i]);
   DrawShape(shapes[i]);
   end; {for}

for i := 1 to walkLength do      {do the random walk}
   begin
   oldShapes := shapes;          {move the shapes}
   for j := 1 to numShapes do
      UpdateShape(shapes[j]);
   for j := 1 to numShapes do    {redraw the shapes}
      begin
      DrawShape(shapes[j]);
      DrawShape(oldShapes[j]);
      end; {for}
   end; {for}
EndGraph;                        {shut down QuickDraw}
end.
```

Solution to problem 11.2.

```
{ Exploring text mixed with graphics. }

program TextAndGraphics(input,output);

uses Common, QuickDrawII;

   procedure InitGraphics;

   { Standard graphics initialization.                          }

   var
      r: rect;                   {screen size}

   begin {InitGraphics}
   StartGraph(640);              {initialize QuickDraw}
   SetPenMode(0);                {pen mode = copy}
   SetPenSize(3,1);              {use a square pen}
   SetSolidPenPat(3);            {paint the screen white}
   GetPortRect(r);
   PaintRect(r);
   SetSolidPenPat(0);            {use a black pen}
   end; {InitGraphics}
```

```
begin
InitGraphics;                   {set up the graphics screen}
SetForeColor(0);                {write some text}
SetBackColor(3);
MoveTo(10,10);
writeln('You can get');
write  ('some ');
SetForeColor(3);
SetBackColor(0);
writeln('interesting effects');
SetForeColor(2);
SetBackColor(3);
writeln('by mixing text');
SetForeColor(1);
write  ('with graphics!');
readln;
EndGraph;                       {shut down QuickDraw}
end.
```

Solution to problem 11.3.

```
{ Hangman                                                 }
{                                                         }
{ This program plays the game of Hangman.  When the       }
{ game starts, you are given a word to guess.  The        }
{ program displays one dash for each letter in the        }
{ word.  You guess a letter.  If the letter is in the     }
{ word, the computer prints the word with all letters     }
{ you have guessed correctly shown in their correct       }
{ positions.  If you do not guess the word, you move      }
{ one step closer to being hung.  After six wrong         }
{ guesses, you loose.                                     }

program HangMan (input, output);

uses Common, TextToolSet;

const
   maxWords = 10;               {possible words}
   maxChars = 8;                {number of characters in each word}

var
   words: array[1..maxWords] of string[maxChars]; {word array}
```

```
procedure StartText;

{ Start the text tools for a stand-alone text program.     }

begin {StartText}
TextStartUp;
SetErrGlobals(127,0);
SetErrorDevice(1,3);
SetInGlobals(127,0);
SetInputDevice(1,3);
SetOutGlobals(127,0);
SetOutputDevice(1,3);
end; {StartText}

procedure EndText;

{ Shut down the text tools for a stand-alone text program. }

begin {EndText}
TextShutDown;
end; {EndText}


procedure FillArray;

{ Fill the word array.                                    }
{                                                         }
{ Variables:                                              }
{    words - word array                                   }

begin {FillArray}
words[1] := 'computer';
words[2] := 'whale';
words[3] := 'megabyte';
words[4] := 'modem';
words[5] := 'chip';
words[6] := 'online';
words[7] := 'disk';
words[8] := 'monitor';
words[9] := 'window';
words[10] := 'keyboard';
end; {FillArray}
```

```
procedure GetSeed;

{ Initialize the random number generator             }

var
   val: integer;             {seed value}

begin {GetSeed}
write('Please enter a random number seed:');
readln(val);
seed(val);
end; {GetSeed}


function RandomValue(max: integer): integer;

{ Return a pseudo-random number in the range 1..max. }
{                                                    }
{ Parameters:                                        }
{    max - largest number to return                  }
{    color - interior color of the rectangle         }

begin {RandomValue}
RandomValue := (RandomInteger mod max) + 1;
end; {RandomValue}


procedure Play;

{ Play a game of hangman.                            }
{                                                    }
{ Variables:                                         }
{    words - word array                              }

var
   allFound: boolean;        {used to test for unknown chars}
   ch: char;                 {character from player}
   done: boolean;            {is the game over?}
                             {characters found by the player}
   found: array[1..maxChars] of boolean;
   len: integer;             {length of word; for efficiency}
   i: integer;               {loop variable}
   inString: boolean;        {is ch in the string?}
   word: string[maxChars];   {word to guess}
   wrong: integer;           {number of wrong guesses}
```

```
begin {Play}
                              {pick a word}
word := words[RandomValue(maxWords)];
len := length(word);          {record the length of the word}
for i := 1 to len do          {no letters guessed, so far}
   found[i] := false;
done := false;                {the game is not over, yet}
wrong := 0;                   {no wrong guesses, yet}
ch := ' ';                    {initialize the character}
repeat
   writeln;                   {write the word}
   write('The word is: "');
   for i := 1 to len do
      if found[i] then
         write(word[i])
      else
         write('-');
   writeln('"');
   write('Guess a character:'); {get the player's choice}
   readln(ch);
   inString := false;         {see if ch is in the string}
   for i := 1 to len do
      if word[i] = ch then begin
         found[i] := true;
         inString := true;
         end; {if}
   if inString then           {handle a correct guess}
      writeln(ch, ' is in the string.')

   else begin                 {handle an incorrect guess}
      writeln(ch, ' is not in the string.');
      wrong := wrong+1;       {one more wrong answer...}
      write('Your ');         {tell the player how they are doing}
      if wrong = 1 then
         write('head')
      else if wrong = 2 then
         write('body')
      else if wrong = 3 then
         write('left arm')
      else if wrong = 4 then
         write('right arm')
      else if wrong = 5 then
         write('left leg')
      else {if wrong = 6 then}
         write('right leg');
      writeln(' is now in the noose!');
      end; {else}
```

```
      if wrong = 6 then begin   {see if the player is hung}
         writeln('Sorry, Jack Ketch got you!');
         writeln('The word was ', word);
         done := true;
         end; {if}
      allFound := true;         {check for unknown characters}
      for i := 1 to len do
         if not found[i] then
            allFound := false;
      if allFound then begin    {see if the player got the word}
         writeln('You got it!  The word is ', word);
         done := true;
         end; {if}
   until done;
   end; {Play}


   function PlayAgain: boolean;

   { See if the player wants to play another game.      }
   {                                                    }
   { Returns:                                           }
   {    True to play again, false to quit.              }

   var
      ch: char;                 {player's response}

   begin {PlayAgain}
   ch := ' ';
   writeln;
   writeln;
   repeat
      write('Would you like to play again (y or n)?');
      readln(ch);
   until (ch = 'y') or (ch = 'Y') or (ch = 'n') or (ch = 'N');
   PlayAgain := (ch = 'y') or (ch = 'Y');
   end; {PlayAgain}


begin
StartText;                      {start the text tools}
FillArray;                      {fill the word array}
GetSeed;                        {initialize the random number generator}
repeat
   Play;                        {play a game}
until not PlayAgain;            {loop if he wants to play again}
EndText;                        {shut down the text tools}
end.
```

Lesson Twelve
A Project: Developing a Break-Out Game

Designing a Program

There is a basic difference between how you write a large program, and how you write a small one. So far, every program you have written in this course has been a small one. I know, some of them have seemed large, but they are really small compared to what experienced programmers can write, even in a single week. Because the programs are so small, it is quite possible that you can keep all of the details about the program in your head at one time. That makes it hard to understand why I stress commenting, breaking programs down into pieces with subroutines, or even a consistent indenting style. For programs under 500 lines or so, these issues are rarely important. I can tell you from experience, though, that structured programming is crucial when you set out to write a 10,000 line program.

This lesson exists for one simple reason: to give you at least one hands-on look at how a real program is developed. While the program we will write in this lesson is not large, it is enormous compared to anything you have written so far. Writing this program gives you a chance to see, first hand, how the techniques of structured programming and good style can help to develop a program. You will also see how a program develops iteratively, and how to use the techniques of top-down and bottom-up design when writing a program.

There is only one problem in this lesson. The problem is to write a game. The text of the lesson concentrates more on the thought process that you need to go through to develop the program than on the details of coding. As you go, though, you should be writing the game. A complete listing is, of course, given in the solution to the lesson.

The User: That's Who We Write For

Absolutely the first step, and the one most often neglected by programmers, engineers, and virtually every other member of a skilled profession, is to step back and realize why we are writing a program in the first place. The program is being written for someone to use. Whether we are writing an arcade game, creating the code to run a pacemaker, or simulating the rise and fall of the Roman Empire, the program exists to serve the need of some person or group of people. The first step in designing a program is to decide who those people are, and what they really want in a program. Before you decide on your first algorithm, before you write your first line of code, you need to decide what the program does, who the program does it for, and what they want the program to do.

Our program is a simple arcade game called Break Out. It's been around since the dawn of time – in computer terms, that was about 1970 or so. When the game starts, there are several rows of bricks along the top of the screen. Along the bottom is a paddle; it shows up as a line about an inch wide. A ball drops from the vicinity of the bricks; the object is to move the mouse to hit the ball, sending it back up to the bricks. Each time the ball hits a brick, the brick goes away, and the payer's score goes up. If all of the bricks get knocked out, a new set of bricks appear, one row closer to the bottom of the screen. If the player misses a ball, it vanishes, and a new ball drops from the screen. We will start the player with three balls, and add one more each time all of the bricks are knocked off of the screen.

This may seem like a pretty simple game, and it is. It can also be very addicting. I have spent hours playing Break Out when there were other things to do. Even my kids enjoy it – when I give them a chance to play!

The first step in designing the program is to make sure you can visualize the screen, and what will happen at each step of the program. For a graphics program like this one, a simple, rough sketch is a great aid. Grab a pencil and paper, and draw a large rectangle to represent the screen. Along the top, draw six rows of bricks. These should be in the top quarter of the screen, and there should be a gap above the bricks.

There are two things a player will want to keep track of while the game is being played: the current score, and the number of balls he has left. We'll put these along the bottom of the screen, in the left and right corner. Right above this text information is where the paddle will be. It is a simple line.

While the game is being played, the only action the player can take is to move the mouse back and forth, which in turn moves the paddle back and forth on the screen. Sketch the paddle just above the text that gives the score and number of balls.

The program itself, of course, is doing a bit more. In addition to tracking the progress of the mouse, the program is moving the ball across the screen. There are several things that can happen to the ball. Outlining

these cases clearly now is pretty easy, especially if you imagine the ball bouncing around on your sketch. If you look at the outline, below, of what the ball does, you can probably visualize what the code to handle the ball will look like, too, in terms of the if statements that will be needed. Here's a table that describes the various things that can happen as the ball moves around the screen.

1. If the ball hits the paddle, it will rebound toward the top of the screen. The paddle is restricted to a row along the bottom of the screen, so we can check to see if the ball has hit the paddle by checking the Y coordinate (the Y coordinate is the vertical position) of the ball. If it matches the Y coordinate for the paddle, we can check to see if the ball has hit the paddle by comparing the X coordinate (the horizontal position) for the ball with the left and right edges of the paddle.

   Something that makes the game a lot more interesting is to add some spin to the paddle. If the ball hits the center of the paddle, we will bounce it straight back up. If the ball hits a little to one side, we can send it off at a small angle. If the ball hits near the edge of the paddle, we send it back up at a steep angle. In all cases, we send the ball back up. The thing we are changing is the velocity of the ball in the y direction. You can probably visualize how this will work on paper; later we will work through the details of how to make it happen in the computer.

2. The next case is if the ball reaches the paddle row, but does not hit the paddle. In other words, the player missed. In that case, the ball vanishes, and we go back to the starting point.

3. The ball could hit the left, right, or top of the screen. In that case, we bounce the ball back toward the middle. We actually wrote a sample program to do this once, a long time ago.

4. The ball could hit a brick. In that case, we do several things. First, we erase the brick. Second, we update the score, adding some points for eliminating the brick. Finally, the ball bounces back.

   An important point to remember here is that the ball can hit a brick from the bottom, the top, or even from the side. For example, if the player pokes a hole in the bricks, the ball can whiz through at an angle, and start bouncing off of the top of the screen and the top of the highest row of bricks. If the ball doesn't make it all the way through the hole, it could hit a brick on the side. We need to make sure that our ball can bounce in an appropriate direction, regardless of which side of the brick it hits.

Now that we know how the action part of the game is played, we need to back up and figure out how to start the game. While a game is being played, the player will eventually miss a ball. When that happens, it would be nice to give the guy a chance to catch his breath. One way to do this is to print a message on the screen and wait for the player to press the mouse button. We'll try that first, and see how it works in our program.

When the game starts, and after each game is played, we need to print some sort of message. They player will need two options: playing a game or quitting. One way to handle this is to write a couple of text messages on the screen, with lines around the messages to make them look like buttons. We will let the person move the same arrow cursor that you are used to around the screen. When the player presses the mouse button, the program will check to see if the arrow is inside one of our boxes; if so, the program will either quit or start a game, depending on which box the arrow is in.

You may notice that this planning process has left out a lot of details. For example, we haven't decided what all of the text messages will be. You don't know how to move the mouse, or how to make an arrow cursor move around the screen. We haven't decided exactly how big the bricks will be, or where they will be placed.

Some of these details are important at this point, and some are best left until later. How to deal with the mouse is important: how it is done will shape the design of the program. That means we need to do a little research, and possibly develop a few subroutines before we really start the program. What text we write on the screen, how big the bricks and paddle are, and exactly where they go doesn't matter. You can jot down some ideas now, or just wait until you are working on that part of the program. The exact size and position of the bricks is sure to be something we change as the program develops. We'll try several possibilities, and pick the one that works best. In short, the thing to be sure of right now is that you know how to do all of the things, like moving the mouse, that you will need to

do. The details can, and in some cases must, be left until the program starts to take shape.

Laying the Groundwork

The first step in developing the program is to learn how to do the things we don't already know how to do. The obvious thing, in this case, is to learn to read and handle the mouse. This is also the stage of development where you might hit the books. For example, if you are writing an astronomy program, this is the time to dig through books to find the appropriate formulas. It's a good time to find out what star data bases are available, too, and what format they come in.

For our program we need to know how Apple IIGS programs deal with the mouse, and how we go about using those abilities. The way this is done is tied up in the concept of an event loop.

To understand what an event loop is, and what it has to do with a mouse, let's start by thinking about how our program might work. Basically, while the game is running, the program needs to do two things: move the ball, handling anything that might happen if the ball hits something, and move the paddle as the player moves the mouse. The way we do this is to loop over basic calls to subroutines until the ball missed the paddle, something like this:

```
repeat
   MoveBall;
   MovePaddle;
until balls = 0;
```

In the language of the Apple IIGS, this is very, very close to being an even loop. The idea of an event loop is to loop, waiting for something to happen. We start the loop with a call to the event manager to get the next event. An event is basically something the player did. It could be pressing a mouse button, pressing a key, or clicking in a menu bar. It can also be a null event, which is a fancy way of saying nothing happened. No matter what kind of event has occurred, though, the event manager fills in a record and passes some information back to us. The important part of that information, from our standpoint, is the current position of the mouse. In a nutshell, that's how we read the mouse. It only takes a tiny change to turn the main loop we just wrote into an Apple IIGS event loop that reads the location of the mouse for us each time through the loop.

```
repeat
   event :=
      GetNextEvent(eventMask, myevent);
   MoveBall;
   MovePaddle;
until balls = 0;
```

The event manager returns a boolean flag that tells us whether an event occurred. We don't really care: in our game, the player can click on the button, pound on the keyboard, or whatever, and the program will ignore him. The only thing that is important to us is where the mouse is. The position of the mouse is returned in myevent, which is a record with a type of eventRecord. Like most records used by the Apple IIGS toolbox, eventRecord is defined in the interface file Common.Intf, which you include in your programs with the uses statement. When you are learning about a new record, it is a good idea to actually look at the source code for the toolbox interface files to see how the record is defined. Here is the definition for an event record:

```
eventRecord = record
    eventWhat:      integer;
    eventMessage:   longint;
    eventWhen:      longint;
    eventWhere:     point;
    eventModifiers: integer;
    taskData:       longint;
    taskMask:       longint;
    end;
```

Most of these fields are of no interest to us at the moment, although we will use a couple more later. The eventWhat field is filled in with a number that indicates what kind of an event occurred. There is one number for a key press, another for pressing on the mouse button, still another for letting up on the mouse button, and so on. We will use this field later, when we try to decide if the player has clicked on a mouse button.

The meaning of the eventMessage field varies, depending on what kind of event occurs. For a key press, for example, this field tells what key was pressed.

The eventWhen field is a primitive timer. The Apple IIGS keeps track of how many 1/60ths of a second have elapsed since the event manager was started; the value is returned in this field. This is a good way to decide if a certain amount of time has passed.

The field we are really interested in at the moment is the eventWhere field. This field is actually a record itself. The record is a point, something you saw briefly in lesson 10. The event manager fills in this field with

the current position of the mouse. As you move the mouse across your desktop, the Apple IIGS keeps track of it, changing the position. The position is reported as a point on the screen; it ranges from 0 to 640 horizontally, and 0 to 200 vertically. If you start the event manager in the 320 graphics mode, the position for the mouse is adjusted so that the horizontal position varies from 0 to 320. In other words, a lot of work is being done to keep things simple for you.

The eventModifiers field contains still more information about the particular event that the event manager is reporting. The taskData and taskMask fields aren't filled in by a GetNextEvent call at all; they are in the record for a different kind of call, called TaskMaster. If you learn to write desktop programs, you will get very familiar with TaskMaster, but we will ignore these fields for now.

There is one minor complication that we will have to deal with. We are used to writing and debugging graphics programs in the graphics window. The event manager returns the position of the mouse on the screen, not it's position within our window. Fortunately, there is a simple call called GlobalToLocal that will change the values of a point so they are given in relation to the current window, rather than the screen as a whole.

Putting all of this together, the following sample program moves a paddle back and forth in the graphics window. Naturally, we have to start the event manager. This program gives an adaptation of the graphics startup code from the last lesson that also starts the event manager. It uses many ideas you have seen before in a new way: animating a shape, finding the size of the graphics window, and so forth. You know what all of the pieces are, but stop and type in the program in listing 12.1 to make sure you know how they are used in this program.

Listing 12.1

```
{ Move a paddle in the graphics window }

program Paddle;

uses Common, QuickDrawII, EventMgr, MemoryMgr;

const
   eventMask = $0F6E;           {GetNextEvent event mask}

var
   event: boolean;              {event flag; returned by GetNextEvent}
   maxX: integer;               {max X distance the paddle can travel}
   myEvent: eventRecord;        {current event record}
   paddlePosition: integer;     {current X position of the paddle}
   screen: rect;                {port rectangle}


   procedure StartTools;

   { Start the tools                                              }

   const
      size = 640;               {graphics mode}

   var
      memory: handle;           {memory returned by NewHandle}
      r: rect;                  {screen size}

   begin {StartTools}
   StartGraph(size);            {initialize QuickDraw}
```

```
SetPenMode(2);                {pen mode = xor}
SetPenSize(1,1);              {use a square pen}
SetSolidPenPat(15);           {paint the screen white}
GetPortRect(r);
PaintRect(r);
SetSolidPenPat(0);            {use a black pen}
memory := NewHandle(256,UserID,$C015,nil); {start up the event mgr}
EMStartUp(ord(memory^),0,0,size,0,200,UserID);
end; {StartTools}


procedure ShutDownTools;

{ Shut down the tools                                          }

begin {ShutDownTools}
EMShutDown;
EndGraph;
end; {ShutDownTools}


procedure DrawPaddle (position,color: integer);

{ Draw the paddle                                              }
{                                                              }
{ Parameters:                                                  }
{    position - position to draw the paddle                    }
{    color - color of the paddle                               }

const
   width = 70;                {width of the paddle}
   height = 3;                {height of the paddle}

var
   y: integer;                {position of paddle on screen}

begin {DrawPaddle}
SetPenSize(width,height);     {set the pen to draw the entire paddle}
SetSolidPenPat(color);        {set the paddle color}
SetPenMode(0);                {use copy mode}
if position+width > maxX then {make sure we don't go off of the screen}
   position := maxX-width;
if position < 0 then
   position := 0;
y := screen.v2-12;            {find the paddle's y position}
MoveTo(position,y);           {draw the paddle}
LineTo(position,y);
end; {DrawPaddle}
```

```
    procedure MovePaddle;

    { Track and move the paddle                                    }
    {                                                              }
    { Variables:                                                   }

    begin {MovePaddle}
    {convert the point to our window}
    GlobalToLocal(myevent.eventWhere);
    {if the mouse moved, move the paddle}
    if myevent.eventWhere.h <> paddlePosition then begin
       DrawPaddle(paddlePosition,3);
       paddlePosition := myevent.eventWhere.h;
       DrawPaddle(paddlePosition,0);
       end; {if}
    end; {MovePaddle}


begin
StartTools;                    {start the tools}
GetPortRect(screen);           {set the limit on the paddle}
maxX := screen.h2;
DrawPaddle(0,0);               {draw the initial paddle}
paddlePosition := 0;

repeat                         {event loop}
   event := GetNextEvent(eventMask, myevent);
   MovePaddle;
until event;

ShutDownTools;                 {shut down the tools}
end.
```

There are a couple of new things in this program that deserve special attention. First off, the hardest thing about initializing a tool by making direct calls to the Apple IIGS toolbox is getting the direct page memory so many of the tools need. That is one of the main reasons that ORCA/Pascal has built-in calls to help you initialize the tools. In this program, we need to start the event manager, but we don't want to start any of the other desktop tools that StartDesk would initialize. While we won't go into details about the tools in this course, the example in this program showing how to start the event manager can be used as a model for starting other tools. If you like, you can look up the calls used in the Apple IIGS Technical Reference Manual to see what was done and why. For the purposes of this course, though, we'll treat the code to start the event manager as a black box: something you can use without understanding it in detail.

There is a new animation trick in this program that I want you to notice, too. The MovePaddle procedure has a check to see if the mouse has moved. If the mouse has not moved, the paddle is not redrawn. There is a very good reason to make this check. Without it, if you leave the mouse in one place, the paddle will flicker slightly as it is continuously erased and redrawn. You don't notice this flicker as much when the paddle is moving, but it is very annoying when the paddle is standing still.

Finally, we took a cheap way of exiting the loop. GetNextEvent returns a boolean value that tells us if some event has taken place. Moving the mouse isn't considered to be an event, so we can move the mouse (and hence the paddle) back and forth to test the program. As soon as we click the mouse button or press a key, though, the event manager reports the event, and we drop out of the repeat loop. Later, our

checks will get more sophisticated, but this easy mechanism lets us work on parts of the program without putting a lot of effort into things that will change. This basic idea of simplifying tasks and leaving work for a later time is a very powerful technique. It allows you to concentrate on a manageable sized part of the program.

Bottom-Up Design Verses Top-Down Design

If you stop and think about it for a moment, we just wrote a program to move a paddle across the screen. "Of course we did," you say. "We are writing a break-out game." True, but there was a method to our madness. After all, we were really trying to learn how the mouse was used. On the other hand, we know we will need to move a paddle across the screen for our game, so we wrote the paddle movement subroutines. As it turns out, we can use these same subroutines, with a few changes in the constants that control the size and position of the paddle, in our game. Programmers have a name for this: it is called bottom-up design.

The idea behind bottom-up design is to look at a program, or any problem, and break it down into small parts. We then do the parts individually, and assemble the finished parts to get a complete program. When the parts are well-defined, like moving a paddle, this technique works very well. We can write and test good sized chunks of the program in small test programs, finding problems and making sure the individual pieces work the way we want them to. That way, when we put the piece in the program, it is probably going to work fine. And, since we used a small test program to develop it, we don't have to recompile the entire program each time we make a small change to our piece. Even more important, if there is a bug, there is a lot less code to search through to find the problem.

If you have been around programming circles, though, you have probably heard about top-down design. This is just the opposite of bottom-up design: instead of breaking the problem down into pieces, and working on the individual pieces, we organize the problem, and write the main part of the program. Any pieces are put in the program as empty subroutines, or maybe as a subroutine that just has a message telling that it was called. These dummy subroutines are called stubs. We then write and test the pieces, one after the other, until the program is finished. The process of writing the pieces is called stepwise refinement.

So which is better? Actually, that's the wrong question. It's like asking if a hammer or a saw is a better tool. The answer depends on what you are doing. In most large programs, we use both methods. In fact, we have already used both in this lesson. We started with a top-down design, laying out the basic goals of the program. We didn't get far before we discovered that there were some things – namely, reading the mouse – that we needed to learn how to do. We researched this problem, and in the process wrote some subroutines that we will need in our finished game. Next, we will return to the top-down design method, writing the shell of the program and gradually refining it.

Here are some rules of thumb to help you choose between the two methods. Like all rules of thumb, an experienced programmer will be able to point out exceptions. These are guidelines, not hard and fast rules.

1. Always start your design process from the viewpoint of the user. The only way to effectively do this is to use the top-down design method.

2. While you are doing the initial program design, look for general themes that apply to many problems of the kind you are about to solve. These are often good targets for bottom-up design. For example, if you are about to write a program to manipulate matrices, you could develop a matrix inversion subroutine that can be used in your program before starting on the main part of the program. If you will be writing an arcade game, it might be wise to develop the animation routines before you start.

3. Once any low-level subroutines are developed, return to the top-down design approach. Write the main program with stubs. Gradually fill in the stubs until the program is finished, adding your low-level routines developed in step 2 as needed.

4. Always skip step 2 if you can. Unless there is a clear reason for developing a subroutine before you start on the main program, it is probably a good idea to implement it as a stub, and fill it in later. The reason is simple: it is very easy to miss a detail in your initial design pass that could be very important when the subroutine is written. If you develop the program from the top down, these kinds of details are obvious by the time you get to the point where you are writing the subroutine.

Starting the Program

Now that we have finished the basic design, researched the things we didn't know about, and written the low-level routines that we wanted to write, it is time to start on the program itself. The program that we used to test the paddle movements is actually a very good place to start. After all, the paddle does move in the window already!

The first step, then, is to flesh out the sample, putting in stubs for the various subroutines we will need later. One stub will be the subroutine that writes a message on the screen, and gives the player a chance to quit or play a game. We can call this stub PlayAGame, and make it a boolean function. This stub will handle all of the details of putting the message on the screen and figuring out if the player wants to play a game or quit. If the player wants to quit, the function will return false; otherwise it will return true. To handle this stub, we will add a while loop around the game's event loop, like this:

```
while PlayAGame do
  repeat                          {event loop}
     event :=
        GetNextEvent(eventMask, myevent);
     MovePaddle;
    until event;
```

This brings up an interesting point, though: does our stub return true or false? It needs to return true so we can play a game. We need a way for it to return false, though, so we can get out of our program. We will solve this problem with some simple "throw-away" code. This is code that we know doesn't work like we want the final program to work, but does the job well enough that we can concentrate on other parts of the program for a while. For this stub, add a global variable, and initialize it to true. PlayAGame should return the value of the global variable, but set it to false. That way, PlayAGame returns true the first time it is called, but false the second time.

It's embarrassing to finish a program and have dead variables around. When I add dummy variables or dummy code to a program, I always mark it with a comment that looks like this:

```
{<<<>>>}
```

This makes it easy to go back later and search for dummy code. That way, with one search, I can make sure that there is no dummy code, or any unused variables left in the program.

In the event loop we need to move the ball, and handle the various situations that pop up when the ball hits something. Add another stub called MoveBall to your event loop. We will need some way of stopping the game before it is finished. MoveBall is where the check will eventually be done, so we will put the dummy check in MoveBall now. We know this is where the check will eventually be, since MoveBall is where we will test to see if we missed the paddle. If so, and if there are no more balls left, the game is over. For now, we will test the last event returned by GetNextEvent to see if the mouse button was released, like this:

```
if myEvent.eventWhat = mouseUpEvt then
   balls := 0;
```

This is the first time we have checked the mouse. There are two basic mouse events, mouseDownEvt and mouseUpEvt. The first is when the mouse button is pressed down, while the second is when the mouse button is let up. Generally, we wait until the mouse is released before performing the action, so this code waits for a mouse up event. Don't forget to change the exit condition for the event loop at the same time! In the paddle program, the event loop stops when an event occurs. In our game, it should stop when balls = 0.

We need to draw the initial screen when we start. Add one last stub right after the call to PlayAGame, called InitScreen. InitScreen should start by filling the entire screen with a black background. Naturally, you should go into the startup code for the tools and remove the lines that fill the initial, black screen with a white background! Finally, move the initial code that draws the paddle and sets paddlePosition into this subroutine.

One of the things you are used to seeing in a desktop program is an arrow that moves across the screen when the mouse moves. This arrow is something you have to initialize. Right after the tools are started, you should put a call to InitCursor in your program.

```
InitCursor;
```

This can be done anytime after QuickDraw is started. It is pretty annoying, though, to have an arrow on the screen while we are moving a paddle. You may want to leave the arrow on the screen for now, to make it easier to use the debugger, but eventually you will want to add a call to HideCursor, right after the InitCursor call. HideCursor, which also has no parameters, makes the arrow invisible. When we add code to the program that allows the player to click on a mouse button, you will want to make the arrow show up again. ShowCursor

does this. Like the other two cursor calls, ShowCursor does not have any parameters.

Before moving on, there is one more topic we need to deal with. The finished program will be a stand-alone graphics program, of course. In lesson 11, though, you found out that it is best to work within the programming environment as long as you can. How can we test the program without leaving the programming environment? The answer involves some tool calls that let you manipulate the size and location of a window. It turns out that the graphics window is the current port while your program is running under PRIZM. Using this fact, the subroutine ExpandGraph, shown in listing 12.2, expands the graphics window to the full size of the screen, and bring it to the front, so it is drawn over all of the other windows. ShrinkGraph returns the graphics window to a reasonable size. If you are using the debugger while your program runs, you may want to leave out the call to BringToFront. That way, the source code window will remain visible while you play the game. The bricks, ball and paddle will move behind the source code window, making it a bit hard to play the game, but at least you will be able to see the source code. Also, when you return from the program, you will need to click on the program window to make it the front window. The menu bar will remind you to do this, since many menu items are not available when the graphics window is the front window.

Add these to your program as throw-away code. These subroutines use tool calls defined in the window manager interface file, so you will need to add WindowMgr to the end of your uses statement. Stop now, and get your entire program to work with the stubs.

Drawing the Bricks

The next step is to draw the initial bricks. There are several things that we need to do to handle the bricks. Up until now, we have ignored the details, but at this point it is time to stop and think through the issues carefully. Putting a word to it, this is stepwise refinement. What we are doing is to design a mini-program just like we would design a complete program. The mini-program we will write in this section draws a row of bricks on the screen, and initializes some tables for later use by other parts of the program.

The program starts with six rows of bricks on the screen. We will use a total of 16 bricks in each row. There are four things we will need to know about these bricks: their color, where they are, how many points they are worth, and whether they have been knocked out yet.

Let's start by deciding where the bricks will be.

Listing 12.2.

```
procedure ExpandGraph;

{ Expand the graphics window to full screen                    }

begin {ExpandGraph}
{<<<>>>}
MoveWindow(0,0,GetPort);     {move the window to the top-left corner}
SetMaxGrow(640,200,GetPort); {let the window get this big}
SizeWindow(640,200,GetPort); {make the window this big}
BringToFront(GetPort);       {bring the graph window to front}
end; {ExpandGraph}


procedure ShrinkGraph;

{ Put the graphics window back in the corner                   }

begin {ShrinkGraph}
{<<<>>>}
MoveWindow(320,115,GetPort); {move graph back to the corner}
SizeWindow(320,85,GetPort);  {make the window a reasonable size}
end; {ShrinkGraph}
```

The first step is to decide how big each brick will be. The screen is 640 pixels wide, so if we want 16 bricks per row, each one will be 40 pixels wide. The six rows of bricks should appear near the top of the screen, say in the top 60 or so pixels. We want some blank space at the top to give the ball some room to rebound, too. We'll allow 8 pixels before the top of the first brick. If we leave two blank lines between each row of bricks, and make each brick six pixels high, we end up using 48 rows of pixels. Along with the 8 blank pixels, this gives us 56 pixels at the top of the screen, which is about right.

Stop now and write the procedure to draw the original set of six rows of bricks on the screen. For consistency with the solution, call the procedure DrawBricks; it should be called from InitScreen. I used a global variable called brickY to decide how high the bricks are. That way, as the game progresses and the bricks get lower, I only had to change one global value before calling DrawBricks. Of course, you need to choose some colors for the bricks. In my program, I made all of the bricks in a single row the same color, and used a black line along the right edge of each brick to separate the bricks.

Assigning the values to the bricks is somewhat a matter of personal taste, of course. All of the bricks in a particular row should have the same value. The bricks in the bottom row should be worth the least, and the bricks in the top row should be worth the most. If the player is skillful enough to move on to the next level of play by knocking out all of the bricks, the rows move down. Since it is harder to knock out the bricks at the lower level, they should be worth more, too. There are several ways to handle all of these factors. You could create an array that holds the point values for each row, with different point values for each level, like this:

```
const
   rows = 6;
   levels = 16;

var
   points: array[1..rows,1..levels]
      of integer;
```

I picked 16 as the maximum number of levels here; we may need to vary that later. This scheme gives you a lot of flexibility when assigning point values to the bricks, but it is pretty complicated, and it will take a lot of code to fill in the array. I finally decided on a simpler mechanism. On level 1, the bottom row of bricks is worth 10 points, and each higher row is worth 5 more points. On level 2, the bottom row is worth 15 points, while each higher row is still worth 5 additional points. This mechanism is so simple that I can use a formula to find the value of a brick, assuming I know the row and playing level. The formula I came up with is

```
   points := (row+level)*5;
```

This is so simple that I don't need to initialize anything in DrawBricks; I just jot the formula down for later use.

The last step is to initialize an array that keeps track of whether a brick has been knocked out or not. This array will be used when we are tracking the ball. When the ball is in the area where the bricks are located, we will calculate the row and column number of the brick it is starting to hit. The row and column number can then be used to index into the brick array. If the brick exists, we know that we have to remove it, and rebound the ball. DrawBricks is responsible for drawing the initial rows of bricks on the screen, so it seems like a good place to initialize the array to true. The array looks like this:

```
const
   rows = 6;
   columns = 16;

var
   stillThere: array[1..rows,1..columns]
      of boolean;
```

Put the code into DrawBricks to initialize all of the elements of this array to true, then test your program to make sure everything works.

A lot of the decisions in this lesson probably seemed arbitrary, and you are wondering how I made them. How do I know, for example, that we want 6 rows of bricks, with 16 bricks in each row? How do I know that the bricks should be 6 pixels thick, with two blank rows of pixels separating the bricks? How do I know the point values I assigned to the bricks will give a playable game?

The answer is, I don't. I made some reasonable guesses. If you ask a dozen experienced programmers to make the same choices, you would come up with several different values for each variable. At this point, though, we can see the bricks on the screen. We can study them, taking a moment to decide if the bricks are the right size, the right proportion, and the right color. If not, now is the time to make some changes. And since you used constants to isolate things like the number of rows and the size of bricks, it is easy to go back and make the changes.

What? You didn't use constants? Well, you should. That was one of those silly little rules of thumb I pointed out a long time ago. Now, developing a large program where you need to make fine adjustments to your program, you can start to see some of the value in that silly rule.

Drawing the Score and Balls

The next step is to draw the score and number of balls on the screen. This is pretty easy: just use MoveTo to set the pen position, SetForeColor and SetBackColor to choose appropriate colors for the letters, and write the scores. I used the following strings and positions, putting green letters near the bottom of the screen.

```
SetBackColor(0);
SetForeColor(2);
MoveTo(0,190);
writeln('Score:');
MoveTo(560,190);
writeln('Balls:');
```

The score and number of balls must be updated throughout the game, so I put the code to write these numbers in two procedures, called WriteScore and WriteBalls. By writing a few extra blanks after the numeric score, I made sure that any old text was erased.

You will also need to move the paddle up a bit. Since we used constants to set the paddle position, this is an easy thing to do. I raised the paddle to 16 pixels above the bottom of the screen.

Go ahead and get the program working up to this point.

Bouncing the Ball

We have a complete playing field now. The next step is to actually start the ball moving. You have written code to bounce a ball before; now is the time to add the same bouncing ball to our program.

The ball should start just below the bottom row of bricks, and head down and to the side. If you move it down two pixels and over four pixels with each movement, it will have a good angle, without being too steep. The horizontal position that it starts at, and the direction of the initial angle, should be chosen at random.

In the past, our simulations have used a fixed random number seed, or asked for a seed from the user. Neither option seems right for the break out game. If we use a fixed seed, the first ball will always come from the same place and the same direction. If we ask for a seed, the player can cheat. We have a pretty good choice for a seed in this program, though. The eventWhen field of the event record is the number of heartbeats since we started the event manager. This number will rarely be the same when we start the game, since the player has to click the mouse to start. (Well, not yet, but he will before we are finished.) In fact, for our purpose, the eventWhen field itself is suitably random! Here is how I chose the position and direction of the ball:

```
if odd(event.eventWhen) then
   dx := -4
else
   dx := 4;
x := ord(event.eventWhen mod 640);
```

The first line uses a function called odd, which returns true if the value is odd, and false if it is not. In effect, I have used a tiny part of the eventWhen field to decide if the ball should head to the left or right when it starts. The last line uses eventWhen again to set the initial horizontal position of the ball.

If you try your program now, the ball will be moving pretty slow. You need to turn debug code off to test the program's playability. The ball will still move too slow; that's something we'll take care of later. It turns out that QuickDraw isn't quick enough; in the 640 drawing mode, the paddle and ball can't be drawn quickly enough for a playable game. Later, we will correct this problem by switching to 320 graphics mode. There is a more serious problem, though. The ball moves noticeably slower when you move the paddle. This is because there is more work to do when the paddle is moved. The way to handle this problem is to put the game on a regular timer. We can do that by checking the eventWhen field, and only moving the ball after the field changes by some fixed amount. This also solves one other problem that plagues many arcade games. You may have seen some games that run too fast when they are played on a computer with an accelerator card. By fixing the speed of the game to the eventWhen field, which is updated 60 times per second no matter how fast the computer is, the game will always run at the same speed.

To use the eventWhat field as a timer, we start be recording the value of eventWhen in a global variable just as the ball starts to move. We can then check the timer, making sure that a certain amount of time has elapsed since the last time the ball was moved. The best place to do this is in our main event loop, which now looks like this:

```
repeat
```

```
   event :=
      GetNextEvent(eventMask, myevent);
   if (myEvent.eventWhen-lastWhen)
      > pause then begin
      MoveBall;
      lastWhen := lastWhen+pause;
      end; {if}
   MovePaddle;
until balls = 0;
```

Looking at this loop, what we are doing is to wait until at least pause heartbeats have occurred before we call MoveBall. We then update the value of lastWhen, but note that we add pause, rather than recording myevent.eventWhen. This will keep the ball movement smooth, even if some spot in our program takes up a bit too much time. Of course, you must pick a value for pause, which should be declared as a constant. Start with any value you like, and try hitting the ball with the paddle. Adjust pause until you like the way the game works.

The Bricks

We have gradually built up to the point where we can almost play a game. The last step is to keep track of when we hit the bricks, removing them when this happens.

In order to lower the bricks each time the screen is cleared, you have to record the position of some row of bricks in a global variable. After the screen is cleared, you can add 8 to this value, and redraw the bricks. This value is also the key to determining if we have hit a brick.

The first step is to decide if we are even near the bricks, and if so, which row of bricks we are near. To figure out how to do this, I will assume that you have defined the variables and constants shown in listing 12.3. If they have different names in your program, you will have to adjust the names used here. If these variables and constants don't exist in your program – if you are using hard-coded values, for example – now is a good time to repent. Go back and make the values constants.

The only variable that may seem a little curious is spacing. We will need to calculate the pixel position for the various rows. To do that, we need to know the distance from the bottom of one row to the bottom of the next row. Spacing is that value. It is the sum of the thickness of the brick, and the number of rows of black pixels that separate each row of bricks.

Using these values, we can quickly decide if we have hit a brick by calculating the row and column number for the ball.

```
row := (brickY-y+spacing)
   div spacing;
dispY := (brickY-y) mod spacing;
if (row in [1..rows])
   and (dispY <= thickness) then
   begin
   column := x div width + 1;
   dispX := x mod width;
   if stillThere[row,column] then
      begin
      {<<<remove the brick>>>}
      {<<<add in the score>>>}
      if dispY = thickness then
         {<<<hit from above>>>}
      else if dispY = 0 then
         {<<<hit from below>>>}
      else if dispX <
         (width div 2) then
         {<<<hit from left>>>}
      else
         {<<<hit from right>>>}
      end; {if}
   end; {if}
```

There is a lot packed into these few lines, so let's take a moment to look them over carefully. The first line calculates the row that the ball is in, returning a negative value if the ball is below all of the rows, and a value greater than the constant rows if the ball is above all of the rows. To see how it works, let's plug in a few sample numbers. When the game starts, brickY is set to 56. As the ball approaches the bricks from below, the value of y decreases steadily. Just before the ball gets to the first row of bricks, the value of Y is 58. BrickY-y+spacing is 6, so row is set to 0. When the ball hits the brick, y is 56. The value of brickY-y+spacing is 8, so the value of row gets set to 1. We have reached the first brick. Try values of y for 46, 48, and 50 to convince yourself that the formula returns the right value as the ball passes from the first row of bricks to the second.

The next line calculates dispY, which is the distance, in pixels, from the bottom of the row. It is 0 if the ball is on the bottom row, thickness if the ball is at the top of the brick, and greater than thickness if the ball is in the gap right above the brick, but before the start of the next row. We use this value twice, once to decide if we are in the gap between rows (in which case we didn't really hit a brick, after all), and again to see if we hit the top or bottom of a brick.

Figuring out the column number uses the same ideas, but is simpler because the leftmost column of bricks starts when x=0.

After checking to make sure the brick is still there, we drop into a series of if checks that decide which side of the brick we hit. Since the ball moves up and down two pixels at a time, and all of our program carefully aligns things to even pixel boundaries, it is easy to check for a hit from above or below: yDisp value is equal to 0 at the bottom of the brick, and thickness at the top. If we hit a corner, the program counts it as a hit from above or below. Checking for a hit from the left or right is tougher, though. The ball moves sideways at a variable rate, depending on the spin on the ball from the last time it hit the paddle. To check for a hit from the side, then, we need to allow for the possibility that the ball skipped right over the edge of the brick, imbedding itself in the side of the brick. Since we already know the ball has, in fact, hit the brick, we can make this check by looking to see if the ball is in the left or right half of the brick.

When I started filling in the stubs for this subroutine, I realized that you do exactly the same thing if the ball hits the top or the bottom of the brick. In both cases, the vertical velocity is reversed. The same was true if the hit was from the left or right: the horizontal velocity gets reversed in both cases. Keeping an eye out for this sort of simplification is an important part of programming. It simplifies this particular algorithm a great deal. Once I was finished, the test looked like this:

```
row := (brickY-y+spacing)
   div spacing;
dispY := (brickY-y) mod spacing;
if (row in [1..rows])
   and (dispY <= thickness) then
   begin
   column := x div width + 1;
   dispX := x mod width;
   if stillThere[row,column] then
      begin
      {<<<remove the brick>>>}
      {<<<add in the score>>>}
      if dispY in [0,thickness]
         then
         dy := -dy
      else
         dx := -dx
      end; {if}
   end; {if}
```

It's pretty straight-forward to fill in the stubs left in this routine, so I will let you do that. I did encapsulate the functions of removing a brick and adding in the new score into a subroutine, since they did involve a bit of code. You can remove the brick by drawing a black rectangle over the brick, and setting the proper spot in stillThere to false. Adding the proper value to the score is accomplished using the formula developed a few pages back. Don't forget to write the new score to the screen!

Once you have written all of the code, test your program.

Listing 12.3

```
const
   rows = 6;                  {# of rows of bricks}
   columns = 16;              {# of columns of bricks}
   thickness = 6;             {thickness of a brick}
   width = 40;                {width of a brick}
   spacing = 8;               {# of pixels between rows}
   maxX = 640;                {screen resolution}
   maxY = 200;

var
   x,y: integer;              {ball position}
   dx,dy: integer;            {ball velocity}
   brickY: integer;           {position of the bottom of the lowest brick}
```

Labarski's Rule of Cybernetic Entomology

Yup. The rule is, "There's always one more bug." (For more enlightenment when the chips get you down, refer to "Murphy's Law, and Other Reasons Why Things Go Wrong.")

There is at least one bug in your program at the moment. Maybe you caught it. If so, great. The point, though, is that in a large program you are going to overlook things. That's why you develop the program in small steps, making adjustments as you go.

You have probably already decided that a 1 pixel by 1 pixel ball is too small, and adjusted it. If you play the game for a while, though, you will eventually end up with the ball skimming along beside a brick, not quite touching it. When the animation routine erases the ball by redrawing it in black, part of the brick gets erased, too.

There are really two problems here. The first can be fixed by changing the way you animate the ball. Back in the sample program where we did a random walk with a variety of shapes, like stars, triangles and squares, you saw a new drawing mode called XOR mode. In this drawing mode, drawing an object twice erases it, since it is drawn by inverting pixels, not copying them to the screen. For example, you could lay out a series of 10 by 10 pennies, face up. XOR drawing mode is like flipping a penny over, showing tails. To erase the pixel, you flip it back - the same operation you used to draw it in the first place. By changing the ball so it is drawn in XOR mode, you will no longer erase parts of a brick that you skim over.

The second problem is that the player will expect the brick to go away if the ball comes in contact with it. Our routine to check to see if a brick has been hit checks to see if the center of the ball hit the brick. That's fine for a direct hit, but doesn't account for the grazing hits that we are seeing in the game. You can handle this situation in one of several ways. The one I like is to modify the ball movement algorithm so that, if the ball hits the space left by a brick that has already been removed, you check to the left or right, like this:

```
row := (brickY-y+spacing) div spacing;
dispY := (brickY-y) mod spacing;
if (row in [1..rows])
   and (dispY <= thickness) then begin
   column := x div width + 1;
   dispX := x mod width;
if stillThere[row,column] then begin
   HitBrick(row,column);
   if dispY in [0,thickness] then
      dy := -dy
   else
      dx := -dx;
   end {if}
else if dispY in [0,thickness] then
   if dispX = 0 then begin
      if stillThere[row,column-1]
         then begin
         HitBrick(row,column-1);
         dy := -dy;
         end; {if}
      end {if}
   else if dispX = width then
      if stillThere[row,column+1]
         then begin
         HitBrick(row,column+1);
         dy := -dy;
         end; {if}
end; {if}
```

Making the check this way presents a common problem, though. We can end up checking column number 0 or columns+1, neither of which actually exists in the array we have defined. Rather than adding even more complicated if checks to the program, most programmers will simply extend the array by one column in each direction, so that it goes from 0 to columns+1, rather than 1 to columns. The extra columns are initialized to false, so that the algorithm never reports that a brick has been hit.

You might ask why I didn't point out these factors when the original code was written. The reason was to make a point. In any large program, you will forget some detail. If your program is logically laid out, if constants have been defined, and if the program is commented well, it is usually very easy to go back and handle the special case. If you are not following the basic rules of structured programming, though, going back and making the change can be frightful.

In fact, programmers often leave out details like this on purpose. In a complex algorithm, you might want to check to be sure that the basic ideas work before spending a lot of time on little details. This is the algorithmic equivalent of a stub. You know what the detail is, but you ignore it for a while so you can concentrate on the overall structure of the algorithm. Once the basics work, you go back and fill in the details.

Filling in the Last Stubs

The program is almost finished. All that is left is to write the messages that control the start of the game, and missing a ball.

Let's do the last one first, since it is the easiest. When the player misses the ball, it would be heartless to toss the next one down right away, as you no doubt know by now! Instead, you need to stop and print a message, then wait for a mouse click. Start by writing a message in the middle of the screen, like this:

```
MoveTo(300,100);
write('Click for the next ball');
```

The next step is to wait for the mouse to be clicked and let back up. This is a pretty easy check.

```
repeat
   event :=
      GetNextEvent(eventMask, myevent);
until myevent.eventWhat = mouseUpEvt;
```

This loop keeps going until the player pushes the mouse button, then releases it.

When the game starts, or when a game is finished, you need to print two choices on the screen. For one choice, use the message, "Click here to start a game." For the other choice, use "Click here to quit." Use MoveTo and LineTo to draw boxes around these choices. I think it looks best if both boxes are the same size. With the messages drawn, you can use a loop very much like the last one to wait for a mouse click. This time, though, you want to make sure the click occurred in one of the buttons you have drawn. Write a function that returns 0 if the click was not in a button or if the mouse was not released, 1 if the click was in the "play a game" button, and 2 if the click was in the "quit" button. Your loop now looks like this:

```
repeat
   event :=
      GetNextEvent(eventMask, myevent);
   MovePaddle;
until WhichButton <> 0;
```

The only other thing that can happen – rare though it may be – is that the player might knock out all of the bricks. You can check for that when you erase a brick. To make the check quickly, you should use a global variable that is set to the number of bricks when they are first drawn. After that, if a brick gets hit, you can simply subtract 1 from the value and check to see if the value is zero. If so, you need to update the playing level, redraw all of the bricks, add one ball to the balls that the player has, and wait for the player to click on the mouse before continuing.

Tidy Up

The program is finished now, but some clean up is still left over. Scan through the program, making sure all of the stubs are filled in. Remove the throw-away code we put in to expand the graphics window, and change the program to an S16 application. Try it out to make sure everything still works as expected.

Unless you have an accelerator card, the program is still a little too slow. To correct this problem, convert the program from the 640 graphics mode to the 320 graphics mode. If you have faithfully used constants throughout your program, it will be fairly easy to scan through the program to make the appropriate changes. With these changes made, the program should be a very playable breakout game.

At this point, many programmers have the tendency to sit back, claiming the program is finished. Nonsense. It's about half to two-thirds there. The next step is to play the game for a while, adjusting the various features. I made a number of changes to the colors used in the program at this point. Once you are satisfied, ask a friend to play the game, and really listen to what he says. Don't be offended if he doesn't like something. Instead, write it down and keep it in mind. If you disagree, ask a few other people. If it turns out that everyone in the world but you makes the wrong choice, you might want to change the program, anyway. Sure, you're right – but a lot more people will be happy if you make the change. Finally, make a last pass through the source code for the program itself, tidying things up and looking for internal improvements. This is an especially important step if you are not satisfied with the speed of the program.

Lesson Twelve
The Complete Game

```
{ This program plays the game of break-out, a classic arcade    }
{ game.                                                         }

program Paddle (output);

uses Common, QuickDrawII, EventMgr, MemoryMgr;

const
   eventMask = $0F6E;           {GetNextEvent event mask}
   letterY = 200;               {height of letters for score, ball count}
   pause = 1;                   {60ths of a sec. to pause}

                                {the paddle}
                                {----------}
   area1 = 7;                   {sensitive areas of the paddle}
   area2 = 14;
   area3 = 21;
   area4 = 28;
   easy = 2;                    {x velocity for easy spin}
   hard = 3;                    {x velocity for hard spin}
   paddleColor = 15;            {paddle color}
   paddleHeight = 3;            {paddle height}
   paddleWidth = 35;            {paddle width}
   paddleY = 184;               {y position of the paddle}

                                {the bricks}
                                {----------}
   columns = 16;                {# of bricks in a row}
   columnsp1 = 17;              {columns + 1}
   rows = 6;                    {# of rows of bricks}
   spacing = 8;                 {spacing of the rows}
   startHeight = 56;            {initial space above the top row of bricks}
   thickness = 6;               {thickness of a brick}
   width = 20;                  {width of a brick}

                                {the ball}
                                {--------}
   ballColor = 15;              {ball color}
   ballHeight = 3;              {ball height; should be odd}
   ballWidth = 3;               {ball width; should be odd}
   balldx = 1;                  {(ballWidth - 1) div 2}
   balldy = 1;                  {(ballHeight - 1) div 2}
   speed = 2;                   {vertical ball speed}
```

```
var
   event: boolean;              {event flag; returned by GetNextEvent}
   lastWhen: longint;           {event timer}
   myEvent: eventRecord;        {current event record}
   score: integer;              {current score}
   screen: rect;                {port rectangle}

                                {the paddle}
                                {----------}
   maxX: integer;               {max X distance the paddle can travel}
   paddlePosition: integer;     {current X position of the paddle}

                                {the bricks}
                                {----------}
   brickY: integer;             {disp to the bottom row of bricks}
   level: integer;              {playing level}
   numBricks: integer;          {# of bricks visible}
   stillThere: array[1..rows,0..columnsp1] of boolean; {brick array}

                                {the ball}
                                {--------}
   balls: integer;              {# of balls left}
   dx,dy: integer;              {speed of the ball}
   x,y: integer;                {position of the ball}


   procedure StartTools;

   { Start the tools                                          }

   const
      size = 320;               {graphics mode}

   var
      memory: handle;           {memory returned by NewHandle}

   begin {StartTools}
   StartGraph(size);            {initialize QuickDraw}
   memory := NewHandle(256,UserID,$C015,nil); {start up the event mgr}
   EMStartUp(ord(memory^),0,0,size,0,200,UserID);
   end; {StartTools}
```

```
procedure ShutDownTools;

{ Shut down the tools                                              }

begin {ShutDownTools}
EMShutDown;
EndGraph;
end; {ShutDownTools}


procedure DrawPaddle (position,color: integer);

{ Draw the paddle                                                  }
{                                                                  }
{ Parameters:                                                      }
{    position - position to draw the paddle                        }
{    color - color of the paddle                                   }

begin {DrawPaddle}
                              {set the pen to draw the entire paddle}
SetPenSize(paddleWidth,paddleHeight);
SetSolidPenPat(color);        {set the paddle color}
SetPenMode(0);                {use copy mode}
MoveTo(position,paddleY);     {draw the paddle}
LineTo(position,paddleY);
end; {DrawPaddle}


procedure MovePaddle;

{ Track and move the paddle                                        }

begin {MovePaddle}
{convert the point to our window}
GlobalToLocal(myevent.eventWhere);
{make sure we don't go off of the screen}
if myevent.eventWhere.h+paddleWidth > maxX then
   myevent.eventWhere.h := maxX-paddleWidth;
{if the mouse moved, move the paddle}
if myevent.eventWhere.h <> paddlePosition then begin
   DrawPaddle(paddlePosition,0);
   paddlePosition := myevent.eventWhere.h;
   DrawPaddle(paddlePosition,paddleColor);
   end; {if}
end; {MovePaddle}
```

```
function PlayAGame: boolean;

{ See if the player wants to play a game or quit.            }
{                                                            }
{ Returns: True to play a game, else false.                  }

const
   left = 110;                       {dimensions of the buttons}
   right = 210;
   top1 = 90;
   bottom1 = 101;
   top2 = 105;
   bottom2 = 116;

var
   r: rect;                          {rect inclosing the buttons}


   function WhichButton: integer;

   { See which button the mouse is in                        }
   {                                                         }
   { Variables:                                              }
   {    myEvent.eventWhere - location of mouse at mouseup    }
   {    myEvent.eventWhat - kind of event                    }

   begin {WhichButton}
   WhichButton := 0;
   if myEvent.eventWhat = mouseUpEvt then
      with myEvent.eventWhere do
         if h >= left then
            if h <= right then
               if v >= top1 then
                  if v <= bottom2 then
                     if v <= bottom1 then
                        WhichButton := 1
                     else if v >= top2 then
                        WhichButton := 2;
   end; {WhichButton}
```

```
begin {PlayAGame}
MoveTo(left+10,bottom1-2);          {draw the messages}
write('Play a Game');
MoveTo(left+30,bottom2-2);
write('Quit');
SetSolidPenPat(14);                  {draw the button outlines}
SetPenMode(0);
SetPenSize(3,1);
MoveTo(left,top1);
LineTo(right,top1);
LineTo(right,bottom1);
LineTo(left,bottom1);
LineTo(left,top1);
MoveTo(left,top2);
LineTo(right,top2);
LineTo(right,bottom2);
LineTo(left,bottom2);
LineTo(left,top2);
ShowCursor;                          {wait for a click in a button}
repeat
   event := GetNextEvent(eventMask,myEvent);
   GlobalToLocal(myEvent.eventWhere);
   MovePaddle;
until WhichButton <> 0;
HideCursor;
r.h1 := left;                        {erase the messages}
r.h2 := right;
r.v1 := top1;
r.v2 := bottom2;
SetSolidPenPat(0);
PlayAGame := WhichButton = 1;        {set the return value}
end; {PlayAGame}
```

```
procedure DrawBrick (row,column,color: integer);

{ Draw a brick on the screen                                    }
{                                                               }
{ Parameters:                                                   }
{    row,column - brick to draw                                 }
{    color - color of the brick                                 }
{                                                               }
{ Variables:                                                    }
{     brickY - distance to the bottom of the bricks             }

var
   r: rect;                  {brick's rectangle}

begin {DrawBrick}
SetPenMode(0);               {get ready to draw}
SetSolidPenPat(color);
SetPenSize(1,1);
r.h1 := (column-1)*width;    {set up the brick's rectangle}
r.h2 := r.h1+width-1;
r.v2 := brickY - row*spacing + spacing;
r.v1 := r.v2-thickness;
PaintRect(r);                {draw the brick}
SetSolidPenPat(0);           {draw a line to separate the bricks}
MoveTo(r.h2+1, r.v1);
LineTo(r.h2+1, r.v2);
end; {DrawBrick}


procedure DrawBricks;

{ Draw a set of bricks                                          }

var
   colors: array[1..rows] of integer; {brick colors}
   column: 1..columns;       {loop variable}
   row: 1..rows;             {loop variable}
   r: rect;                  {brick rectangle}
```

```
begin {DrawBricks}
numBricks := rows*columns;   {set the brick count}
colors[1] := 7;              {fill in the brick color array}
colors[2] := 6;
colors[3] := 9;
colors[4] := 10;
colors[5] := 13;
colors[6] := 12;
for row := 1 to rows do      {draw the bricks}
   for column := 1 to columns do
      DrawBrick(row,column,colors[row]);
for column := 1 to columns do {all bricks are still there}
   stillThere[1,column] := true;
stillThere[1,0] := false;
stillThere[1,columnsp1] := false;
for row := 2 to rows do
   stillThere[row] := stillThere[1];
end; {DrawBricks}


procedure WriteBalls;

{ Draw the number of balls left                              }
{                                                            }
{ Variables:                                                 }
{    balls - number of balls left                            }

const
   ballX = 290;

begin {WriteBalls}
MoveTo(ballX,letterY);
write(balls:1, ' ':10);
end; {WriteBalls}


procedure WriteScore;

{ Draw the current score                                     }
{                                                            }
{ Variables:                                                 }
{    score - score to draw                                   }

const
   scoreX = 50;

begin {WriteScore}
MoveTo(scoreX,letterY);
write(score:1, ' ':10);
end; {WriteScore}
```

```
procedure DrawBall;

{ Draw a ball at the current position                          }
{                                                              }
{ Variables:                                                   }
{    x,y: ball position                                        }
{                                                              }
{ Note: This procedure is used to draw an initial ball or     }
{       to erase one after a ball is missed.  MoveBall uses   }
{       its own method, which is faster when the ball is      }
{       being animated.                                       }

begin {DrawBall}
SetSolidPenPat(ballColor);
SetPenMode(2);
SetPenSize(ballWidth,ballHeight);
MoveTo(x-balldx,y-balldy);
LineTo(x-balldx,y-balldy);
end; {DrawBall}


procedure StartBall;

{ Start a ball                                                 }
{                                                              }
{ Variables:                                                   }
{    x,y - position of the ball                                }
{    dx,dy - speed of the ball                                 }

begin {StartBall}
if odd(myEvent.eventWhen) then       {set the speed, position}
   dx := -easy
else
   dx := easy;
x := ord(myEvent.eventWhen mod screen.h2);
dy := speed;
y := startHeight+spacing*rows+4;

DrawBall;                            {draw the ball}

event := GetNextEvent(eventMask, myevent); {set the timer}
lastWhen := myEvent.eventWhen;
end; {StartBall}
```

```
procedure WaitForClick;

{ Pause until the player is ready for a ball                }

var
   r: rect;                          {used to erase the message}

begin {WaitForClick}
MoveTo(65,100);                      {write the message}
write('Click for the next ball');
repeat                               {wait for the click}
   event := GetNextEvent(eventMask,myEvent);
   MovePaddle;
until myEvent.eventWhat = mouseUpEvt;
r.h1 := 0;                           {erase the message}
r.h2 := 320;
r.v1 := 90;
r.v2 := 100;
SetSolidPenPat(0);
SetPenMode(0);
PaintRect(r);
end; {WaitForClick}


procedure CheckBricks;

{ Move the ball.                                            }
{                                                           }
{ Variables:                                                }
{    x,y - position of the ball                             }
{    numBricks - # of bricks left                           }

var
   ball: rect;                       {used to erase the ball}
   row,column: integer;              {brick row,column}
   dispX,dispY: integer;             {position along the brick}
```

```
   procedure HitBrick (row,column: integer);

   { Handle a hit brick                                          }
   {                                                             }
   { Parameters:                                                 }
   {    row,column - brick that was hit                          }

   begin {HitBrick}
   stillThere[row,column] := false;  {remove the brick}
   ball.h1 := x-balldx;              {erase the ball}
   ball.h2 := ball.h1+ballWidth;
   ball.v1 := y-balldy;
   ball.v2 := ball.v1+ballHeight;
   SetPenMode(2);
   SetSolidPenPat(ballColor);
   PaintRect(ball);
   DrawBrick(row,column,0);          {erase the brick}
   SetPenMode(2);                    {redraw the ball}
   SetSolidPenPat(ballColor);
   PaintRect(ball);
   score := score+(row+level)*5;     {add in the score}
   WriteScore;
   numBricks := numBricks-1;         {see if they are all gone}
   if numBricks = 0 then begin
      DrawBall;
      balls := balls+1;
      WriteBalls;
      brickY := brickY+spacing;
      level := level+1;
      DrawBricks;
      WaitForClick;
      StartBall;
      end; {if}
   event := GetNextEvent(eventMask, myevent); {reset the timer}
   lastWhen := myEvent.eventWhen;
   end; {HitBrick}

begin {CheckBricks}
row := (brickY-y+spacing) div spacing; {find the vertical brick values}
dispY := (brickY-y) mod spacing;
if (row in [1..rows]) and (dispY <= thickness) then begin
   column := x div width + 1;        {find the horizontal brick values}
   dispX := x mod width;
```

```
                                    {check for a hit}
   if stillThere[row,column] then begin
      HitBrick(row,column);
      if dispY in [0,thickness] then
         dy := -dy
      else
         dx := -dx;
      end {if}
   else if dispY in [0,thickness] then
      if dispX = 0 then begin
         if stillThere[row,column-1] then begin
            HitBrick(row,column-1);
            dy := -dy;
            end; {if}
         end {if}
      else if dispX = width then
         if stillThere[row,column+1] then begin
            HitBrick(row,column+1);
            dy := -dy;
            end; {if}
   end; {if}
end; {CheckBricks}


procedure MoveBall;

{ Move the ball.                                                  }
{                                                                 }
{ Variables:                                                      }
{    x,y - position of the ball                                   }
{    dx,dy - speed of the ball                                    }

var
   oldBall,newBall: rect;    {ball rectangles}
   px: integer;              {disp of ball on paddle surface}
```

```
   procedure GetANewBall;

   { Missed; get a new ball                                  }

   begin {GetANewBall}
   balls := balls-1;                {reduce the number of balls}
   WriteBalls;
   if balls <> 0 then begin
      PaintRect(oldBall);           {erase the old ball}
      WaitForClick;                 {wait until the player is ready}
      StartBall;                    {start a new ball}
      oldBall.h1 := x-balldx;       {form its rectangle}
      oldBall.h2 := oldBall.h1+ballWidth;
      oldBall.v1 := y-balldy;
      oldBall.v2 := oldBall.v1+ballHeight;
      end; {if}
   end; {GetANewBall}


begin {MoveBall}
SetPenMode(2);               {get ready to draw}
SetSolidPenPat(ballColor);
oldBall.h1 := x-balldx;      {form the old ball rectangle}
oldBall.h2 := oldBall.h1+ballWidth;
oldBall.v1 := y-balldy;
oldBall.v2 := oldBall.v1+ballHeight;
x := x+dx;                   {move the ball}
if x < 0 then begin
   x := 0;
   dx := -dx;
   end {if}
else if x > screen.h2 then begin
   x := screen.h2;
   dx := -dx;
   end; {else if}
y := y+dy;
if y < 0 then begin
   y := 0;
   dy := -dy;
   end {if}
```

```
else if y >= paddleY then begin
   if (x < paddlePosition) or (x > paddlePosition+paddleWidth) then
      GetANewBall
   else begin
      px := x-paddlePosition;
      if px < area1 then
         dx := -hard
      else if px < area2 then
         dx := -easy
      else if px < area3 then
         dx := 0
      else if px < area4 then
         dx := easy
      else
         dx := hard;
      dy := -dy;
      y := paddleY;
      end; {else}
   end; {else if}
newBall.h1 := x-balldx;      {form the new ball rectangle}
newBall.h2 := newBall.h1+ballWidth;
newBall.v1 := y-balldy;
newBall.v2 := newBall.v1+ballHeight;
PaintRect(newBall);          {draw the ball in the new spot}
PaintRect(oldBall);          {erase the old ball}
end; {MoveBall}


procedure InitScreen;

{ Draw the initial screen                                    }

var
   column: 1..columns;       {loop variable}
   row: 1..rows;             {loop variable}

begin {InitScreen}
SetSolidPenPat(0);           {erase the old screen contents}
SetPenMode(0);
PaintRect(screen);
brickY := startHeight;       {draw the initial set of bricks}
DrawBricks;
paddlePosition := 0;         {draw the initial paddle}
DrawPaddle(0,paddleColor);
```

```
   SetForeColor(11);            {draw the initial score, ball count}
   SetBackColor(0);
   MoveTo(0,letterY);
   writeln('Score:');
   MoveTo(240,letterY);
   writeln('Balls:');
   score := 0;
   WriteScore;
   balls := 3;
   WriteBalls;
   end; {InitScreen}


begin
StartTools;                     {start the tools}
InitCursor;                     {set up the cursor}
HideCursor;
GetPortRect(screen);            {set the limit on the paddle}
maxX := screen.h2;
level := 1;                     {play level starts at 1}
InitScreen;                     {give them something to look at}

while PlayAGame do begin
   InitScreen;                  {set up the screen}
   StartBall;                   {start a ball}
   repeat                       {event loop}
      event := GetNextEvent(eventMask, myevent);
      if (myEvent.eventWhen-lastWhen) > pause then begin
         MoveBall;
         CheckBricks;
         lastWhen := lastWhen+pause;
         end; {if}
      MovePaddle;
   until balls = 0;
   DrawBall;                    {erase the last ball}
   end; {while}

ShutDownTools;                  {shut down the tools}
end.
```

Lesson Thirteen
Scanning Text

The Course of the Course

This lesson, and the three that follow, mark a changing point in the Learn to Program course. Instead of springing it on you with no warning, I thought it would be best to stop and look at what we have done so far, and what is left.

The first eleven lessons were concerned primarily with teaching you the mechanics of programming. In those lessons, you learned most of the features of Standard Pascal, as well as a few common extensions to Pascal. While we used a number of real programs to illustrate the features of the Pascal language, and frequently discussed principals of good programming practice, crafting a program was not the primary topic.

In Lesson 12, you put the Pascal language to use for the first time in a significant sized program. In that lesson, we also reviewed and expanded upon the basic techniques used to craft a program. Hopefully, you came away from that lesson with a better idea of why top-down design, logical organization, encapsulating ideas in subroutines, and commenting are important.

It turns out that a few tasks turn up repeatedly in many different kinds of programs. The next four lessons deal with some of these basic techniques. In the process, you will get a chance to hone your programming skills.

Because the nature of the material is changing, we will also change our approach a bit. In the first part of the course, the text was laced with complete programs to illustrate the basic ideas. As the topics have changed, we have gradually moved away from that technique. Starting with this lesson, we will abandon it almost completely. Instead, we will talk about the concepts behind a particular algorithm. Many times, complete subroutines will be shown. The problems, for the most part, will involve using these ideas to create complete programs. As always, the solutions are given, so if you get stuck you can always refer to the complete solution.

There are a number of reasons for changing to this approach. One is that you know how to create a program, now, but you still need lots of practice to get really good at it. Another is that we will be able to cover a lot more material this way. Finally, when the course is over, I want you to know how to read intermediate computer science books – the kind of books that teach you about data structures, compiler theory, animation, and so on. Most of these books also give algorithms. If you are used to learning about programming methods by studying algorithms when you see these books for the first time, you will get a lot more out of them. I think it is better to learn to read an algorithm in a setting like this course, when complete programs are at least provided as part of the solution to a problem. In the algorithm books, you won't generally find any complete programs at all.

Manipulating Text

In today's world of graphically based computers, it might seem that manipulating text just isn't important anymore. As it turns out, though, that simply isn't true. Stop and think about it for a moment. The editor you use to type in programs manipulates text. The dialogs you use to enter search strings handle text. The Pascal compiler that creates programs starts with a text file. From word processors to spread sheets to adventure games, text is still the most common way to store information in a computer, so programs still have to manipulate text. That means that, as a programmer, you should know some of the basic techniques used to deal with text.

Programs that deal with text generally divide the task up into well-defined subtasks. These are called scanning, parsing, and semantics. A compiler is a classic example of a program that manipulates text, so we will start by looking at each of these tasks from the standpoint of a Pascal compiler. Later, we will see how many other programs use these same ideas.

Scanning, also called lexical analysis, is the process of collecting characters from the text and forming the characters into words. It's not that hard to do, but the idea is a very powerful one. As a quick example, let's look at a simple Pascal program, and see how a scanner would break it up into words.

```
program Hello(output);

begin
writeln('Hello, world.');
end.
```

It is tempting to look at this program as a collection of characters, but if you stop and think about it for a minute, that isn't the way you read it. Instead of individual letters, you group the program into words.

Compilers do the same thing. The scanner is responsible for reading the characters and forming words from the characters. These words are called tokens. The main driver for the compiler never even looks at the characters. Instead, it calls a subroutine, which we will call GetToken, that reads characters until a complete word is formed, then returns a single value that indicates what the word is. The scanner would break our short sample program down into reserved words and reserved symbols, like program and); constants, like the string written by writeln; and identifiers, like Hello. In the case of the identifiers, the scanner also returns a string variable with the name of the identifier. For constants, it returns the value of the constant.

Scanner's aren't limited to compilers. Virtually any program that deals with words uses a scanner of some sort. Spelling checkers, text adventure games, and even some advanced database programs that accept English-like questions are just a few of the programs that use a scanner.

The next step in the process is called parsing. The parser looks at a sequence of tokens to see if they fit certain preconceived patters. For example, the Pascal compiler knows that the first thing it should see is the reserved word program, and that this should be followed by a symbol. Compilers, grammar checkers and adventure games are all examples of programs that use parsers.

The last step is called semantic analysis. That's a fancy way of saying that the program figures out what the words mean. In the case of a compiler, semantic analysis is when the compiler decides what machine code instructions will do what you want the program to do. In an adventure game, semantic analysis is when the game decides that "I want to go north" means that the character should be moved from his current location to another location.

Building a Simple Scanner

The first step in writing a scanner is to decide, in very precise terms, what we mean by a token. In the case of a spelling checker, we could define a token as any stream of characters that starts with a letter, and contains only letters. Any other characters, such as punctuation marks or numbers, can be ignored, since you can't misspell a number or a comma. You can misuse them, of course, but not misspell them. A Pascal compiler can't afford to skip commas or numbers, but it can skip comments, spaces, and end of line marks. On other words, one of the jobs of the scanner is to skip characters that are not relevant to the main program.

Let's start with a scanner for a spelling checker. We will skip characters until we get to an alphabetic character, then collect the characters into a string until we get to a non-alphabetic character. There are two problems that have to be dealt with. The first is how to know how big a word can be, while the second is how to tell the main program that there are no more words.

To solve the first problem, we will use a bit of trivia. The longest word in the English language is antidisestablishmentarianism. It has 28 characters. If we allow 29 characters in a word, then, we can hold any legitimate word in the English language. (We need 29 characters instead of 28 so that we will have room for an erroneous character at the end of an otherwise legitimate 28 character word.) Anything else must be misspelled. In the scanner for our spelling checker, we will collect up to 29 characters, and return those characters. Any other characters will be ignored.

The second problem can be solved in a variety of ways. For a spelling checker, though, we will use a particularly simple one. When we get to the end of the file, we will return a null string.

All of this can be expressed in a very short subroutine called GetToken, shown in listing 13.1.

Take a close look at how the subroutine works. Think it through by writing the values of len, f^ and token on a sheet of paper, and tracing through the subroutine by hand for a short text sample. Make sure you understand how it meets our basic requirements to collect words, skip unneeded characters, return a null string at the end of the file, and handle words longer than maxLength. (Naturally, maxLength is a constant, in this case 29, defined in the main part of the program.)

Problem 13.1. Write a program based on GetToken that will scan a text file and write a list of the words in the file, one word per line. As a test, try the program on the source code for the program itself.

Testing a program for unusual conditions is a very important part of the programming process. In this program, the unusual condition is a word that is too long. One way to test for a word that is too long is with a special test file, but there is another way, too. Since the maximum length of a word is a constant, you can change the constant to 8, or some other small value, and try the program again. Use this method to make sure your program handles words longer than maxLength correctly.

Listing 13.1

```
procedure GetToken;

{ Read a word from the source file                                    }
{                                                                     }
{ Variables:                                                          }
{    f - source file                                                  }
{    token - string read                                              }

var
   len: integer;               {length of the string}

begin {GetToken}
{initialize the length of the string}
len := 0;
{skip to the first character}
while (not (f^ in ['a'..'z','A'..'Z'])) and (not eof(f)) do
   get(f);
{read the word}
while (not eof(f)) and (f^ in ['a'..'z','A'..'Z']) do begin
   if len < maxLength then begin
      len := len+1;
      token[len] := f^;
      end; {if}
   get(f);
   end; {while}
{set the length of the string}
token[0] := chr(len);
end; {GetToken}
```

Symbol Tables

One way to write a spelling checker is to collect each word and search for it in a dictionary. Depending on how the spelling checker works, if you find a word that is not in the dictionary, you could print it, display it and let the user correct or accept it, or save it and print a list of words later. This approach works pretty well for interactive spelling checkers. Not so long ago, though, spelling checkers were generally not built right into word processors. Instead, they were separate programs. In this kind of spelling checker, instead of looking up a word as soon as it is found, the words are saved in a linked list. Only one copy of each word is saved. After the entire document has been scanned, each word is looked up in the dictionary. This drastically cuts the number of times the program needs to look up a word. As a result, the spelling checker is a lot faster than one that looks up each word when it is read from the source file.

This list of words has a name: it is called a symbol table. Finding words in a symbol table is such a common task that an enormous amount of effort has gone into finding very fast ways to look up a word. We'll look at some of these later. For now, though, we will use a linked list.

To keep things simple, we generally don't put a word in a symbol table in the GetToken subroutine. Instead, the main program repeatedly calls GetToken, then another subroutine which we will call Insert. Insert creates the symbol table.

In most real programs, we put more than just the symbol itself in the symbol table. In our program, we will also keep track of how many times the word appeared in the file. The Insert procedure shows how this is done. It uses a record called symbolRecord, which defines a single entry in the symbol table. This record is defined globally, so that we can also use a

global variable to point to the first element of the linked list. The record looks like this:

```
symbolPtr = ^symbolRecord;
symbolRecord = record
   next: symbolPtr;
   count: integer;
   symbol: tokenType;
   end;
```

The type tokenType is a string type. You can also use it for the type of the global variable token, which is returned by GetToken. This is shown in Listing 13.2.

Problem 13.2. Using GetToken and Insert, create a program that will count the number of words in a file, and print the number of times each word appears in the file.

Parsing

At one time or another, you have probably played one of the adventure games that lets you type text commands. Did you ever wonder how they worked? Some of them can recognize all of these sentences, and in each case they will move the character to the north:

```
Go north.
Run to the north.
I want to move north, now.
North is the direction that I
   would like to go.
```

Many of these programs are pretty small, so they can't be doing anything particularly difficult. How do they work?

There is one surprisingly simple way to create a

Listing 13.2

```
procedure Insert;

{ Insert a word in the symbol table                                    }
{                                                                      }
{ Variables:                                                           }
{    token - symbol                                                    }
{    table - pointer to the first element in the symbol table          }

label 1;

var
   ptr: symbolPtr;              {work pointer}
   len: integer;                {length of the string}

begin {Insert}
ptr := table;                   {see if the symbol already exists}
while ptr <> nil do begin
   if ptr^.symbol = token then begin
      ptr^.count := ptr^.count+1;  {yes -> update the count and exit}
      goto 1;
      end; {if}
   ptr := ptr^.next;
   end; {while}
new(ptr);                       {no -> create a new entry}
ptr^.next := table;
table := ptr;
ptr^.count := 1;
ptr^.symbol := token;
1:
end; {Insert}
```

program that can recognize and act on all of these commands. It involves building a verb and subject table. Look carefully at the sentences. In each of our examples, there is a verb that indicates you want to move, like go or run. There is also a direction, north. The simple parsers used in the adventure games scan a sentence, looking for a verb and subject the program recognizes. All of the other words are simply discarded. The parser returns the verb and subject, and the program takes some action.

Games aren't the only place this method is used. The same basic idea is used in a program called Eliza, the first computer psychologist. This simple demonstration program is surprisingly effective at giving almost human-like responses, yet it is only a few dozen lines long. An even more direct application of this technology is found in some database query programs written for people who don't normally use computers. For example, you might type

```
Where can I find information
about Kansas and wheat crops?
```

The database program scans the line, finding just a few relevant words. The verb is find. There are two subjects, Kansas and wheat, separated by a boolean operator, and. The database program scans its list of articles and books, looking for all of the ones that have both Kansas and wheat in the list of key words.

Let's put these ideas to work in a simple parser to move a spot around on the screen. We are creating a simple robotic control language to move an object around. It would be natural for a person to use a variety of words to describe a direction, and a variety of words to describe movement. For movement, our parser will recognize go and move. For directions, it will recognize left, right, up, down, north, south, east and west. It is the parser's job to make things easy for the main program, so it will report only one value for each direction. We also need a way to quit, so we will add the verb quit to the parser. Quit does not have a subject; it simply means that we are finished. Stop will also be recognized as another form of quit. Our parser assumes that the scanner is converting all characters to uppercase, and that the scanner reads and processes one line at a time, rather than an entire file. In the GetAction subroutine that does the parsing, pay special attention to how none and nada are used to indicate that nothing has been found yet. These "empty" values simplify the program quite a bit.

This is a simple example of a parser. As the number of subjects and verbs increases, the number of rules that are used to combine them also goes up. Some subjects will apply only to certain verbs. In our program, we have an example of a verb, quit, that doesn't even have a subject. Some programs also allow subjects with no verb. For example, the adventure game Zork lets you type north, with no verb, to move north. As the possibilities grow, programmers start to use other techniques besides writing if statements for each possibility. Arrays can be used for moderate numbers of subjects and verbs. You index into the array by the subject and verb to find out which subroutine to call. For even more complex programs, techniques for writing rule-based programs can be used. In short, this subroutine gives you some basic ideas you can use to write a program that reads text. If you will be writing large programs using these ideas, though, you should spend some time looking at the more advanced techniques before starting your program.

Problem 13.3. Write a program to move a spot in the graphics window. The program should use a modified form of the GetToken parser that reads characters from a line, instead of a file. GetToken should also uppercase all of the characters in a token.

With these changes in mind, the business end of the main program should include a main loop that looks like this:

```
repeat
   GetAction;
   if verb = go then begin
      DrawPoint(x,y,3);
      case subject of
         up:    y := y-moveY;
         down:  y := y+moveY;
         left:  x := x-moveX;
         right: x := x+moveX;
         end; {case}
      DrawPoint(x,y,0);
      end; {if}
until verb = stop;
```

DrawPoint, of course, is the subroutine that actually draws the spot you are moving. You pass the location and color of the spot. MoveX and moveY are constants that tell how far to move the spot.

Be sure you remember to initialize x and y, and draw the initial spot, before the program starts.

Listing 13.3

```
type
   subjectType = (none,up,down,left,right);
   verbType = (nada,go,stop);

   procedure GetAction;

   { Find out what the player wants to do                        }
   {                                                             }
   { Variables:                                                  }
   {    verb - action to take                                    }
   {    subject - what we do the action to or with               }

   var
      done: boolean;                    {loop exit variable}

   begin {GetAction}
   done := false;                       {cycle until we get a good command}
   repeat
      readln(cline);                    {get a command line}
      lineIndex := 1;
      verb := nada;                     {start with no subject,verb}
      subject := none;
      repeat
         GetToken;                      {get a token}
                                        {handle a subject}
         if (token = 'NORTH') or (token = 'UP') then
            subject := up
         else if (token = 'SOUTH') or (token = 'DOWN') then
            subject := down
         else if (token = 'EAST') or (token = 'RIGHT') then
            subject := right
         else if (token = 'WEST') or (token = 'LEFT') then
            subject := left
                                        {handle a verb}
         else if (token = 'QUIT') or (token = 'STOP') then
            verb := stop
         else if (token = 'GO') or (token = 'MOVE') then
            verb := go;
      until length(token) = 0;
      case verb of                      {make sure the input is consistent}
         nada:
            writeln('Please tell me what to do (go or stop).');

         stop:
            done := true;

         go:
            if subject = none then
```

```
            writeln('Please tell me which way to go.')
         else
            done := true;
      end; {case}
until done;
end; {GetAction}
```

Lesson Thirteen
Solutions to Problems

Solution to problem 13.1.

```
program Words (output);

{ Write the words in a file }

const
   maxLength = 29;                     {length of a word}
   fname = 'prob.13.1.pas';            {file name to scan}

var
   token: string[maxLength];           {last word read}
   f: text;                            {file being scanned}


   procedure GetToken;

   { Read a word from the source file                              }
   {                                                               }
   { Variables:                                                    }
   {    f - source file                                            }
   {    token - string read                                        }

   var
      len: integer;                    {length of the string}

   begin {GetToken}
   {initialize the length of the string}
   len := 0;
   {skip to the first character}
   while (not (f^ in ['a'..'z','A'..'Z'])) and (not eof(f)) do
      get(f);
   {read the word}
   while (not eof(f)) and (f^ in ['a'..'z','A'..'Z']) do begin
      if len < maxLength then begin
         len := len+1;
         token[len] := f^;
         end; {if}
      get(f);
      end; {while}
   {set the length of the string}
   token[0] := chr(len);
   end; {GetToken}
```

```
begin
reset(f,fname);                          {open the file}
repeat                                   {scan the file}
   GetToken;
   if length(token) <> 0 then
      writeln(token);
until length(token) = 0;
end.
```

Solution to problem 13.2.

```
program Words (output);

{ Write the number of times a word occurs in a file }

const
   maxLength = 29;                       {length of a word}
   fname = 'prob.13.2.pas';              {file name to scan}

type
   tokenType = string[maxLength];        {a word from the file}
   symbolPtr = ^symbolRecord;            {ptr to a symbol table entry}
   symbolRecord = record                 {symbol table entry}
      next: symbolPtr;
      count: integer;
      symbol: tokenType;
      end;

var
   token: tokenType;                     {last word read}
   f: text;                              {file being scanned}
   table: symbolPtr;                     {the symbol table}


   procedure GetToken;

   { Read a word from the source file                          }
   {                                                           }
   { Variables:                                                }
   {    f - source file                                        }
   {    token - string read                                    }

   var
      len: integer;                      {length of the string}
```

```
begin {GetToken}
{initialize the length of the string}
len := 0;
{skip to the first character}
while (not (f^ in ['a'..'z','A'..'Z'])) and (not eof(f)) do
   get(f);
{read the word}
while (not eof(f)) and (f^ in ['a'..'z','A'..'Z']) do begin
   if len < maxLength then begin
      len := len+1;
      token[len] := f^;
      end; {if}
   get(f);
   end; {while}
{set the length of the string}
token[0] := chr(len);
end; {GetToken}


procedure Insert;

{ Insert a word in the symbol table                          }
{                                                            }
{ Variables:                                                 }
{    token - symbol                                          }
{    table - pointer to the first element in the symbol      }
{       table                                                }

label 1;

var
   ptr: symbolPtr;                  {work pointer}
   len: integer;                    {length of the string}

begin {Insert}
ptr := table;                       {see if the symbol already exists}
while ptr <> nil do begin
   if ptr^.symbol = token then begin
      ptr^.count := ptr^.count+1;   {yes -> update the count and exit}
      goto 1;
      end; {if}
   ptr := ptr^.next;
   end; {while}
new(ptr);                           {no -> create a new entry}
ptr^.next := table;
table := ptr;
ptr^.count := 1;
ptr^.symbol := token;
1:
end; {Insert}
```

```
   procedure PrintSymbols;

   { Print the symbol table                                          }
   {                                                                 }
   { Variables:                                                      }
   {    table - pointer to the first element in the symbol           }
   {       table                                                     }

   var
      ptr: symbolPtr;                  {work pointer}

   begin {PrintSymbols}
   ptr := table;
   while ptr <> nil do begin
      with ptr^ do
         writeln(count:10, '  ', symbol);
      ptr := ptr^.next;
      end; {while}
   end; {PrintSymbols}


begin
reset(f,fname);                         {open the file}
table := nil;                           {no symbols, yet}
repeat                                  {scan the file}
   GetToken;
   if length(token) <> 0 then
      Insert;
until length(token) = 0;
PrintSymbols;                           {print the symbol table}
end.
```

Solution to problem 13.3.

```
program Robot (input, output);

{ Move a "robot" around on the graphics screen }

uses Common, QuickDrawII;

const
   maxLength = 10;                      {length of a word}
   maxLine = 255;                       {max length of a line}
   moveX = 30; moveY = 10;              {distance of one move}

type
   subjectType = (none,up,down,left,right); {command subjects}
   verbType = (nada,go,stop);               {commands}
```

```
var
   cline: string[maxLine];              {command line}
   lineIndex: integer;                  {index of next char in line}
   token: string[maxLength];            {last word read}
   x,y: integer;                        {position of the robot}

   subject: subjectType;                {subject of the last command}
   verb: verbType;                      {verb of the last command}


   procedure GetToken;

   { Read a word from the source file                                  }
   {                                                                   }
   { Variables:                                                        }
   {    f - source file                                                }
   {    token - string read                                            }

   var
      len: integer;                     {length of the string}


      function ToUpper (ch: char): char;

      { Return the uppercase equivalent of a character                 }
      {                                                                }
      { Parameters:                                                    }
      {    ch - character to convert                                   }
      {                                                                }
      { Returns: Uppercase equivalent of ch                            }

      begin {ToUpper}
      if ch in ['a'..'z'] then
         ch := chr(ord(ch)-ord('a')+ord('A'));
      ToUpper := ch;
      end; {ToUpper}


   begin {GetToken}
   {initialize the length of the string}
   len := 0;
   {skip to the first character}
   while (not (cline[lineIndex] in ['a'..'z','A'..'Z']))
      and (lineIndex <= length(cline)) do
      lineIndex := lineIndex+1;
```

```
{read the word}
while (lineIndex <= length(cline))
   and (cline[lineIndex] in ['a'..'z','A'..'Z']) do begin
   if len < maxLength then begin
      len := len+1;
      token[len] := ToUpper(cline[lineIndex]);
      end; {if}
   lineIndex := lineIndex+1;
   end; {while}
{set the length of the string}
token[0] := chr(len);
end; {GetToken}


procedure GetAction;

{ Find out what the player wants to do                          }
{                                                               }
{ Variables:                                                    }
{    verb - action to take                                      }
{    subject - what we do the action to or with                 }

var
   done: boolean;                    {loop exit variable}

begin {GetAction}
done := false;                      {cycle until we get a good command}
repeat
   readln(cline);                    {get a command line}
   lineIndex := 1;
   verb := nada;                     {start with no subject,verb}
   subject := none;
   repeat
      GetToken;                      {get a token}
                                     {handle a subject}
      if (token = 'NORTH') or (token = 'UP') then
         subject := up
      else if (token = 'SOUTH') or (token = 'DOWN') then
         subject := down
      else if (token = 'EAST') or (token = 'RIGHT') then
         subject := right
      else if (token = 'WEST') or (token = 'LEFT') then
         subject := left
                                     {handle a verb}
      else if (token = 'QUIT') or (token = 'STOP') then
         verb := stop
      else if (token = 'GO') or (token = 'MOVE') then
         verb := go;
   until length(token) = 0;
```

```
      case verb of                    {make sure the input is consistent}
         nada:
            writeln('Please tell me what to do (go or stop).');

         stop:
            done := true;

         go:
            if subject = none then
               writeln('Please tell me which way to go.')
            else
               done := true;
         end; {case}
   until done;
   end; {GetAction}


   procedure DrawPoint (x,y,color: integer);

   { Draw the robot                                                }
   {                                                               }
   { Variables:                                                    }
   {    x,y - position of the robot                                }
   {    color - robot color                                        }

   begin {DrawPoint}
   SetSolidPenPat(color);
   SetPenMode(0);
   SetPenSize(9,3);
   MoveTo(x,y);
   LineTo(x,y);
   end; {DrawPoint}


begin
x := 60;                             {set up the initial robot}
y := 20;
DrawPoint(x,y,0);
repeat
   GetAction;                       {find out what we are supposed to do}
   if verb = go then begin             {if it is a movement then move}
      DrawPoint(x,y,3);                {erase the old robot}
      case subject of                  {move the robot}
         up:    y := y-moveY;
         down:  y := y+moveY;
         left:  x := x-moveX;
         right: x := x+moveX;
         end; {case}
      DrawPoint(x,y,0);                {draw the robot in the new spot}
      end; {if}
```

```
until verb = stop;
end.
```

Lesson Fourteen
Recursion

A Quick Look at Recursion

By now, you are well acquainted with defining and calling procedures. You know that a procedure has to be defined before it can be called. An interesting point about procedures that we haven't talked about, and that you may not have noticed, is that a procedure can call itself. After all, the definition of the procedure comes before the statements, so the procedure has been defined. The ability of a procedure to call itself opens up a whole new concept in programming, called recursion.

We will start our look at recursion using a simple example. The purpose of this first section is to tell you about the mechanics of recursion. You will learn a little about stack frames, and use the debugger to investigate how stack frames work. With the mechanics out of the way, we will look at recursion as a problem solving technique, solving the classic problem of the Towers of Hanoi. We will then combine recursion with a simple scanner, like the ones you wrote in the last lesson, to create a recursive descent expression evaluator.

How Procedures Call Themselves

Let's start by looking at a short program. This program multiplies two positive integers.

```
program Multiply (output);

   function Mult(x,y: integer): integer;

   begin {Mult}
   if y = 0 then
      Mult := 0
   else
      Mult := Mult(x, y-1) + x;
   end; {Mult}

begin
writeln(Mult(4,5));
end.
```

Let's face it, that's a pretty weird looking program. To understand how it works, we will start by tracing through the program. It would be a great idea to fire up the debugger and follow along on the computer.

Stepping through the program, the first thing that happens is Mult gets called with x = 4 and y = 5. After testing to see if y is zero, the subroutine executes this statement:

```
Mult := Mult(x, y-1) + x;
```

This statement is fairly strange all by itself. Here we have a function call, Mult(x,y-1), and an assignment to the function. You have seen both of these things by themselves, but never together on the same statement. What does this mean? Well, the statement calls Mult again, this time with x = 4, and y = 4. Assuming for the moment that this subroutine really will do a multiply properly, the function must return 16. (Following along with the debugger, you should use the Step Through command to execute the function call at full speed.) We then add x, which is 4, getting an answer of 20. This value is assigned to Mult, so it is the value the subroutine will return. Of course, 4*5 is, in fact, 20, so if the call to Mult with x = 4 and y = 4 works, the function will actually return the correct answer.

It is fair to ask how the compiler knows the difference between calling a function and assigning a value for the function to return. After all, the name of the identifier is the same in both cases, and as we have just seen, a function call and an assignment to set the value returned by the function can occur in the same subroutine. The answer lies in which side of the assignment operator the function name is used. If the function name occurs on the left side of the assignment operator, like this:

```
Mult := <some expression>;
```

the compiler evaluates the expression on the right side, and assigns the value to the function. The function then returns this value to whoever called the function. If the function name is used as part of an expression, like this:

```
<someplace to put the value> :=
   Mult(x, y-1) + x;
```

the compiler calls the function, and uses the value it returns.

Try running the program again, but this time, when you get into the subroutine, use the debugger to display x and y; the debugger will show 4 and 5, respectively. Continue to single step through the program until the

function calls itself again. The values for x and y vanish from the variables window. Enter them again. This time, the debugger shows that the values are 4 and 4, instead of 4 and 5.

The reason that the variable names disappeared when Mult called itself is because a new stack frame was created. A stack frame is a piece of dynamically allocated memory that the program reserves each time a procedure or function is called. This memory has a few housekeeping values that tell the compiler who to return to, as well as the parameters that are passed, and any locally declared variables. When Mult calls itself, a new stack frame is created. As with the first time the function is called, you need to tell the compiler which variables you want to look at.

A very crucial point is that the old stack frame still exists. In the old stack frame, y has a value of 5. To see the old stack frame, click on the up arrow in the variables window.

You can continue this process, single stepping through the subroutine until the final call, when y is set to 0. At this point, there are a total of six stack frames for the Mult function, each with a different value of y. Still, nothing has been returned. This time, though, the function does something different. Instead of calling itself again, the function returns 0. After returning, x is added to the zero that is returned, and the function returns again. This continues until the original stack frame is reached. At that point, the function returns the result of 20.

Problem 14.1. The example showed you how to do a multiplication using recursion. Basically, the program made use of the fact that, when n is any number greater than 0, m*n gives the same result as m*(n-1)+m. You can find the exponent of a number the same way. For example, 2^3 (2 raised to the power 3) is 8, or 2*2*2. This is the same as (2^2)*2. Change the program so it calculates an exponent, given two integers as input. Use it to verify that 5^4 is 625. As with the addition example, be sure and step through the program with the debugger.

Recursion is a Way of Thinking

Looking at all of those stack frames, I don't think it will be hard to convince you that you can't think about recursion the same way you think about if statements, while loops, and so forth. You will get so tangled up the the details of keeping track of all of the stack frames that you will forget what you are trying to accomplish. You may start to think that anyone that understands recursion must have a mind that would have made Einstein envious. I've watched a number of beginning programmers who would agree as they struggled with recursion, trying to analyze all of those stack frames, and keep track of all of those variables. It reminds me of the time I opened the course outline for Classical Mechanics in college and saw, on the front page of the outline, in the middle of the page, boldfaced, the following quote: "Any problem, no matter how difficult, can be made still more difficult by looking at it in the right way."

No kidding.

Once you understand that stack frames exist, and that they hold different copies of the variables, you should never trace through a recursive subroutine, trying to follow the stack frames, again. If you do, you are simply thinking about the problem the wrong way.

Instead, think about a piece of the problem, not the whole thing. Instead of thinking about the multiply as a series of function calls, look at what happens on any particular call. For the multiply function, there are two possibilities: either y is zero, or it is not. As you know, zero multiplied by any other number is still zero, so we know it is correct for the function to return zero if y is zero. If y is not zero, we apply a simple rule: x*y is the same as x*(y-1) + x. So, what is x*(y-1)? We don't know. More important, we don't care. The rule works all of the time, so we truly don't have to worry about what x*(y-1) is; a call to a correct multiply routine gives us that answer. With the answer to x*(y-1) in hand, we add x and return the correct answer for x*y. The crucial point to remember is that we don't try to trace through the morass of function calls to see what x*(y-1) will give us: we recognize that if the function returns the correct value for one terminal case, in our example when y = 0, and that if it returns the correct answer for x and y, assuming that x*(y-1) is done correctly, that it must return the correct answer all of the time. Mathematicians call this a proof by induction.

A good way to keep this in mind is to remember that any recursive subroutine must satisfy two conditions to work. First, it has to have a way to stop. In the case of the multiply subroutine, we stopped when y reached zero. Second, each call must move you closer to the stopping place than you were when the subroutine was called. In our multiply subroutine, any call that was made with y greater than 0 reduced y.

Let's put these ideas to work to solve a classic puzzle, the Towers of Hanoi. This is a puzzle that quickly befuddles anyone who tries to solve it iteratively, the way you have been writing programs up until this lesson. The puzzle starts with six disks, all of a different size, sitting on one of three pegs, like this:

The object is to move all of the disks from the left-hand peg to the right-hand peg. On each turn, you can move only one disk. The only other restriction is that you can never cover one disk with a larger disk. Stop and try this before going on. You can cut the six disks from pieces of paper, and stack them on your desk instead of using pegs.

So, did you solve the puzzle iteratively? Even if you didn't make any mistakes, it takes 63 different moves to solve the puzzle. Can you keep that many moves straight in your head? If so, you have a better mind than mine.

The way to solve the puzzle is to turn it around. Instead of trying to move the top disk, you have to realize that the real problem is to move the bottom disk! The goal is to move the top five disks from the first peg to the second, like this:

The next step is to move the bottom disk to the third peg.

The last step is to move the pile of five disks from the second peg to the third.

Expressing this as a Pascal procedure, we get something like this:

```
procedure Tower (fromPeg, toPeg,
   sparePeg: integer;
   count: integer);

begin {Tower}
if count <> 0 then begin
   Tower(fromPeg, sparePeg,
      toPeg, count-1);
   Move(fromPeg, toPeg);
   Tower(sparePeg, toPeg,
      fromPeg, count-1);
   end; {if}
end; {Tower}
```

Move, of course, is a subroutine that takes the top disk from one peg and places it on another. We could represent the different pegs as three arrays, one for each peg, with six spots in each array. Each spot could be empty, or it might have one of the disks.

The important thing to recognize is that we haven't worried about how to move five disks from the first peg to the second. We know that if we can move six disks by first moving the top five, then moving the bottom disk, and finally moving the top five disks again, that we can use exactly the same idea to move the five disks. After all, to move five disks, we start by moving four of them to the spare peg, then we move the bottom disk, and finally we move the four disks to the correct peg. To move four disks... well, you get the idea. Eventually, we end up with the trivial problem of moving one disk.

Problem 14.2. Write a program that solves the Towers of Hanoi problem. Draw the disks in the graphics window as they are moved around by the call to Move.

Problem 14.3. Recursion can be used to process a linked list in reverse order. To see this idea in action, write a program that builds a linked list, stuffing the numbers 1 to 10 in the records, like this:

```
for i := 1 to 10 do begin
   new(ptr);
   ptr^.next := list;
   ptr^.value := i;
   list := ptr;
   end; {for}
```

Next, write a recursive procedure that prints the values in the list. On each call, the recursive procedure should return if the pointer that is passed

to it is nil. If the pointer is not nil, the procedure should call itself, then print the current value, like this:

```
Print(prt^.next);
writeln(ptr^.value);
```

After you write the program, reverse the last two statements, and run it again. This time, the program prints the numbers in reverse order. Make sure you understand why, tracing a few iterations with the debugger if you need to.

A Practical Application of Recursion

In the last lesson, we looked briefly at scanners and parsers. One of the easiest kind of parser to implement is called a recursive descent parser. To see how recursion can be used in a parser, we will solve a problem that had computer scientists stumped for a long time back in the early days of computing, when they were trying to write the first compilers. The problem is to solve a mathematically expressed equation.

For example, you know that

```
(4+5)*(1+2)
```

is evaluated by adding the terms in parenthesis first, then doing the multiply. How can we write a program that can do this? It's not an idle problem: over the years I have been asked to write a number of programs that had to solve an equation like this one. The problem doesn't just crop up in computer languages, either. You need to solve equations in math programs that graph functions, in spread sheets, and even in some databases.

To see how to solve this problem, we will write a simple expression evaluator that can add, subtract, multiply and divide. It will accept integer numbers and parenthesis. Just as in algebra and Pascal, add and subtract will have the same precedence, and multiply and divide will have the same precedence, but multiply and divide have a higher precedence than add or subtract.

To get a grasp on how the expression evaluator will work, let's look at this expression:

```
4*5 + 9/2 - 6
```

To solve this expression by hand, we would first scan through, doing all of the multiply and divide operations, leaving only numbers and the add and subtract operations.

```
20 + 4 - 6
```

This equation can be solved by working from left to right, adding and subtracting each new value to the old value. Thinking recursively, we can solve this equation by calling a function to do all of the stuff besides addition and subtraction, then checking to see if there is an add or subtract operation, and finally looping. In true recursive style, not to mention structured programming style, we won't worry about how the subroutine that does the multiplies and divides works. Instead, we solve the smaller problem. Here is our solution, a function that calls another function, factor, to read numbers, do multiplication, and handle parenthesis, does the adds and subtracts that are left over, and returns the result. Our function assumes that the main program calls GetToken one time to collect the first token from the input line before expression is called; this is a very common technique in recursive descent parsers.

```
function Expression: integer;

{ Evaluate an expression }

var
   value,newValue: integer;
   operation: tokenType;

begin {Expression}
value := Factor;
while token in [add,subtract] do
   begin
   operation := token;
   GetToken;
   newValue := Factor;
   if operation = add then
      value := value+newValue
   else
      value := value-newValue;
   end; {while}
Expression := value;
end; {Expression}
```

Let's trace through this function with our sample expression,

```
20 + 4 - 6
```

to see how it works. When the function is called, the main program has already called GetToken, so the global variable token already has a value. It is holding an integer whose value is 20. So far, the function

Factor doesn't have to do much. It just checks to be sure that token is an integer value, returns the value, and reads in the next token. When we get to the start of the while loop, then, value is 20. The + character has been read, and token has been set to add.

At the start of the while loop, we save the operation in a variable called, surprisingly enough, operation, and read the next number. If there is an operation, there must be a number after it. We'll trust Factor to flag an error if the number is missing. We then call Factor to get the next number, skipping the number token in the process, and do the operation. At the end of the while loop, value is 24, and token is subtract. One more pass through the while loop finishes off the expression, and we return a final value of 30.

The next step is to handle multiplication and division. That's no trick, really. They work the same way addition and subtraction do! In this case, we will call a function called Term to handle numbers and parenthesis. Everything else is an echo of the function that handles addition and subtraction.

```
function Factor: integer;

{ Do multiplies and divides }

var
   value,newValue: integer;
   operation: tokenType;

begin {Factor}
value := Term;
while token in
   [multiply,divide] do begin
   operation := token;
   GetToken;
   newValue := Term;
   if operation = multiply then
     value := value*newValue
   else
     value := value div newValue;
   end; {while}
Factor := value;
end; {Factor}
```

Trace through our sample equation

```
4*5 + 9/2 - 6
```

to see how Factor works, and how Factor and Expression work together to make sure the operations are done in the correct order. For this short example, keeping track of the global variables term and token on a piece of paper should work out well.

The last step is to write the subroutine that handles numbers. There is one other thing that can appear at this point, though, and that is a parenthesis. Term handles that particular problem by calling Expression to evaluate whatever appears between the parenthesis! Expression can then call Factor, which will call Term, and so forth. This recursive call is what allows our expression handler to handle very complex equations.

```
function Term: integer;

{ Handle a number or        }
{ parenthesis               }

var
   value: integer;
   operation: tokenType;

begin {Term}
if token = int then begin
   Term := tokenValue;
   GetToken;
   end {if}
else if token = lparen then
   begin
   GetToken;
   Term := Expression;
   if token = rparen then
      GetToken
   else
      writeln(') expected');
   end; {else}
end; {Term}
```

Take a close look at the error message that is printed if Term finds an opening parenthesis, but no closing parenthesis. Does it look familiar? If not, you might glance through the list of error messages at the end of the ORCA/Pascal manual. Now you know where those error messages come from!

Problem 14.4. Write a program to evaluate an expression and write the value. Your program should handle addition, subtraction, multiplication, division, and parenthesis. All operations should be on integers.

Your program should start by prompting the user for an expression. It should then call GetToken to fetch the first token from the line, followed by a

call to Expression to evaluate the expression. The program should loop repeatedly, reading new expressions, until the line typed by the user is a null string.

While the text did not cover writing the GetToken subroutine, all of the concepts were covered in the last lesson. Try to write GetToken on your own; if you get stuck, refer to the solution.

Lesson Fourteen
Solutions to Problems

Solution to problem 14.1.

```
program Exponent (output);

   function Exp(x,y: integer): integer;

   begin {Exp}
   if y = 0 then
      Exp := 1
   else
      Exp := Exp(x, y-1) * x;
   end; {Exp}

begin
writeln(Exp(5,4));
end.
```

Solution to problem 14.2.

```
{ Graphic solution to the Towers of Hanoi puzzle. }

program Tower;

uses Common, QuickDrawII;

const
   disks = 6;                          {# of disks to move}

type
   peg = array[1..disks] of integer;    {one peg, with contents}

var
   i: integer;                         {loop variable}
   pegs: array[1..3] of peg;           {all three pegs}
```

```
procedure DrawDisk(disk, peg, height, color: integer);

{ Draw a disk on the screen                                    }
{                                                              }
{ Parameters:                                                  }
{    disk - disk (i.e. size) to draw                           }
{    peg - peg to draw the disk on                             }
{    height - distance from the bottom of the pile             }
{    color - color to draw the disk                            }

var
   x,y: integer;                  {position of the center of the disk}

begin {DrawDisk}
x := peg*80 - 40;                   {find the position}
y := 50 - height*4;
SetSolidPenPat(color);              {set the pen color}
MoveTo(x - disk*5, y);              {draw the disk}
LineTo(x + disk*5, y);
end; {DrawDisk}


procedure Tower(fromPeg, toPeg, sparePeg: integer; count: integer);

{ Move count pegs from peg # fromPeg to peg # toPeg            }
{                                                              }
{ Parameters:                                                  }
{    fromPeg - peg to move the disks from                      }
{    toPeg - peg to move the disks to                          }
{    sparePeg - unused peg                                     }
{                                                              }
{ Variables:                                                   }
{    pegs - the current content of each peg                    }


   procedure Move (fromPeg, toPeg: integer);

   { Move a disk                                               }
   {                                                           }
   { Parameters:                                               }
   {    fromPeg - peg to move the disk from                    }
   {    toPeg - peg to move the disk to                        }

   var
      i: integer;                   {peg array index}
      disk: integer;                {disk being moved}
```

```
      begin {Move}
      i := disks;                        {find the disk to move}
      while pegs[fromPeg,i] = 0 do
         i := i-1;
      disk := pegs[fromPeg,i];           {remove the disk}
      pegs[fromPeg,i] := 0;
      DrawDisk(disk, fromPeg, i, 3);     {erase the disk}
      i := 1;                            {find the new spot for the disk}
      while pegs[toPeg,i] <> 0 do
         i := i+1;
      pegs[toPeg,i] := disk;             {place the disk on the peg}
      DrawDisk(disk, toPeg, i, 0);       {draw the disk}
      end; {Move}


   begin {Tower}
   if count <> 0 then begin
      Tower(fromPeg, sparePeg, toPeg, count-1);
      Move(fromPeg, toPeg);
      Tower(sparePeg, toPeg, fromPeg, count-1);
      end; {if}
   end; {Tower}

begin
SetPenMode(0);                           {initialize the graphics pen}
SetPenSize(8,3);
for i := 1 to disks do begin             {set up the game}
   pegs[1,i] := disks-i+1;
   pegs[2,i] := 0;
   pegs[3,i] := 0;
   DrawDisk(pegs[1,i], 1, i, 0);
   end; {for}
Tower(1, 3, 2, disks);                   {move the disks}
end.
```

Solution to problem 14.3.

```
{ Use recursion to reverse the elements in a linked list }

program Print (output);

type
   listPtr = ^listRecord;
   listRecord = record                   {element in the list}
      next: listPtr;
      value: integer;
      end;
```

```
var
   i: integer;                          {loop variable}
   list: listPtr;                       {the list}
   ptr: listPtr;                        {work pointer}


   procedure Print (ptr: listPtr);

   { Print the values in the list in reverse order             }

   begin {Print}
   if ptr <> nil then begin
      Print(ptr^.next);
      writeln(ptr^.value);
      end; {if}
   end; {Print}

begin
for i := 1 to 10 do begin               {create the list}
   new(ptr);
   ptr^.next := list;
   ptr^.value := i;
   list := ptr;
   end; {for}
Print(list);                            {print the list}
end.
```

Solution to problem 14.4.

```
{ A simple, recursive descent expression evaluator.  This       }
{ program handles +, -, * and /, as well as parenthesis.  All   }
{ operations are integer operations.                            }

program Expression (input, output);

type
                                        {tokens in an expression}
   tokenType = (add,subtract,multiply,divide,int,lparen,rparen,eol);

var
   ch: char;                            {last char read by GetCh}
   index: integer;                      {index into str}
   str: string[80];                     {string read from the keyboard}
   token: tokenType;                    {last token read}
   tokenValue: integer;                 {value of last integer token}
```

```
procedure GetToken;

{ Read a token from the input string                          }


   procedure GetCh;

   { Read the next character from str                         }
   {                                                          }
   { Variables:                                               }
   {    ch - char read; chr(0) if at the end of the string    }
   {    index - index into str                                }
   {    str - string to read the character from               }

   begin {GetCh}
   if index >= length(str) then
      ch := chr(0)
   else begin
      index := index+1;
      ch := str[index];
      end; {else}
   end; {GetCh}


begin {GetToken}
while ch = ' ' do                    {skip to the first real character}
   GetCh;
if ch = chr(0) then
   token := eol                      {handle an end of line}
else if ch = '+' then begin
   token := add;                     {handle add}
   GetCh;
   end {else if}
else if ch = '-' then begin
   token := subtract;                {handle subtract}
   GetCh;
   end {else if}
else if ch = '*' then begin
   token := multiply;                {handle multiply}
   GetCh;
   end {else if}
else if ch = '/' then begin
   token := divide;                  {handle divide}
   GetCh;
   end {else if}
else if ch = '(' then begin
   token := lparen;                  {handle ( }
   GetCh;
   end {else if}
```

```
else if ch = ')' then begin
   token := rparen;                  {handle ) }
   GetCh;
   end {else if}
else if ch in ['0'..'9'] then begin
   token := int;                     {handle a number}
   tokenValue := 0;
   while ch in ['0'..'9'] do begin
      tokenValue := tokenValue*10 + ord(ch)-ord('0');
      GetCh;
      end; {while}
   end {else if}
else begin                           {handle bad input}
   writeln('"', ch, '" is an illegal character');
   GetCh;
   end; {else}
end; {GetToken}


function Expression: integer;

{ Evaluate an expression                                    }

var
   value, newValue: integer;         {values from Factor}
   operation: tokenType;             {type of the operation}


   function Factor: integer;

   { Do multiplies and divides                              }

   var
      value, newValue: integer;      {values from Term}
      operation: tokenType;          {type of the operation}


      function Term: integer;

      { Handle a number or parenthesis                      }

      begin {Term}
      if token = int then begin
         Term := tokenValue;         {handle an integer}
         GetToken;
         end {if}
```

```
      else if token = lparen then begin
         GetToken;                    {skip the ( }
         Term := Expression;          {evaluate the expression}
         if token = rparen then       {skip the ) }
            GetToken
         else
            writeln(') expected');
         end; {else}
      end; {term}


   begin {Factor}
   value := Term;                     {get the first value}
   while token in [multiply,divide] do begin
      operation := token;             {skip the operation}
      GetToken;
      newValue := Term;               {get the second value}
      if operation = multiply then    {do the operation}
         value := value*newValue
      else
         value := value div newValue;
      end; {while}
   Factor := value;                   {return the result}
   end; {Factor}


begin {Expression}
value := Factor;                      {get the first value}
while token in [add,subtract] do begin
   operation := token;                {skip the operation}
   GetToken;
   newValue := Factor;                {get the second value}
   if operation = add then            {do the operation}
      value := value+newValue
   else
      value := value-newValue;
   end; {while}
Expression := value;                  {return the result}
end; {Expression}
```

```
begin
repeat
   write('Expression: ');
   readln(str);
   if length(str) <> 0 then begin
      ch := ' ';
      index := 0;
      GetToken;
      writeln('The value is ', Expression:1);
      writeln;
      end; {if}
until length(str) = 0;
end.
```

Lesson Fifteen
Sorts

Sorting

Way back in Lesson 5, you got your first look at a sort. Sorting is a pretty common topic in programming courses for a number of reasons. First, there are many places in real programs where you need to sort some information. In some cases, it is pretty obvious that a sort is needed. For example, you may have sorted a database to put a list of people in alphabetical order. You may have sorted the same database to put the list in zip code order to get ready for a mass mailing. In other cases, the fact that something is being sorted is not so obvious, but sorts are none-the-less used. For example, the link editor that creates executable programs from your object files can create a sorted list of the symbols that appear in your program. A card playing game may sort a deck of cards.

Another reason sorts are a popular topic is because sorting is a topic that people have spent enough time on to understand fairly well. Computer scientists who deal with the efficiency of algorithms have studied sorts for a long time. In the process, they have compiled a rather impressive list of different ways to sort information.

The Shell Sort

The shell sort is one of several basic sorting methods that are easy to implement, easy to understand, and reasonably efficient for small amounts of information. In the shell sort, you loop over the information to be sorted, swapping entries if they are out of order. If you make a swap, you also set a flag to remind you that you found entries that were out of order. In that case, you will need to make another pass over the data to make sure it is in the right order. You keep doing this until you make a pass over the data without finding anything that is out of order. If you are a little fuzzy about the details, refer back to Lesson 5, where this sort was first performed. Here's a simple version of the sort that sorts an array of size numbers, where size is a constant or variable telling how many entries are in the array.

```
repeat
   swap := false;
   for i := 1 to size-1 do
      if nums[i] > nums[i+1] then
         begin
         temp := nums[i];
         nums[i] := nums[i+1];
         nums[i+1] := temp;
         swap := true;
         end; {if}
until not swap;
```

When we start to worry about how efficient a sort is, we usually look at how many times we have to compare the numbers, since that is the operation we do most often. Let's trace through this routine for a short example, and find out how efficient it is. We'll use a size of 5, with starting numbers of 5, 4, 3, 2 and 1, in that order. You should follow along with a pencil and paper, writing down the values of variables, executing this algorithm by hand, and counting the operations on your own.

The first time through the loop, we do four compares, and four swaps. The numbers in the array are ordered like this after the first time through the loop:

```
4  3  2  1  5
```

We still have to do four compares each time through the loop. After the next loop, and four more compares, the array looks like this:

```
3  2  1  4  5
```

This process continues until the numbers are sorted. We have to do one extra pass after all of the numbers are sorted, since we keep going until swap stays false. Here are the numbers in the array, along with the total number of compares we have performed, up to and including this time through the loop:

```
2  1  3  4  5       12
1  2  3  4  5       16
1  2  3  4  5       20
```

While we won't go through a formal mathematical proof, by trying a few cases, you can probably convince

yourself that if you are sorting n things, and the numbers start out in reverse order, the number of compares will be n*(n-1). Starting with the array in reverse order is the worst possible situation for this sort, so we call this the worst case run time.

In a sense, it is pretty unfair to judge anything by the worst case. This is especially true in computer science, since it turns out that in many situations, the typical run time for an algorithm is very different than the worst case run time. In fact, there are many situations where the algorithm that has the best worst case run time is not the one with the best typical run time. On the other hand, you do need to know the worst case time, too, since you may be planning a program that is very time critical. In other words, it pays to know as much about algorithms and their efficiency as you can take the time to learn. You may end up picking one method of sorting in one program, and a different method in another.

For most algorithms you are likely to need, you will be able to find the worst case run time in published books. What if you can't find out about the algorithm from a book? Or, what if you find the algorithm, but they don't tell you the typical run time, only the worst case run time? Well, you've already seen one way to find the worst case run time, by tracing through the program by hand. You could also do the same thing by machine, of course. While this doesn't give you a mathematical proof, counting the operations does give you a good handle on the run time of an algorithm. You can use the same idea to find the typical run time. These ideas are expanded on in the problems.

Problem 15.1. Write a program that creates an array of integers in reverse order, like the array we looked at in the example in this section. Be sure and use a constant for the size of the array. Sort the array using the algorithm shown, but add a counter than counts the number of compares. Print this value.

Run this program with arrays that have 2, 3, 4, 5, and 10 values. Do all of the numbers match the value n*(n-1)?

Problem 15.2. Finding the typical run time for an algorithm is a lot like finding the worst case run time, like you did in problem 15.1. If you have some actual samples of numbers you plan to sort, you can use the samples to find the typical run time. Another way is to use a simulation, filling the arrays with random values several times, then averaging the run time for the various sorts.

Try this method to find the typical run time for the shell sort. Modify the program from problem 15.1 so it uses a random number generator to fill the array with values between 1 and the size of the array. To keep things simple, allow duplicates. In other words, you don't have to check to be sure that the random number generator returns each possible value once; it is fine if the array has some duplicates. Do this 100 times, and average the number of compares. Find the values for arrays with 2, 3, 4, 5, and 10 elements.

Quick Sort

There are several ways of sorting information that are a little faster than the shell sort, but these generally still have a run time that is proportional to n*n, or something pretty close to n*n, like the n*(n-1) that we found for the shell sort. There are also some sorts that have a typical run time proportional to n*log(n)/ln(2). To see what this means, let's stop and think about a fairly common sorting problem, sorting a mailing list to zip-code order. There are a variety of mailing lists that come in a variety of sizes, but it isn't uncommon to have 100,000 names in a mailing list. Sorting 100,000 names using the shell sort has a worst case run time of 100,000*(100,000-1), or 9,999,900,000 compares. To say the least, doing nearly ten billion compares takes some serious computer time, especially if you are comparing floating-point numbers, or worse yet, strings. The faster sorts that work in n*log(n)/log(2) time, though, would do the same thing using 1,660,964 compares, which is over 6000 times faster!

The most popular of the fast sorts is a recursive sort called quick sort. Quick sort uses a divide and conquer technique. On each step, a pivot value is picked. Picking a good pivot value is something of a fine art, and it is a very important step. In most cases, the middle value is a good choice for the pivot value. For example, if you are sorting an array with indices from 1 to 100, you would use the 50th element as the pivot value. The routine then moves anything smaller than the pivot value to the left of the pivot, and anything larger than the pivot value to the right of the pivot. The recursive step comes next: the quick sort procedure calls itself, passing the part of the array to the left of the pivot, then makes another recursive call to sort the right half of the array.

Understanding how this works is pretty tricky, so let's get used to it slowly. Type in the program shown in listing 15.1 and make sure it works. It uses quick sort to sort a small array with ten values.

Listing 15.1

```
{ A sample of quick sort.                                          }

program Sort(output);

const
   size = 10;

var
   a: array[1..size] of integer;


   procedure Fill;

   { Fill the array                                                }
   {                                                               }
   { Variables:                                                    }
   {    a - array to fill                                          }

   var
      i: integer;

   begin {Fill}
   for i := 1 to size do
      a[i] := size-i+1;
   end; {Fill}


   procedure Sort (left,right: integer);

   { Sort an array                                                 }
   {                                                               }
   { Parameters:                                                   }
   {    left - leftmost part of the array to sort                  }
   {    right - rightmost part of the array to sort                }
   {                                                               }
   { Variables:                                                    }
   {    a - array to sort                                          }

   var
      i,j: integer;                      {array indices}
      pivot: integer;                    {pivot value}
      temp: integer;                     {used to swap values}

   begin {Sort}
   if right > left then begin            {quit if there is only 1 element}
      i := (left-1) + ((right-left+1) div 2); {find the pivot index}
      pivot := a[i];                     {put the pivot at the end}
      a[i] := a[right];                  {(remember the pivot, too)}
```

```
        a[right] := pivot;
        i := left;                          {set up the start indices}
        j := right-1;
        while i <> j do begin               {partition the array}
           while (a[i] <= pivot) and (i <> j) do
              i := i+1;
           while (a[j] >= pivot) and (i <> j) do
              j := j-1;
           temp := a[i];
           a[i] := a[j];
           a[j] := temp;
           end; {while}
        if a[i] < pivot then                {find the pivot insert point}
           i := i+1;
        temp := a[i];                       {replace the pivot}
        a[i] := a[right];
        a[right] := temp;
        Sort(left, i-1);                    {sort to the left of the pivot}
        Sort(i+1, right);                   {sort to the right of the pivot}
        end; {if}
     end; {Sort}


     procedure Print;

     { Print the array                                              }
     {                                                              }
     { Variables:                                                   }
     {    a - array to print                                        }

     var
        i: integer;

     begin {Print}
     for i := 1 to size do
        writeln(a[i]);
     end; {Print}


  begin
  Fill;
  Sort(1, size);
  Print;
  end.
```

We will use the debugger to see how this program works. Type it in, then run the program once to make sure it is typed in correctly. Now single-step into the program, and step through the Fill procedure. Just before the call to sort, bring up the variables window and type in a[1], a[2], and so forth, so that you can see all of the values in the array. You will have to resize the variables window to see all ten values at one time.

For our first look at the Sort procedure, we will not worry too much about how each statement works.

Instead, let's look closely at what happens on the whole. The Sort procedure is really divided into four distinct steps:

1. Find a pivot value.
2. Put everything smaller than the pivot to the left of the pivot value, and everything larger than the pivot value to the right of the pivot.
3. Sort the values to the left of the pivot.
4. Sort the values to the right of the pivot.

This is a classic example of recursion as we saw it in the last lesson. To understand quick sort, it is very important to look at what happens on one step, not worrying about how we "sort everything to the left of the pivot."

The first few lines of the procedure find the pivot value and move it to the right-hand side of the array, where it is out of the way:

```
i := (left-1)
   + ((right-left+1) div 2);
pivot := a[i];
a[i] := a[right];
a[right] := pivot;
```

It may seem strange to go to all of the work to pluck a pivot from the middle of the array and move it to the right-hand side of the array, but there really is a good reason to do this. The algorithm to shuffle the values smaller than the pivot to the left, and the values larger than the pivot to the right, is a lot simpler and faster if we move the pivot value out of the way. It might seem like a good idea to simply use the right-hand value for the pivot, then. It turns out that this is a rotten idea. If you pick the right-hand value for the pivot, and start with a sorted array, quick sort gives the worst performance possible. In practice, picking the middle element of the array for the pivot works very well.

Getting back to the debugger, step into the Sort procedure. You will have to click on the up arrow key in the variables window to get the array back. Step through the lines that choose a pivot; the array will end up looking like this:

```
10 9  8  7  1  5  4  3  2  6
```

The next step is to shuffle through the array, moving any value smaller than the pivot to the left end of the array, and any value larger than the pivot to the right of the array. To do this, we use two array indices, i and j. They start at opposite ends of the array, working their way towards the middle until they meet (which means we are finished) or they hit a value that is in the wrong spot. If a value is found that is out of place, it is swapped with another value that is out of place on the other end of the array. Step through the procedure, watching how this happens. Here's a summary of what happens to the array; you should see the same thing in the debugger:

```
10 9  8  7  1  5  4  3  2  6
2  9  8  7  1  5  4  3 10  6
2  3  8  7  1  5  4  9 10  6
2  3  4  7  1  5  8  9 10  6
2  3  4  5  1  7  8  9 10  6
```

When we drop out of the while loop, we are almost done. All of the values that are less than the pivot of 6 are in the first 5 elements of the array, while all of the values that are larger than the pivot are in the 4 array elements that follow. The only value that is out of place is the pivot itself. The lines right after the while loop put the pivot value in place. After the pivot is in place, the array looks like this:

```
2  3  4  5  1  6  8  9 10  7
```

At this point, the pivot is in the right place. The values to the left of the pivot need to be sorted, but they have nothing to do with the values to the right of the pivot. The values to the right of the pivot also need to be sorted, but they have nothing to do with the values to the left of the pivot.

Let's face it: quick sort is quite a bit more complicated than the shell sort. Why is it faster? After all, if you count the compares in the while loop that partitions the array, we still end up with about n compares. The trick, though, is that quick sort doesn't have to go through its main loop as many times as the shell sort does. In this example, we've divided the problem in half. Thinking about that in terms of the shell sort, where the worst case sort time is n*(n-1), you can see what an advantage this is. If we are sorting 100 values with the shell sort, the worst case run time is 100*(100-1), or 9900. If we sort 2 arrays, each with 50 elements, though, the run time is proportional to 2*(50*(50-1)), or 4900. You can see that the savings would mount up pretty quickly, since quick sort would divide the 50 element arrays in half, too.

Problem 15.3. How many times does the sort procedure get called in the example shown in this section? (Hint: put a counter in the Sort procedure and run the program.)

Problem 15.4. Find the typical run time for quick sort for arrays that have 2, 3, 4, 5 and 10 elements. Use

the same method that you used in problem 15.2. Count the compares of values in the array, but don't count the compares of array indices. There are three places in the subroutine where you will need to increment the counter: inside each of the short while loops, and right after you exit the large while loop.

How do these values compare to the ones you found in problem 15.2?

How Fast Are They?

All of this mathematical gobbledy-goop about theoretical efficiency may be making your head spin. It can also be taken too far. There are a surprising number of people running around with a degree in computer science who will tell you that quick sort is always faster than a shell sort. Even in theory, this simply isn't true. There are some rare cases where the shell sort will outperform the quick sort, if the values in the array happen to be placed just right.

On average, though, quick sort seems like it should work better than the shell sort. It turns out that this isn't quite true. The shell sort has one advantage over quick sort: it is simpler. Recursive procedure calls take some time; far more time than looping through a while loop. There are also a lot of compares and tests in the Sort procedure that aren't needed in the shell sort. It turns out that the shell sort is actually faster than quick sort for small arrays. Some sophisticated sorting subroutines take advantage of this fact by using quick sort to sort the array until it is divided into small chunks, then using the shell sort, or one of its close relatives, to sort the small pieces.

This is where practice meets theory. A computer scientist who really understands his topic knows all of this, of course. The theoretical run times are very important, but it is also important to keep the overhead in mind. Unfortunately, while a computer scientist can use mathematical proofs to find the theoretical run time for an algorithm, there is no easy way to predict the actual run time. That depends on a lot of variables, like how efficient subroutine calls are (they are more efficient compared to loops on an Apple IIGS, for example, than on an IBM 370 mainframe, which does not have a stack), what kind of information you are comparing (integer compares are much faster than string compares), and how long it takes to swap elements of the array (for arrays of records, the swap may take longer than the compare!).

As a programmer, you need a practical way to compare algorithms in a real setting. ORCA/Pascal has a tool called a profiler which can help you do this. There are several different kinds of profilers, but basically, all of them tell you how long it takes to actually run a particular subroutine. The profiler in ORCA/Pascal uses a Monte-Carlo technique. What that means is that the profiler does a random sample. Every 60th of a second, the profiler looks to see which subroutine it is in. A counter is incremented for that subroutine. After the program finishes, the profiler prints the counters for each subroutine. It also prints the number of times each subroutine was called, and the amount of time spent in each subroutine as a percentage of the overall time to run the program.

Like any statistical technique, the profiler gives the best answers when you give it a lot of information. In the case of the profiler, this means letting the program run for a long time. For example, the results will be more accurate if you run a subroutine 100 times than if you run the subroutine one time.

Using the profiler is very easy. Pull down the Debug menu and select Profile, then run the program like you normally would. Be sure you leave debug code on – the profiler works with the debugger to time your program.

Problem 15.5. Put the shell sort procedure and the quick sort procedure into the same program, and write a main program that will call each of these procedures to sort a copy of the same array. Be sure you use a constant to represent the size of the array. Use a loop in the mail program to repeat the process 10 times, and use a random-number generator to create a new array on each of the 100 loops.

Use the profiler to find the size of the array that will give roughly equal performance for the shell sort and quick sort. For example, if you try an array with 3 elements, you will find that the shell sort is faster. (You can see this by looking at the counter printed by the profiler for each subroutine.) For an array with 25 elements, quick sort is faster. Vary the size of the array until they take about the same amount of time.

Would the results be the same if the array used real values instead of integer values in the array? What about strings? What does this tell you about theoretical run time?

Quick Sort Can Fail!

One little point has been ignored up to now. Quick sort is very fast, especially for large arrays. Quick sort is a little tougher to implement, but you can modify the

Sort procedure from this lesson fairly easily. The big problem with quick sort is that it doesn't always work.

This may come as quite a shock to you. After all, you stepped through the Sort procedure fairly carefully. You saw how it worked. How could it fail?

The answer is that there is nothing wrong with the basic idea behind quick sort. Quick sort will always work unless it runs out of memory. You see, every time you make a procedure call, your program uses a small amount of memory from the stack. The stack is limited in size. By default, programs written in ORCA/Pascal have an 8096 byte stack. You can use the stacksize directive to increase this to about 32K; the exact amount depends on which program launcher you use, what desk accessories you have installed, and what version of the operating system you are using.

In ORCA/Pascal, every procedure call uses 27 byes from the stack frame. If you call a procedure several times from a loop, the procedure uses the same 27 bytes each time you call it, but if a procedure calls itself recursively, each recursive call uses a new chunk of memory. You also have to add the space used by the parameters and local variables. In the case of the Sort procedure, there are 2 parameters and 4 local variables. They use an additional 12 bytes of stack space, so that each call uses 39 bytes. The program has also used some stack space before Sort is called for the first time. For a variety of reasons, there is no good way to tell in advance exactly how much stack space will be used. With the default stack size of 8K, and the Sort procedure we have used in this lesson, it is easy to see that the Sort procedure cannot safely recur more than 207 levels deep. In practice, the value is a little smaller.

If Sort happens to hit a worst-case situation, it will recur as deep as the size of the array. In the best case, Sort will recur ln(n)/ln(2) levels deep, where n is the size of the array. This happens when Sort splits the array exactly in half on each call.

All of this points out that you really have to understand not only the advantages of a particular algorithm, but its disadvantages as well. Any algorithm has to be viewed with a critical eye. Quick sort is a lot faster than the shell sort for large arrays, but the shell sort never fails.

Fortunately, there is a solution to this mess. You can use a counter to keep track of how deep you have recurred in the Sort procedure. If you exceed a preset limit, you can use a shell sort to sort the piece of the array that you are working on, rather than recurring deeper. As you saw in problem 15.5, the shell sort is also more efficient than quick sort for small arrays, so you can also use the shell sort if the array is small, increasing the overall speed of the sort!

Problem 15.6. Modify Sort so it uses a shell sort if the number of array elements to sort is smaller than shellSize, a constant in your Sort procedure. Use the results of problem 15.5 to choose a value for shellSize. Also, add a new parameter to Sort, a counter that is set to 1 when Sort is called from the main program. Inside Sort, increment this value, and pass the new value when Sort is called recursively. This counter will always be the recursion depth. (If you don't see why, implement the subroutine anyway, and then use the debugger to see how count works.) Use the shell sort if the counter exceeds maxDepth, a constant you define. A good value for maxDepth is 100.

Sorting Summary

Sorting has given you your first real taste of writing efficient programs. You can start to see some of the trade-offs that you will have to make when you write programs, as well as some of the techniques you can use to see the impact of these trade-offs.

You probably know that this lesson has only scratched the surface of sorting. Complete books – long ones, at that – have been written on the topic of sorting. The methods covered in this lesson will work in almost any programming situation you are likely to come across, but if you are ever writing a program that is doing a lot of sorting, it would pay to dig into some books to learn about some of the other sorting methods.

Lesson Fifteen
Solutions to Problems

Solution to problem 15.1.

By changing the value of the constant size in the following program, you can find out how many compares the program does for various sized arrays. All of the values do, in fact, match the formula n*(n-1). The values are:

size	number of compares
2	2
3	6
4	12
5	20
10	90

```
{ Count the compares needed to sort a reverse-order array with  }
{ a shell sort.                                                 }

program ShellSort (output);

const
   size = 2;                            {size of the array to sort}

var
   count: integer;                      {number of compares}
   a: array[1..size] of integer;        {array to sort}


   procedure Sort;

   { Sort an array                                              }
   {                                                            }
   { Variables:                                                 }
   {    a - array to sort                                       }

   var
      i: integer;                       {loop variable/array index}
      swap: boolean;                    {was a value swapped?}
      temp: integer;                    {temp; used for swapping}
```

```
   begin {Sort}
   repeat
      swap := false;
      for i := 1 to size-1 do begin
         count := count+1;
         if a[i] > a[i+1] then begin
            temp := a[i];
            a[i] := a[i+1];
            a[i+1] := temp;
            swap := true;
            end; {if}
         end; {for}
   until not swap;
   end; {Sort}


   procedure Fill;

   { Fill an array                                                  }
   {                                                                }
   { Variables:                                                     }
   {    a - array to fill                                           }

   var
      i: integer;                       {loop variable}

   begin {Fill}
   for i := 1 to size do
      a[i] := size+1-i;
   end; {Fill}


begin
count := 0;
Fill;
Sort;
writeln('There were ', count:1, ' compares.');
writeln('n*(n-1) is ', size*(size-1):1);
end.
```

Solution to problem 15.2.

By changing the value of the constant size in the following program, you can find out the average number of compares the program does for various sized arrays when the arrays are filled with random values. The results are:

size	number of compares
2	1.22
3	3.62
4	7.83
5	12.96
10	63.81

```
{ Count the compares needed to sort a pseudo-random array with  }
{ a shell sort.                                                 }

program ShellSort (output);

const
   size = 2;                            {size of the array to sort}
   trials = 100;                        {number of trial runs}

var
   a: array[1..size] of integer;        {array to sort}
   count: integer;                      {number of compares}
   i: 1..trials;                        {loop variable}


   procedure Sort;

   { Sort an array                                              }
   {                                                            }
   { Variables:                                                 }
   {    a - array to sort                                       }

   var
      i: integer;                       {loop variable/array index}
      swap: boolean;                    {was a value swapped?}
      temp: integer;                    {temp; used for swapping}
```

```
begin {Sort}
repeat
   swap := false;
   for i := 1 to size-1 do begin
      count := count+1;
      if a[i] > a[i+1] then begin
         temp := a[i];
         a[i] := a[i+1];
         a[i+1] := temp;
         swap := true;
         end; {if}
      end; {for}
until not swap;
end; {Sort}


function RandomValue(max: integer): integer;

{ Return a pseudo-random number in the range 1..max.         }
{                                                            }
{ Parameters:                                                }
{    max - largest number to return                          }

begin {RandomValue}
RandomValue := (RandomInteger mod max) + 1;
end; {RandomValue}


procedure Fill;

{ Fill an array                                              }
{                                                            }
{ Variables:                                                 }
{    a - array to fill                                       }

var
   i: integer;                       {loop variable}

begin {Fill}
for i := 1 to size do
   a[i] := RandomValue(size);
end; {Fill}
```

```
begin
Seed(2345);                          {initialize the random number generator}
count := 0;                                {no compares, so far}
for i := 1 to trials do begin              {do the trial runs}
   Fill;
   Sort;
   end; {for}
                                           {print the results}
writeln('The average number of compares is ', count/trials:1:2, '.');
end.
```

Solution to problem 15.3.

The quick sort subroutine is called 13 times.

Solution to problem 15.4.

By changing the value of the constant size in the following program, you can find out the average number of compares the program does for various sized arrays when the arrays are filled with random values. The table below shows the results for both this program and the earlier problem that examined the shell sort.

size	shell sort	quick sort
2	1.22	1.00
3	3.62	4.60
4	7.83	8.56
5	12.96	12.71
10	63.81	38.62

```
{ A sample of quick sort.                                              }

program Sort(output);

const
   size = 2;                               {size of the array}
   trials = 100;                           {number of trial runs}

var
   a: array[1..size] of integer;
   count: integer;                         {number of compares}
   i: 1..trials;                           {loop variable}
```

```
function RandomValue(max: integer): integer;

{ Return a pseudo-random number in the range 1..max.         }
{                                                            }
{ Parameters:                                                }
{    max - largest number to return                          }

begin {RandomValue}
RandomValue := (RandomInteger mod max) + 1;
end; {RandomValue}


procedure Fill;

{ Fill an array                                              }
{                                                            }
{ Variables:                                                 }
{    a - array to fill                                       }

var
   i: integer;                       {loop variable}

begin {Fill}
for i := 1 to size do
   a[i] := RandomValue(size);
end; {Fill}


procedure Sort (left,right: integer);

{ Sort an array                                                }
{                                                              }
{ Parameters:                                                  }
{    left - leftmost part of the array to sort                 }
{    right - rightmost part of the array to sort               }
{                                                              }
{ Variables:                                                   }
{    a - array to sort                                         }

var
   i,j: integer;                     {array indices}
   pivot: integer;                   {pivot value}
   temp: integer;                    {used to swap values}
```

```
   begin {Sort}
   if right > left then begin            {quit if there is only 1 element}
      i := (left-1) + ((right-left+1) div 2); {find the pivot index}
      pivot := a[i];                      {put the pivot at the end}
      a[i] := a[right];                   {(remember the pivot, too)}
      a[right] := pivot;
      i := left;                          {set up the start indices}
      j := right-1;
      while i <> j do begin               {partition the array}
         count := count+1;
         while (a[i] <= pivot) and (i <> j) do begin
            i := i+1;
            count := count+1;
            end; {while}
         count := count+1;
         while (a[j] >= pivot) and (i <> j) do begin
            j := j-1;
            count := count+1;
            end; {while}
         temp := a[i];
         a[i] := a[j];
         a[j] := temp;
         end; {while}
      count := count+1;
      if a[i] < pivot then                {find the pivot insert point}
         i := i+1;
      temp := a[i];                       {replace the pivot}
      a[i] := a[right];
      a[right] := temp;
      Sort(left, i-1);                    {sort to the left of the pivot}
      Sort(i+1, right);                   {sort to the right of the pivot}
      end; {if}
   end; {Sort}


begin
Seed(2345);                         {initialize the random number generator}
count := 0;                               {no compares, so far}
for i := 1 to trials do begin             {do the trial runs}
   Fill;
   Sort(1,size);
   end; {for}
                                          {print the results}
writeln('The average number of compares is ', count/trials:1:2, '.');
end.
```

Solution to problem 15.5.

The shell sort and quick sort require almost exactly the same amount of time when the array has 18 elements. Naturally, the speed of a compare and the time it takes to copy the values from one place in the array to another impact this number. By changing the array to an array of real, instead of an array of integer, the break-even point changes from 18 elements to 15 elements.

```
{ Compare the time for a quick sort to the time for a shell    }
{ sort.                                                        }

program Sort(output);

const
   size = 18;                           {size of the array}
   trials = 10;                         {number of trial runs}

var
   a,b: array[1..size] of integer;      {array(s) to sort}
   i: 1..trials;                        {loop variable}


   function RandomValue(max: integer): integer;

   { Return a pseudo-random number in the range 1..max.         }
   {                                                            }
   { Parameters:                                                }
   {    max - largest number to return                          }

   begin {RandomValue}
   RandomValue := (RandomInteger mod max) + 1;
   end; {RandomValue}


   procedure Fill;

   { Fill an array                                              }
   {                                                            }
   { Variables:                                                 }
   {    a - array to fill                                       }

   var
      i: integer;                       {loop variable}

   begin {Fill}
   for i := 1 to size do
      a[i] := RandomValue(size);
   end; {Fill}
```

```
procedure ShellSort;

{ Sort an array                                                    }
{                                                                  }
{ Variables:                                                       }
{    a - array to sort                                             }

var
   i: integer;                        {loop variable/array index}
   swap: boolean;                     {was a value swapped?}
   temp: integer;                     {temp; used for swapping}

begin {ShellSort}
repeat
   swap := false;
   for i := 1 to size-1 do
      if a[i] > a[i+1] then begin
         temp := a[i];
         a[i] := a[i+1];
         a[i+1] := temp;
         swap := true;
         end; {if}
until not swap;
end; {ShellSort}


procedure QuickSort (left,right: integer);

{ Sort an array                                                  }
{                                                                }
{ Parameters:                                                    }
{    left - leftmost part of the array to sort                   }
{    right - rightmost part of the array to sort                 }
{                                                                }
{ Variables:                                                     }
{    a - array to sort                                           }

var
   i,j: integer;                       {array indices}
   pivot: integer;                     {pivot value}
   temp: integer;                      {used to swap values}
```

```
   begin {QuickSort}
   if right > left then begin            {quit if there is only 1 element}
      i := (left-1) + ((right-left+1) div 2); {find the pivot index}
      pivot := a[i];                      {put the pivot at the end}
      a[i] := a[right];                   {(remember the pivot, too)}
      a[right] := pivot;
      i := left;                          {set up the start indices}
      j := right-1;
      while i <> j do begin               {partition the array}
         while (a[i] <= pivot) and (i <> j) do
            i := i+1;
         while (a[j] >= pivot) and (i <> j) do
            j := j-1;
         temp := a[i];
         a[i] := a[j];
         a[j] := temp;
         end; {while}
      if a[i] < pivot then                {find the pivot insert point}
         i := i+1;
      temp := a[i];                       {replace the pivot}
      a[i] := a[right];
      a[right] := temp;
      QuickSort(left, i-1);               {sort to the left of the pivot}
      QuickSort(i+1, right);              {sort to the right of the pivot}
      end; {if}
   end; {QuickSort}


begin
Seed(2345);                         {initialize the random number generator}
for i := 1 to trials do begin             {do the trial runs}
   Fill;
   b := a;
   QuickSort(1,size);
   a := b;
   ShellSort;
   end; {for}
end.
```

Solution to problem 15.6.

```
{ A sample of quick sort.                                          }

program Sort(output);

const
   size = 10;                           {size of the array}

var
   a: array[1..size] of integer;        {array to sort}


   function RandomValue(max: integer): integer;

   { Return a pseudo-random number in the range 1..max.            }
   {                                                               }
   { Parameters:                                                   }
   {    max - largest number to return                             }

   begin {RandomValue}
   RandomValue := (RandomInteger mod max) + 1;
   end; {RandomValue}


   procedure Fill;

   { Fill an array                                                 }
   {                                                               }
   { Variables:                                                    }
   {    a - array to fill                                          }

   var
      i: integer;                       {loop variable}

   begin {Fill}
   for i := 1 to size do
      a[i] := RandomValue(size);
   end; {Fill}
```

```
      procedure Sort (left,right,depth: integer);

      { Sort an array                                                  }
      {                                                                }
      { Parameters:                                                    }
      {    left - leftmost part of the array to sort                   }
      {    right - rightmost part of the array to sort                 }
      {    depth - recursion depth                                     }
      {                                                                }
      { Variables:                                                     }
      {    a - array to sort                                           }

      const
         maxDepth = 100;                    {max recursion depth}
         shellSize = 18;                    {size where quick sort is
better}

      var
         i,j: integer;                      {array indices}
         pivot: integer;                    {pivot value}
         swap: boolean;                     {was a value swapped?}
         temp: integer;                     {used to swap values}

      begin {Sort}
      if ((right-left) < shellSize) or (depth > maxDepth) then begin
         repeat                                {do a shell sort}
            swap := false;
            for i := left to right-1 do
               if a[i] > a[i+1] then begin
                  temp := a[i];
                  a[i] := a[i+1];
                  a[i+1] := temp;
                  swap := true;
                  end; {if}
         until not swap;
         end {if}
      else begin                            {do a quick sort}
         i := (left-1) + ((right-left+1) div 2); {find the pivot index}
         pivot := a[i];                     {put the pivot at the end}
         a[i] := a[right];                  {(remember the pivot, too)}
         a[right] := pivot;
         i := left;                         {set up the start indices}
         j := right-1;
```

```
      while i <> j do begin               {partition the array}
         while (a[i] <= pivot) and (i <> j) do
            i := i+1;
         while (a[j] >= pivot) and (i <> j) do
            j := j-1;
         temp := a[i];
         a[i] := a[j];
         a[j] := temp;
         end; {while}
      if a[i] < pivot then                {find the pivot insert point}
         i := i+1;
      temp := a[i];                       {replace the pivot}
      a[i] := a[right];
      a[right] := temp;
      Sort(left, i-1, depth+1);           {sort to the left of the pivot}
      Sort(i+1, right, depth+1);          {sort to the right of the pivot}
      end; {else}
   end; {Sort}


   procedure Print;

   { Print the array                                              }
   {                                                              }
   { Variables:                                                   }
   {    a - array to print                                        }

   var
      i: integer;

   begin {Print}
   for i := 1 to size do begin
      write(a[i]:4);
      if (i mod 8) = 0 then
         writeln;
      end; {for}
   end; {Print}


begin
Seed(2345);                         {initialize the random number generator}
Fill;                                     {fill the array}
Sort(1,size,1);                           {sort the array}
Print;                                    {print the array}
end.
```

Lesson Sixteen
Searches and Trees

Storing and Accessing Information

The title for this lesson is "Searches and Trees," but a more down-to-earth description would be "better ways to store and find information." Why is this important? Why spend the very last lesson of an introductory programming course on this topic, when there are so many more?

To answer that, let's step back from the trees a bit and look at the forest. Computers are used for a lot of things, but desktop computers are used most often to display information, make calculations, or store and retrieve information. That's a pretty broad statement, but I think it is true. Spread sheets and engineering calculations are obviously applications where we make calculations. Spread sheets, data bases and spelling checkers are examples of applications where one goal is to store or retrieve information. Word processors, page layout programs, paint programs, and some database programs display information. What about an adventure game, though? Most adventure games are really databases inside, concerned with storing and retrieving information about the adventure world. A chess program is calculation intensive. The list goes on and on.

You already know a few basic ways to store and access information. You have used arrays when you knew how much information would be stored in advance, or when you could put a reasonable limit on the amount of information that would be stored. You have used linked lists when the fixed size of an array created problems. You have even used files when the information had to be written to disk.

This lesson concentrates on two basic themes. If the information is stored in an array, linked list, or disk file, how can you find it quickly? And, what are some better ways to store the information so you can find it even quicker?

Sequential Searches

If you have an array, linked list, or file, the simplest way to find a particular piece of information is to start at the beginning and scan through the data structure until you find the entry you want. This is called a sequential search, and it is nothing new to you. You used a sequential search in Lesson 10 to look for a particular name in a linked list of strings. Of course, you can use a sequential search to look for something in a file or array, too. To look for a numeric value in an array of records, a sequential search would look like this:

```
i := 1;
found := false;
repeat
   if a[i].age = 40 then
      found := true
   else
      i := i+1;
until found or (i = maxIndex);
```

On average, you will have to look through half of the information to find the record you want. If the record doesn't exist – if, for example, you are looking for someone who is 40, but there are no 40 year olds in your data base – you will always scan the entire list. A sequential search, then, has a typical run time of O(n/2) if the item you are looking for is found, and a worst case run time of O(n), where n is the number of things to look at.

The Binary Search

The sequential search is a very common kind of search to implement, and it is often the best kind of search to use. In some cases, though, you know more about the information you are searching. For example, one common thing that you might know is that the information is sorted in some kind of order. If you are looking for a man named Smith, for example, you may have ordered your data base so that all of the people are listed in alphabetical order. If you are looking for hospital patients using a Social Security Number, you may be searching a database that is sorted by Social Security Numbers.

When you are searching a list of items that is sorted, and you know in advance how many things are in the array, there is a much better way of finding the information than scanning the array sequentially. The "better way" is called a binary search. The binary search is basically a divide and conquer method, just like quick sort. Binary searches are usually not implemented with recursion, though.

The idea behind a binary search is to start be checking the middle value, rather than the first value.

To see how this works, let's assume we are looking for the number 44 in an array of 100 things. The array is very simple: each value is the same as its index, so a[44] is 44. We'll start by looking at the middle value, a[50]. The value is 50, which is too large. Since the array is sorted, we know that the value we are looking for must be in the portion of the array from a[1] to a[49], assuming it exists at all. We split the array in half again, and so forth. The table below shows our progress.

index	value	result
50	50	too big
25	25	too small
37	37	too small
43	43	too small
46	46	too big
44	44	match

This divide and conquer search is extremely powerful. Its worst case run time is O(ln(n)/ln(2)). For our sample of 100 items, a few seconds with a calculator gives the value of 6.64, which tells us that the search will always succeed after no more than 7 compares. That's a big improvement over the sequential sort, with a typical run time for the same array of 50 – the binary search is 7 times faster. The larger the array, the bigger the difference, too. For an array with 100,000 values, the sequential search will look at an average of 50,000 values. The binary search will only need to look at 17 values! For an array with 100,000 elements, the binary search is nearly 3,000 times faster.

While there are many twists on the sequential search and binary search, these two basic ideas are at the core of many searches in real programs. Whenever the information you need to search is in no particular order, or is in a linked list, the sequential search is a good choice. If the information is sorted, the binary search is the best choice. Most other searching methods depend on organizing the information better to start with.

Problem 16.1. Develop a binary search algorithm, and test in on a simple array. The search should be implemented as a function that returns the index into the array if the value you pass it is found, and zero if it is not. Use an array of 100 integers, with each array element containing an even number. For example, a[1] would be 2, a[2] is 4, and so forth. Test your search by looking for all of the even numbers from 2 to 200. Make sure the search works when values are not found by passing it 0, 202, and 101.

A Cross Reference Program for Pascal

A binary search is an extremely efficient way of looking for a particular piece of information, but it does have one drawback. While it works well for arrays, it is impossible to implement an efficient binary search for a linked list, simply because you can't hop into the middle of the linked list. In Standard Pascal, you have the same problem with files. Fortunately, most Pascal implementations support a procedure called seek, or something like it, that lets you jump to any position you like in a file. The problem with linked lists is a little more severe, though.

The two most common ways of searching records in dynamically allocated memory are called binary trees and hash tables. Both of these methods use a different way of organizing information to make the search faster. We're going to use a Pascal cross reference program to look at binary trees. The purpose of this lesson isn't really to make you write a Pascal cross reference program, so this section gives you one to start with. This Pascal cross reference program uses a linked list for the symbol table.

There are two things that the cross reference program will do that are new to you, so lets start by going over these new techniques in short programs. One of the things that we have always done so far is to prompt the user for a file name. That works, but if you will be using a program a lot, it isn't the easiest way to get a file name. Our cross reference program will be a shell program that reads a file name from the command line. To run the program, you will move to the shell window, and type the name of the program, followed by the name of the file to process. It turns out that reading the command line is very easy; ORCA/Pascal has a built-in procedure called CommandLine that returns the command line. It returns the whole thing, so we do need to skip the name of the program itself. Listing 16.1 is a simple program showing how it's done. The subroutine even asks for a file name if the user forgets to give one.

Listing 16.1

```
{ Read a file name from the command line }

program CLine (input, output);

const
   fNameLength = 64;                        {max length of a file name}

type
   fNameType = string[fNameLength];         {type of a file name}

var
   fName: fNameType;                        {file name}


procedure GetFileName (var fName: fNameType);

{ Read a file name from the command line.  If none is given,   }
{ prompt for one.  Return a null string if no name is given    }
{ when a name is asked for.                                    }
{                                                              }
{ Parameters:                                                  }
{    fName - the file name is returned here                    }

var
   cline: string[255];                      {command line}
   i,j: integer;                            {string indices}


   procedure SkipBlanks;

   { Skip blanks in the command line                           }

   begin {SkipBlanks}
   while (i < length(cline)) and (cline[i] = ' ') do
      i := i+1;
   end; {SkipBlanks}


begin {GetFileName}
CommandLine(cline);                         {read the command line}
i := 1;
SkipBlanks;                                 {skip leading blanks}
while (i < length(cline))                   {skip the program name}
   and (cline[i] <> ' ') do
   i := i+1;
SkipBlanks;                                 {skip to the file name}
```

```
  if i >= length(cline) then begin          {if needed, prompt for a name}
     write('File to cross reference: ');
     readln(cline);
     i := 1;
     SkipBlanks;
     end; {if}

  if i <= length(cline) then begin
     j := 0;                                {read the file name}
     while (j < fNameLength)
        and (i <= length(cline))
        and (cline[i] <> ' ') do begin
        j := j+1;
        fName[j] := cline[i];
        i := i+1;
        end; {while}
     fName[0] := chr(j);                    {set the length}
     SkipBlanks;                            {check for extra input}
     if i <= length(cline) then
        writeln('Extra input ignored');
     end {if}
  else
     fName := '';                           {return a null string}
  end; {GetFileName}


  begin
  GetFileName(fName);                       {Test the procedure}
  writeln('File name = "', fName, '"');
  end.
```

You can run this program just like you do any other program. If you do, it will stop and ask for a file name, just like all of your other programs have. To run it from the shell window, start by turning off debug code. You can't use the debugger with a program that you run from the shell window. With the debug code off, compile the program the way you always do. I called my program FNAME.PAS; if you used a different name, you will need to substitute your program's name in the instructions that follow. Click on the shell window, and type

```
fname myfile
```

and press the RETURN key. The program will print the name of the file. Try it again, but don't give the program a file name. This time, the program, will ask for a file name. Finally, put some extra stuff on the end of the line, like this:

```
fname myfile junk
```

The program still prints the file name, but this time it also prints a warning that there were some extra characters on the command line, and these were ignored.

The other improvement we will make to our standard file processing program is to read the file a different way. In this program, shown in listing 16.2, we will read the file by calling the operating system directly. If you have the reference manual for ProDOS or GS/OS, you can follow along in either of those books. If not, you can treat the ReadFile procedure as a black box. Be sure and name the source file READFILE.PAS, or change the file name in the main part of the program.

Listing 16.2

```
{ Read a file and echo it to the console }

program ReadFile (output);

uses Common, ProDOS, MemoryMgr;

const
   fNameLength = 64;                        {max length of a file name}

type
   fNameType = string[fNameLength];         {type of a file name}
   bytePtr = ^byte;                         {pointer into the file buffer}

var
   fName: fNameType;                        {file name}
   fPtr: bytePtr;                           {file pointer}
   fLength: longint;                        {length of the file}


procedure ReadFile (fName: fNameType; var ptr: bytePtr;
                    var length: longint);

{ Read a file from disk                                              }
{                                                                    }
{ Parameters:                                                        }
{    fName - file name to load                                       }
{    ptr - return a pointer to the 1st char in the file              }
{    length - length of the file, in characters (also bytes)         }

label 99;

var
   openRec: openDCB;                        {ProDOS records}
   eofRec: eofDCB;
   readRec: readWriteDCB;
   closeRec: closeDCB;
   fileHandle: handle;                      {file handle}


   procedure Error(err: integer; close: boolean);

   { Flag a ProDOS disk error                                        }
   {                                                                 }
   { Parameters:                                                     }
   {    err - error number                                           }
   {    close - close the file?                                      }
```

```
      begin {Error}
      writeln('ProDOS error ', toolerror:1, ' processing ', fname);
      if close then
         P16Close(closeRec);
      goto 99;
      end; {Error}


   begin {ReadFile}
   ptr := nil;                                {assume we will get an error}
   length := 0;
   openRec.pathName := @fName;                {open the file}
   openRec.reserved := 0;
   P16Open(openRec);
   if toolerror = 0 then begin
      closeRec.refNum := openRec.refNum;      {get ready for a close}
      eofRec.refNum := openRec.refNum;        {find the length of the file}
      P16Get_EOF(eofRec);
      if toolerror <> 0 then
         Error(toolerror, true);
      length := eofRec.fileSize;
      if length <> 0 then begin
                                              {get some memory file the file}
         fileHandle := NewHandle(length, UserID, $C000, nil);
         if toolerror <> 0 then
            Error(toolerror, true);
         readRec.refNum := eofRec.refNum;     {read the file}
         readRec.dataBuffer := pointer(fileHandle^);
         readRec.requestCount := length;
         P16Read(readRec);
         if toolerror <> 0 then
            Error(toolerror, true);
         ptr := bytePtr(fileHandle^);         {set the file pointer}
         P16Close(closeRec);                  {close the file}
         end; {if}
      end {if}
   else
      Error(toolerror, false);
   99:
   end; {ReadFile}


   begin
   fName := 'readfile.pas';                   {read the file}
   writeln('Reading the file...');
   ReadFile(fName, fPtr, fLength);
   writeln('...done.');
```

```
while fLength <> 0 do begin                {echo the file}
   if fPtr^ = 13 then
      writeln
   else
      write(chr(fPtr^));
   fPtr := pointer(ord4(fPtr)+1);
   fLength := fLength-1;
   end; {while}
end.
```

This program dynamically allocates the file, returning a pointer to the file buffer. The file buffer itself can be any size. Since characters in a text file use one byte of memory, but Pascal char variables use two bytes of memory, the pointer is a pointer to byte, not a pointer to char. Naturally, this means that we will have to use the chr function to convert the byte to a character, but that is a small price to pay for the extra efficiency. A minor point about the chr function is that it doesn't actually generate any extra code in ORCA/Pascal, or most other Pascal compilers, either. The purpose of the chr function is to tell the compiler that you really do want to treat some integer value as a character, and that you know what you are doing.

It's fair to ask why we are going to all of this trouble when Pascal's text files can do all of this for us. The answer is speed. By calling the operating system directly, we dramatically improve the performance of the program. You might want to try using a text file to compare the amount of time it takes to read a file each way.

The last step is to tie all of this together into a Pascal cross reference generator. This program uses the same scanning techniques that we discussed back in Lesson 13, although a few new features have been added to handle comments and to keep track of line numbers. Once a token is found, the program searches for the token in a symbol table that is a simple linked list. If the token does not exist, the search routine creates a new entry in the symbol table. Finally, the program places the line number where the token was found in a linked list in the symbol table. While both the symbol table itself and the line numbers are simple linked lists, this is the first time you have seen a linked list where each element of the linked list point to yet another linked list. There are no new concepts involved in creating linked lists this way, but the details are interesting enough to make it worth looking at the program shown in listing 16.3 carefully.

If you have time, you might want to try writing this program on your own before typing in the version you see here.

Listing 16.3

```
{ XREF                                                          }
{                                                               }
{ This program generates a cross reference of a Pascal program, }
{ showing where any symbol is used.  To use XREF, start by      }
{ selecting the shell window.  Type                             }
{                                                               }
{    xref filename                                              }
{                                                               }
{ where filename is the name of the program you want to cross-  }
{ reference.                                                    }

program ReadFile (input, output);

uses Common, ProDOS, MemoryMgr;
```

```
const
   fNameLength = 64;                        {max length of a file name}
   symbolLength = 80;                       {max length of a symbol}

type
   bytePtr = ^byte;                         {pointer into the file buffer}
   fNameType = string[fNameLength];         {type of a file name}

   tokenType = string[symbolLength];        {type of a symbol name}

   linePtr = ^lineRecord;                   {line number list}
   lineRecord = record
      next: linePtr;
      number: integer;
      end;

   symbolPtr = ^symbolRecord;               {symbol table entry}
   symbolRecord = record
      next: symbolPtr;
      symbol: string[symbolLength];
      lines: linePtr;
      end;

var
   ch: char;                                {current character}
   fLength: longint;                        {length of the file}
   fName: fNameType;                        {file name}
   fPtr: bytePtr;                           {file pointer}
   lineNumber: integer;                     {current line number}
   symbols: symbolPtr;                      {symbol table}
   token: tokenType;                        {current token}
   tokenLine: integer;                      {line number at start of token}


procedure GetFileName (var fName: fNameType);

{ Read a file name from the command line.  If none is given,   }
{ prompt for one.  Return a null string if no name is given    }
{ when a name is asked for.                                    }
{                                                              }
{ Parameters:                                                  }
{    fName - the file name is returned here                    }

var
   cline: string[255];                      {command line}
   i,j: integer;                            {string indices}
```

```
   procedure SkipBlanks;

   { Skip blanks in the command line                          }

   begin {SkipBlanks}
   while (i < length(cline)) and (cline[i] = ' ') do
      i := i+1;
   end; {SkipBlanks}


begin {GetFileName}
CommandLine(cline);                        {read the command line}
i := 1;
SkipBlanks;                                {skip leading blanks}
while (i < length(cline))                  {skip the program name}
   and (cline[i] <> ' ') do
   i := i+1;
SkipBlanks;                                {skip to the file name}
if i >= length(cline) then begin           {if needed, prompt for a name}
   write('File to cross reference: ');
   readln(cline);
   i := 1;
   SkipBlanks;
   end; {if}

if i <= length(cline) then begin
   j := 0;                                 {read the file name}
   while (j < fNameLength)
      and (i <= length(cline))
      and (cline[i] <> ' ') do begin
      j := j+1;
      fName[j] := cline[i];
      i := i+1;
      end; {while}
   fName[0] := chr(j);                     {set the length}
   SkipBlanks;                             {check for extra input}
   if i <= length(cline) then
      writeln('Extra input ignored');
   end {if}
else
   fName := '';                            {return a null string}
end; {GetFileName}
```

```
procedure ReadFile (var fName: fNameType; var ptr: bytePtr;
                    var length: longint);

{ Read a file from disk                                          }
{                                                                }
{ Parameters:                                                    }
{    fName - file name to load                                   }
{    ptr - return a pointer to the 1st char in the file          }
{    length - length of the file, in characters (also bytes)     }

label 99;

var
   openRec: openDCB;                      {ProDOS records}
   eofRec: eofDCB;
   readRec: readWriteDCB;
   closeRec: closeDCB;
   fileHandle: handle;                    {file handle}


   procedure Error(err: integer; close: boolean);

   { Flag a ProDOS disk error                                    }
   {                                                             }
   { Parameters:                                                 }
   {    err - error number                                       }
   {    close - close the file?                                  }

   begin {Error}
   writeln('ProDOS error ', toolerror:1, ' processing ', fname);
   if close then
      P16Close(closeRec);
   goto 99;
   end; {Error}


begin {ReadFile}
ptr := nil;                               {assume we will get an error}
length := 0;
openRec.pathName := @fname;               {open the file}
openRec.reserved := 0;
P16Open(openRec);
if toolerror = 0 then begin
   closeRec.refNum := openRec.refNum;     {get ready for a close}
   eofRec.refNum := openRec.refNum;       {find the length of the file}
   P16Get_EOF(eofRec);
   if toolerror <> 0 then
      Error(toolerror, true);
```

```
      length := eofRec.fileSize;
      if length <> 0 then begin
                                          {get some memory file the file}
         fileHandle := NewHandle(length, UserID, $C000, nil);
         if toolerror <> 0 then
            Error(toolerror, true);
         readRec.refNum := eofRec.refNum;    {read the file}
         readRec.dataBuffer := pointer(fileHandle^);
         readRec.requestCount := length;
         P16Read(readRec);
         if toolerror <> 0 then
            Error(toolerror, true);
         ptr := bytePtr(fileHandle^);        {set the file pointer}
         P16Close(closeRec);                 {close the file}
         end; {if}
      end {if}
   else
      Error(toolerror, false);
   99:
   end; {ReadFile}


   procedure GetToken;

   { Read a word from the source file                                  }
   {                                                                   }
   { Variables:                                                        }
   {    fPtr - pointer to the next character in the file               }
   {    fLength - number of characters left in the file                }
   {    lineNumber - current line number                               }
   {    token - string read                                            }
   {    tokenLine - line number at the start of the token              }

   var
      len: integer;                        {length of the string}
      tPtr: ^byte;                         {used for 1 char look-ahead}


      procedure NextCh;

      { Get the next character from the file, skipping comments      }
```

```
      procedure GetCh;

      { get the next character from the file                    }

      begin {GetCh}
      if fLength = 0 then
         ch := chr(0)
      else begin
         fPtr := pointer(ord4(fPtr)+1);
         fLength := fLength-1;
         ch := chr(fPtr^);
         end; {else}
      end; {GetCh}


      procedure SkipComment;

      { Skip comments in the program                            }

      begin {SkipComment}
      repeat
         GetCh;
         if ch = '*' then begin
            GetCh;
            if ch = ')' then
               ch := '}';
            end; {if}
      until ch in ['}', chr(0)];
      end; {SkipComment}


   begin {NextCh}
   GetCh;                              {get the next character}
   if ch = chr(13) then                {handle the end of a line}
      lineNumber := lineNumber+1
   else if ch = '{' then               {skip comments}
      SkipComment
   else if ch = '(' then begin
      tPtr := pointer(ord4(fPtr)+1);
      if chr(tPtr^) = '*' then begin
         GetCh;
         GetCh;
         SkipComment;
         end; {if}
      end; {else if}
   end; {NextCh}
```

```
   begin {GetToken}
   len := 0;                                  {no token, yet}
   if fLength <> 0 then begin
                                              {skip to the next token}
      while not (ch in ['a'..'z','A'..'Z','_',chr(0)]) do
         NextCh;
      tokenLine := lineNumber;                {record the line number}
                                              {record the token}
      while ch in ['a'..'z','A'..'Z','_','0'..'9'] do begin
         if len < symbolLength then begin
            len := len+1;
            token[len] := ch;
            end; {if}
         NextCh;
         end; {while}
      end; {if}
   token[0] := chr(len);                      {set the length of the string}
   end; {GetToken}


   procedure Insert;

   { Insert a symbol use in the symbol table.  If the symbol does    }
   { not exist, create a new entry.                                  }
   {                                                                 }
   { Variables:                                                      }
   {    tokenLine - line number at the start of the token            }
   {    token - symbol to insert                                     }
   {    symbols - pointer to the first entry in the symbol table     }

   label 1;

   var
      lPtr: linePtr;                          {current line number pointer}
      sPtr: symbolPtr;                        {current symbol pointer}


      function Match (s1, s2: tokenType): boolean;

      { Do a case insensitive compare of two strings for equality    }
      {                                                              }
      { Parameters:                                                  }
      {    s1,s2 - strings to compare                                }
      {                                                              }
      { Returns:                                                     }
      {    True if the strings are equal, else false                 }

      label 1;

      var
```

```
        i: integer;                          {loop variable}
        len: integer;                        {length of the first string}


        function ToUpper(ch: char): char;

        { Return the uppercase equivalent of ch                    }

        begin {ToUpper}
        if ch in ['a'..'z'] then
           ch := chr(ord(ch)-ord('a')+ord('A'));
        ToUpper := ch;
        end; {ToUpper}


     begin {Match}
     len := length(s1);
     if len = length(s2) then begin
        Match := true;
        for i := 1 to len do
           if ToUpper(s1[i]) <> ToUpper(s2[i]) then begin
              Match := false;
              goto 1;
              end; {if}
        end {if}
     else
        Match := false;
     1:
     end; {Match}


  begin {Insert}
  sPtr := symbols;                           {try to find the symbol}
  while sPtr <> nil do begin
     if Match(token, sPtr^.symbol) then
        goto 1;
     sPtr := sPtr^.next;
     end; {while}
  new(sPtr);                                {none exists: create a new entry}
  sPtr^.next := symbols;
  symbols := sPtr;
  sPtr^.symbol := token;
  sPtr^.lines := nil;
  1:
  new(lPtr);                                 {enter the line number}
  lPtr^.next := sPtr^.lines;
  sPtr^.lines := lPtr;
  lPtr^.number := tokenLine;
  end; {Insert}
```

```
procedure PrintSymbols;

{ Print the symbols found and line numbers                          }
{                                                                   }
{ Variables:                                                        }
{    symbols - pointer to the first entry in the symbol table       }

var
   sPtr: symbolPtr;                        {current symbol pointer}


   procedure PrintNumber (nPtr: linePtr);

   { Recursively print the line numbers in reverse order            }
   {                                                                }
   { Parameters:                                                    }
   {    nPtr - pointer to the remainder of the line number list     }

   begin {PrintNumber}
   if nPtr <> nil then begin
      PrintNumber(nPtr^.next);
      write(nPtr^.number:1, ' ');
      end; {if}
   end; {PrintNumber}


begin {PrintSymbols}
sPtr := symbols;
while sPtr <> nil do begin
   write(sPtr^.symbol:16, '  ');
   PrintNumber(sPtr^.lines);
   writeln;
   sPtr := sPtr^.next;
   end; {while}
end; {PrintSymbols}


begin
symbols := nil;                         {nothing in the symbol table}
lineNumber := 1;                        {first line}
ch := ' ';                              {initialize the scanner}
GetFileName(fName);                     {get the file name}
if length(fName) <> 0 then begin
   ReadFile(fName, fPtr, fLength);      {read the file}
   fPtr := pointer(ord4(fPtr)-1);       {scanner will skip to 1st char}
   if fPtr <> nil then begin
      repeat                            {find all of the symbols}
         GetToken;
         if length(token) <> 0 then
```

```
            Insert;
        until length(token) = 0;
        PrintSymbols;                          {print the symbols}
        end; {if}
     end; {if}
  end.
```

There are a couple of problems with the Pascal cross reference program you just tried. The most subtle problem is that it is a lot slower than it could be, simply because it takes so darn long to deal with a sequential linked list. This is the main problem we will try to solve in the next section. The program is even slower if you forget to turn off debug code after the program is finished. The most obvious problem, though, is that the symbols are printed in the reverse order of when they are first seen in the program. It would be a lot more convenient if they were printed in alphabetical order. We will take care of this problem as a side effect of getting rid of the linked list. The last problem is that any sequence of alphanumeric characters is treated as a symbol. Your program reports all of the places where you used the reserved word end, for example. That one you will solve yourself a bit later, as one of the problems.

The Binary Tree

The major problem with a linked list is the same as the major problem with a sequential search: the program has to scan through an average of half of the list to find a particular entry. If the entry doesn't exist, the program scans through the entire list. A binary tree is another way of handling dynamically allocated records that essentially does the same thing for linked lists that the binary search did for searches. At each level, the tree divides the search in half.

The way this works is to include two pointers to another record in each record, rather than one. In a linked list, each record has a pointer we have called next that points to the next record in the list. In a binary tree, each record has two pointers, which we will call left and right. If we look at a particular record, and the one we want is "smaller" that the one we are looking at, we follow the left link. If the one we want is "larger" than the one we are looking at, we follow the right link.

To see how this works, we'll use a few short programs. The first task is to learn to add a new item to a binary tree. This is a little harder than it was for a linked list, but the same basic ideas are involved. The program below reads strings from the keyboard and adds them to a binary tree.

Listing 16.4

```
{ Create a binary tree from keyboard strings }

program BinaryTree (input, output);

type
   treePtr = ^treeRecord;                  {tree entry}
   treeRecord = record
      left,right: treePtr;
      str: string[20];
      end;

var
   tree, tPtr: treePtr;                    {top of the tree}
   str: string[20];                        {work string}


   procedure Add(var ptr: treePtr; rec: treePtr);
```

```
    { Add a record to the tree                                    }
    {                                                             }
    { Parameters:                                                 }
    {    ptr - next node in the tree                              }
    {    rec - record to add to the tree                          }

    begin {Add}
    if ptr = nil then
       ptr := rec
    else if rec^.str < ptr^.str then
       Add(ptr^.left, rec)
    else if rec^.str > ptr^.str then
       Add(ptr^.right, rec);
    end; {Add}


begin
tree := nil;                          {nothing in the tree}
repeat
   write('string: ');                 {get a string}
   readln(str);
   if length(str) <> 0 then begin
      new(tPtr);                      {create a new record}
      tPtr^.left := nil;
      tPtr^.right := nil;
      tPtr^.str := str;
      Add(tree, tPtr);                {add it to the tree}
      end; {if}
until length(str) = 0;                {loop until no string is given}
end.
```

Looking at this program, one of the first things you might notice is that we are using a recursive subroutine again. Just as with any situation where recursion is useful, we can look at the tree as a piecemeal problem. Let's look at an example to see how this will work. As an example, let's place four states in the tree. We'll use Main, Oregon, Texas and Colorado for our states. Main is simple: we create a new record, set left and right to nil, record the string, and call Add. The procedure Add sees that ptr, which is a var parameter, is nil, and records rec there. The effect on the global variables is to assign tPtr to tree, so tree now points to the first record in our list, Main. Symbolically, we write the tree like this:

```
Main
```

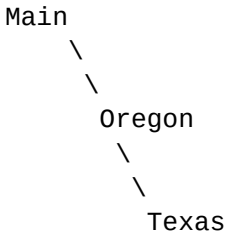
Well, there isn't much there, yet, so our meager tree doesn't look very impressive. Adding Oregon shapes things up a bit, though. This time, when we call Add, the procedure sees that ptr is not nil, and checks to see if Oregon is less than Main. It isn't, so it moves on to the next check to be sure that Oregon is greater than main. It is, but let's stop for a moment and consider what would happen if it wasn't. The only way a name could fail both checks is if it matched the name in ptr^.str exactly. The series of checks, then, prevent duplicates. You can have duplicates in a binary tree, but your search has to take it into account if you do. We don't need them.

At this point, Add calls itself, passing ptr^.right as the new top of the tree. Ptr^.right is nil, so rec is added as the so-called "right child" of Main. It makes as much sense to call Oregon a branch of Main, but for historical reasons, we refer to Oregon as the right child of Main, and Main as the parent of Oregon. Our tree looks like this, now:

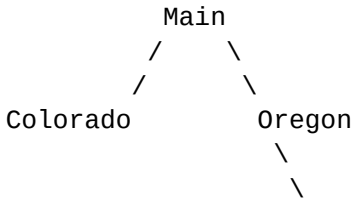
```
Main
    \
     \
      Oregon
```

Notice how recursion handled the problem of tracing the tree fairly neatly. Once we decided that the top node existed, and which way to go, we called Add again, treating ptr^.right as a brand-new tree, which in a sense it is. If you recall, when recursion was first introduced, I said that the way to think about recursion was to think about one part of the problem at a time. We used that method to solve the Tower of Hanoi problem, where we conceptually moved an entire pile of disks, rather than thinking about the problem as moving individual disks. The same idea cropped up when we used recursion for quick sort, where the subroutine split the problem in half and called itself to solve each half. Here we see the same idea again: Add decides which half of the tree is the important part, then calls itself, precessing the appropriate half of the tree as a new tree.

The next state to add is Texas, which makes two recursive calls, getting tacked onto the tree as the right child of Oregon. Follow through the code, writing the steps down on paper if necessary, to see how this is done.

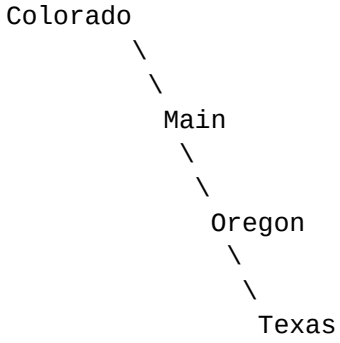


The last state is Colorado. Since Colorado is less than Main, it is added as the left child of Main. Our final tree looks like this:



Texas

By now, you may have noticed one of the problems with binary trees. To keep the search time to a minimum, you want the tree to be balanced. What that means is that, when you start at the top, the top element of the tree is also the middle element, so that the compare splits the tree in half. In this example, if we had started with Colorado, adding the states in alphabetical order, we would have ended up with a pretty poor excuse for a tree:



You can add a new record to the tree and shuffle the tree around at the same time to make sure it stays balanced. We won't cover how, since it involves some fairly advanced pointer manipulation. In practical situations, it also isn't necessary to create a perfectly balanced tree. If records are added to the tree in a fairly random manner, the savings of using a tree instead of a linked list are still enormous. Whether the extra effort involved in balancing the tree is worth the time depends on how often the tree will be searched and how random the records are. In our application, they are fairly random.

Searching a binary tree is pretty trivial once you know how to create one. After all, adding a new record to the tree searches the tree as a side effect! Listing 16.5 shows a function, based on the Add procedure, that will search the tree, returning a pointer to the correct record, or nil if the record does not exist:

Listing 16.5

```
function Search(ptr: treePtr; str: string20): treePtr;

{ Search the tree for a record                                    }
{                                                                 }
{ Parameters:                                                     }
{    ptr - next node in the tree                                  }
{    str - value to search for                                    }
{                                                                 }
{ Returns:                                                        }
{    Pointer to the matching record; nil if the record does       }
{    not exist.                                                   }

begin {Search}
if ptr = nil then
   Search := nil
else if rec^.str < str then
   Search := Search(ptr^.left, rec)
else if rec^.str > str then
   Search := Search(ptr^.right, rec)
else
   Search := ptr;
end; {Search}
```

This is one of those subroutines that you might struggle for a long time to come up with on your own, but is so simple that once you see it, it is easy to understand and remember. Trace through the subroutine, looking for Oregon and Indiana if you aren't sure how it works.

subroutine that will trace through the tree, doing something in order. In our case, we want to print the symbols found in the Pascal program. Listing 16.6 shows a simple print subroutine that prints the states in our example program; the subroutine in the Pascal cross reference program will have exactly the same structure.

Listing 16.6

```
procedure Print(ptr: treePtr);

{ Print the tree                                                  }
{                                                                 }
{ Parameters:                                                     }
{    ptr - next node in the tree                                  }

begin {Print}
if ptr <> nil then begin
   Print(ptr^.left);
   writeln(ptr^.str);
   Print(ptr^.right);
   end; {if}
end; {Print}
```

Finally, we come to a subject that impacts directly on our cross-reference program. Using a method called recursive tree traversal, we can write a very simple

Notice how, once again, recursion simplifies the problem. At any particular place in the tree, we need to print all of the names that come before the one we are

working on first, so we call Print to do that. Next, we need to print the record we are working on. Finally, we print all of the names that come after the one we just printed. The initial check to make sure ptr is not nil keeps us from stepping off of the "end" of the tree.

Problem 16.2. Add the print subroutine to the BinaryTree sample program. Try the program with a variety of names, using the debugger to see how the program works if you are not sure, yet.

Problem 16.3. Change the XREF program so it builds a binary tree for the symbol table instead of a linked list. The easy way to do this is to use the Insert subroutine to insert each symbol in the program into the symbol table. Because of the way the insert subroutine is written, if the symbol already exists, a new symbol is not created. You then call the Search subroutine to find the correct entry in the symbol table (which must exist, since you just created one if there wasn't one already), and enter the appropriate line number.

A more challenging, and more efficient way to implement the program is to combine the Search and Insert subroutines, creating a function that returns a pointer to the correct entry in the symbol table, creating one if one did not already exist. This is the method the solution uses.

In either case, printing the symbol table is a simple matter of modifying the Print subroutine from the text.

Problem 16.4. Add a new check to the XREF program that checks to see if the symbol just found is a reserved word in Pascal. You can find a list of the reserved words in your Pascal reference manual.

An easy way to handle reserved words is to add a new flag to each symbol table entry that tells if the entry is a reserved word. If you find a reserved word, you skip adding the line number to the line number list. When printing the symbol table, you again skip reserved words.

Creating the reserved word list in the first place is a little tedious. You will need a subroutine that calls Insert for each of the reserved words. There is an optimum order to add the reserved words. See if you can figure it out by thinking about the way trees are created, referring to the example where the names of four states were entered into a tree.

Ruffles and Flourishes

Well, a few weeks ago, you couldn't spell recursive tree traversal, and now you know what it is. Not bad. Let me be the first to congratulate you on joining the ranks of real programmers, who do it with bytes and nibbles.

Of course, as I have pointed out so many times that you may be sick of hearing it, programming is a skill. Like all skills, the more you practice, the easier it gets. There are also a lot more things to learn about programming. Where you go from here depends on your own interests.

ORCA/Pascal meets both the ANSI and ISO standard for Pascal. With the exception of a few features that deal specifically with the Apple IIGS, all of the things you have learned in this course will work on any ISO or ANSI standard Pascal, and on most other Pascals, too. The reverse is also important: any book that uses ISO or ANSI Pascal (often called Standard Pascal) will work with ORCA/Pascal. If you would like to learn more about programming, there are many fine books that meet this requirement. Here are a few of my favorites that I recommend highly. You may have to special order a few, but they are worth it.

The Byte Book of Pascal
Blaise Liffick
BYTE/McGraw Hill

This is a fun book to curl up with in front of your computer. It's a collection of articles about Pascal from the pages of early issues of Byte magazine, back when it was a programmer's magazine. Articles include such things as a small Pascal compiler written in BASIC, an APL interpreter and a Chess program written in Pascal, and several articles about the language itself.

Oh! Pascal!
Doug Cooper and Michael Clancy
W.W. Norton & Company

Oh! Pascal! is a college level Pascal textbook, and, I think, one of the best. It is also available in a shorted form, bound with Standard Pascal User Reference Manual. This combined book is called Condensed Pascal. I would highly recommend that you follow the Pascal course you just finished with something like this book. It will review most of the material you just covered, many times with a different twist. If you read this book now, it will help to firm up the basic concepts of programming.

Pascal Programs for Scientists and Engineers
Alan R. Miller
Sybex

There are a lot of books like this one that will introduce the art of mathematical computing, something we have not covered much in this course. If you are interested in fractals, three-dimensional graphics, numerical analysis, or any other math oriented subject, look for this book or one of its many cousins.

Algorithms
Robert Sedgewick
Addison-Wesley Publishing Company

Do you want to find out how to sort a linked list, encrypt a data file, or parse a program? This fine book is a collection of useful algorithms that you can use in your own programs. It was the source for the version of quick sort used in this course, for example.

Algorithms+Data Structures = Programs
Niklaus Wirth
Prentice-Hall

This classic book is a great introduction to intermediate techniques in computer science. It only has five chapters: Fundamental Data Structures, Sorting, Recursive Algorithms, Dynamic Information Structures and Language Structures and Compilers. These chapters give you a basic understanding of data structures that can improve your programming skills enormously. I think every programmer should have a copy of this book. Incidentally, Niklaus Wirth is a name you should be familiar with: he is the creator of the Pascal programming language.

All of these books cover general programming concepts or Pascal programming techniques that are useful on any machine. If you would like to learn to program the toolbox, writing desktop programs with pull down menus and so forth, you need to study a different set of books. First and foremost are the three volumes of the Apple IIGS Toolbox Reference Manual, written by Apple's staff of programmers and published by Addison-Wesley (except for Volume 3, which is currently available only from APDA, Apple's distribution arm for programming products). There are no good introductory books for learning to program the toolbox that use Pascal. To learn the toolbox, I would suggest reading the front sections of the chapters in the reference manuals, reading magazine articles, and asking a lot of questions on one of the major on-line services, like America Online or GEnie.

Whatever you decide to do from here, I hope you enjoyed the course, and learned a few things along the way. Once again, congratulations on completing the course!

Lesson Sixteen
Solutions to Problems

Solution to problem 16.1.

```
{ Implement a binary search                                          }

program List (output);

const
   size = 100;                          {size of the array}

var
   a: array[1..size] of integer;        {array to search}


   function Find (val: integer): integer;

   { Find the index of a matching array element                      }
   {                                                                 }
   { Parameters:                                                     }
   {    val - value to find                                          }
   {                                                                 }
   { Returns:                                                        }
   {    Returns the array index of the matching array value.         }
   {    If there are no matching array values, a zero is             }
   {    returned.                                                    }
   {                                                                 }
   { Variables:                                                      }
   {    a - array to search                                          }

   var
      left,right,middle: integer;       {array indices}

   begin {Find}
   left := 1;
   right := size;
   repeat
      middle := (left+right) div 2;
      if val < a[middle] then
         right := middle-1
      else
         left := middle+1;
   until (val = a[middle]) or (left > right);
```

```
   if val = a[middle] then
      Find := middle
   else
      Find := 0;
   end; {Find}


   procedure Fill;

   { Fill the array                                              }
   {                                                             }
   { Variables:                                                  }
   {    a - array to search                                      }

   var
      i: integer;                       {loop index}

   begin {Fill}
   for i := 1 to size do
      a[i] := i*2;
   end; {Fill}


   procedure Test;

   { Test the Find procedure                                     }

   var
      i: integer;                       {loop index}

   begin {Test}
   for i := 1 to size do        {check to be sure each value can be found}
      if a[Find(i*2)] <> i*2 then
         writeln('Failed to find ', i*2:1);
   if Find(0) <> 0 then    {check to be sure missing values are not found}
      writeln('Reported "found" for Find(0)');
   if Find(101) <> 0 then
      writeln('Reported "found" for Find(101)');
   if Find(202) <> 0 then
      writeln('Reported "found" for Find(202)');
   end; {Test}


begin
Fill;                                {fill the array}
Test;                                {test the Find procedure}
end.
```

Solution to problem 16.2.

```
{ Create a binary tree from keyboard strings }

program BinaryTree (input, output);

type
   treePtr = ^treeRecord;                {tree entry}
   treeRecord = record
      left,right: treePtr;
      str: string[20];
      end;

var
   tree, tPtr: treePtr;                  {top of the tree}
   str: string[20];                      {work string}


   procedure Add(var ptr: treePtr; rec: treePtr);

   { Add a record to the tree                                        }
   {                                                                 }
   { Parameters:                                                     }
   {    ptr - next node in the tree                                  }
   {    rec - record to add to the tree                              }

   begin {Add}
   if ptr = nil then
      ptr := rec
   else if rec^.str < ptr^.str then
      Add(ptr^.left, rec)
   else if rec^.str > ptr^.str then
      Add(ptr^.right, rec);
   end; {Add}


   procedure Print(ptr: treePtr);

   { Print the tree                                                  }
   {                                                                 }
   { Parameters:                                                     }
   {    ptr - next node in the tree                                  }

   begin {Print}
   if ptr <> nil then begin
      Print(ptr^.left);
      writeln(ptr^.str);
      Print(ptr^.right);
      end; {if}
   end; {Print}
```

```
begin
tree := nil;                              {nothing in the tree}
repeat
   write('string: ');                     {get a string}
   readln(str);
   if length(str) <> 0 then begin
      new(tPtr);                          {create a new record}
      tPtr^.left := nil;
      tPtr^.right := nil;
      tPtr^.str := str;
      if tree = nil then                  {add it to the tree}
         tree := tPtr
      else
         Add(tree, tPtr);
      end; {if}
until length(str) = 0;                    {loop until no string is given}
Print(tree);                              {print the tree}
end.
```

Solution to problem 16.3.

```
{ XREF                                                              }
{                                                                   }
{ This program generates a cross reference of a Pascal program,    }
{ showing where any symbol is used.  To use XREF, start by         }
{ selecting the shell window.  Type                                 }
{                                                                   }
{    xref filename                                                  }
{                                                                   }
{ where filename is the name of the program you want to cross-      }
{ reference.                                                        }

program ReadFile (input, output);

uses Common, ProDOS, MemoryMgr;

const
   fNameLength = 64;                      {max length of a file name}
   symbolLength = 80;                     {max length of a symbol}

type
   bytePtr = ^byte;                       {pointer into the file buffer}
   fNameType = string[fNameLength];       {type of a file name}

   tokenType = string[symbolLength];      {type of a symbol name}
```

```
   linePtr = ^lineRecord;                 {line number list}
   lineRecord = record
      next: linePtr;
      number: integer;
      end;

   symbolPtr = ^symbolRecord;             {symbol table entry}
   symbolRecord = record
      left,right: symbolPtr;
      symbol: string[symbolLength];
      lines: linePtr;
      end;

var
   ch: char;                              {current character}
   fLength: longint;                      {length of the file}
   fName: fNameType;                      {file name}
   fPtr: bytePtr;                         {file pointer}
   lineNumber: integer;                   {current line number}
   symbols: symbolPtr;                    {symbol table}
   token: tokenType;                      {current token}
   tokenLine: integer;                    {line number at start of token}


procedure GetFileName (var fName: fNameType);

{ Read a file name from the command line.  If none is given,   }
{ prompt for one.  Return a null string if no name is given    }
{ when a name is asked for.                                    }
{                                                              }
{ Parameters:                                                  }
{    fName - the file name is returned here                    }

var
   cline: string[255];                    {command line}
   i,j: integer;                          {string indices}


   procedure SkipBlanks;

   { Skip blanks in the command line                           }

   begin {SkipBlanks}
   while (i < length(cline)) and (cline[i] = ' ') do
      i := i+1;
   end; {SkipBlanks}
```

```
begin {GetFileName}
CommandLine(cline);                          {read the command line}
i := 1;
SkipBlanks;                                  {skip leading blanks}
while (i < length(cline))                    {skip the program name}
   and (cline[i] <> ' ') do
   i := i+1;
SkipBlanks;                                  {skip to the file name}
if i >= length(cline) then begin             {if needed, prompt for a name}
   write('File to cross reference: ');
   readln(cline);
   i := 1;
   SkipBlanks;
   end; {if}

if i <= length(cline) then begin
   j := 0;                                   {read the file name}
   while (j < fNameLength)
      and (i <= length(cline))
      and (cline[i] <> ' ') do begin
      j := j+1;
      fName[j] := cline[i];
      i := i+1;
      end; {while}
   fName[0] := chr(j);                       {set the length}
   SkipBlanks;                               {check for extra input}
   if i <= length(cline) then
      writeln('Extra input ignored');
   end {if}
else
   fName := '';                              {return a null string}
end; {GetFileName}


procedure ReadFile (var fName: fNameType; var ptr: bytePtr;
                    var flength: longint);

{ Read a file from disk                                              }
{                                                                    }
{ Parameters:                                                        }
{    fName - file name to load                                       }
{    ptr - return a pointer to the 1st char in the file              }
{    flength - length of the file, in characters (also bytes)        }

label 99;
```

```
var
   openRec: openDCB;                        {ProDOS records}
   eofRec: eofDCB;
   readRec: readWriteDCB;
   closeRec: closeDCB;
   fileHandle: handle;                      {file handle}


   procedure Error(err: integer; close: boolean);

   { Flag a ProDOS disk error                                    }
   {                                                             }
   { Parameters:                                                 }
   {    err - error number                                       }
   {    close - close the file?                                  }

   begin {Error}
   writeln('ProDOS error ', toolerror:1, ' processing ', fname);
   if close then
      P16Close(closeRec);
   goto 99;
   end; {Error}


begin {ReadFile}
ptr := nil;                                 {assume we will get an error}
flength := 0;
openRec.pathName := @fName;                 {open the file}
openRec.reserved := 0;
P16Open(openRec);
if toolerror = 0 then begin
   closeRec.refNum := openRec.refNum;       {get ready for a close}
   eofRec.refNum := openRec.refNum;         {find the length of the file}
   P16Get_EOF(eofRec);
   if toolerror <> 0 then
      Error(toolerror, true);
   flength := eofRec.fileSize;
```

```
   if flength <> 0 then begin
                                        {get some memory file the file}
      fileHandle := NewHandle(flength, UserID, $C000, nil);
      if toolerror <> 0 then
         Error(toolerror, true);
      readRec.refNum := eofRec.refNum;    {read the file}
      readRec.dataBuffer := pointer(fileHandle^);
      readRec.requestCount := flength;
      P16Read(readRec);
      if toolerror <> 0 then
         Error(toolerror, true);
      ptr := bytePtr(fileHandle^);        {set the file pointer}
      P16Close(closeRec);                 {close the file}
      end; {if}
   end {if}
else
   Error(toolerror, false);
99:
end; {ReadFile}


procedure GetToken;

{ Read a word from the source file                                  }
{                                                                   }
{ Variables:                                                        }
{    fPtr - pointer to the next character in the file               }
{    fLength - number of characters left in the file                }
{    lineNumber - current line number                               }
{    token - string read                                            }
{    tokenLine - line number at the start of the token              }

var
   len: integer;                        {length of the string}
   tPtr: ^byte;                         {used for 1 char look-ahead}


   procedure NextCh;

   { Get the next character from the file, skipping comments      }
```

```
   procedure GetCh;

   { get the next character from the file                    }

   begin {GetCh}
   if fLength = 0 then
      ch := chr(0)
   else begin
      fPtr := pointer(ord4(fPtr)+1);
      fLength := fLength-1;
      ch := chr(fPtr^);
      end; {else}
   end; {GetCh}


   procedure SkipComment;

   { Skip comments in the program                            }

   begin {SkipComment}
   repeat
      GetCh;
      if ch = '*' then begin
         GetCh;
         if ch = ')' then
            ch := '}';
         end; {if}
   until ch in ['}', chr(0)];
   end; {SkipComment}


begin {NextCh}
GetCh;                                 {get the next character}
if ch = chr(13) then                   {handle the end of a line}
   lineNumber := lineNumber+1
else if ch = '{' then                  {skip comments}
   SkipComment
else if ch = '(' then begin
   tPtr := pointer(ord4(fPtr)+1);
   if chr(tPtr^) = '*' then begin
      GetCh;
      GetCh;
      SkipComment;
      end; {if}
   end; {else if}
end; {NextCh}
```

```
begin {GetToken}
len := 0;                                  {no token, yet}
if fLength <> 0 then begin
                                           {skip to the next token}
   while not (ch in ['a'..'z','A'..'Z','_',chr(0)]) do
      NextCh;
   tokenLine := lineNumber;                {record the line number}
                                           {record the token}
   while ch in ['a'..'z','A'..'Z','_','0'..'9'] do begin
      if len < symbolLength then begin
         len := len+1;
         token[len] := ch;
         end; {if}
      NextCh;
      end; {while}
   end; {if}
token[0] := chr(len);                      {set the length of the string}
end; {GetToken}


procedure Insert;

{ Insert a symbol use in the symbol table.  If the symbol does    }
{ not exist, create a new entry.                                  }
{                                                                 }
{ Variables:                                                      }
{    tokenLine - line number at the start of the token            }
{    token - symbol to insert                                     }
{    symbols - pointer to the first entry in the symbol table     }

var
   lPtr: linePtr;                          {current line number pointer}
   sPtr: symbolPtr;                        {current symbol pointer}
```

```
function FindSymbol (var ptr: symbolPtr): symbolPtr;

{ Find a symbol; create one if none exists                    }
{                                                             }
{ Parameters:                                                 }
{    ptr - pointer to the next symbol to check                }
{                                                             }
{ Returns:                                                    }
{    a pointer to the symbol table entry                      }
{                                                             }
{ Variables:                                                  }
{    token - name of the symbol to find                       }

var
   test: integer;                    {result of the string compare}
   tPtr: symbolPtr;                  {symbol to return}


   function Compare (s1, s2: tokenType): integer;

   { Do a case insensitive compare of two strings             }
   {                                                          }
   { Parameters:                                              }
   {    s1,s2 - strings to compare                            }
   {                                                          }
   { Returns:                                                 }
   {    -1   if s1 < s2                                       }
   {     0   if s1 = s2                                       }
   {     1   if s1 > s2                                       }

   label 1;

   var
      c1, c2: char;                  {uppercase characters}
      i: integer;                    {loop variable}
      len: integer;                  {length of the smallest string}


      function ToUpper(ch: char): char;

      { Return the uppercase equivalent of ch                 }

      begin {ToUpper}
      if ch in ['a'..'z'] then
         ch := chr(ord(ch)-ord('a')+ord('A'));
      ToUpper := ch;
      end; {ToUpper}
```

```
   begin {Compare}
   len := length(s1);           {find the length of the shorter string}
   if length(s2) < len then
      len := length(s2);
   for i := 1 to len do begin          {check the characters}
      c1 := ToUpper(s1[i]);
      c2 := ToUpper(s2[i]);
      if c1 < c2 then begin
         Compare := -1;                {handle s1 < s2}
         goto 1;
         end {if}
      else if c1 > c2 then begin
         Compare := 1;                 {handle s1 > s2}
         goto 1;
         end; {else if}
      end; {for}
   if length(s1) < length(s2) then   {characters match - check lengths}
      Compare := -1
   else if length(s1) > length(s2) then
      Compare := 1
   else
      Compare := 0;
   1:
   end; {Compare}


begin {FindSymbol}
if ptr = nil then begin
   new(tPtr);                         {none exists: create a new entry}
   with tPtr^ do begin
      left := nil;
      right := nil;
      symbol := token;
      lines := nil;
      end; {with}
   ptr := tPtr;
   end {if}
else begin
                                      {compare the token to this entry}
   test := Compare(token, ptr^.symbol);
   if test < 0 then
      tPtr := FindSymbol(ptr^.left)    {too big: move left}
   else if test > 0 then
      tPtr := FindSymbol(ptr^.right)   {too small: move right}
   else
      tPtr := ptr;                     {else just right: return ptr}
   end; {else}
FindSymbol := tPtr;
end; {FindSymbol}
```

```
begin {Insert}
sPtr := FindSymbol(symbols);              {find or create a symbol}
new(lPtr);                                {enter the line number}
lPtr^.next := sPtr^.lines;
sPtr^.lines := lPtr;
lPtr^.number := tokenLine;
end; {Insert}


procedure PrintSymbols (ptr: symbolPtr);

{ Print the symbols found and line numbers                        }
{                                                                 }
{ Parameters:                                                     }
{    ptr - pointer to the current entry in the symbol table       }


   procedure PrintNumber (nPtr: linePtr);

   { Recursively print the line numbers in reverse order          }
   {                                                              }
   { Parameters:                                                  }
   {    nPtr - pointer to the remainder of the line number list   }

   begin {PrintNumber}
   if nPtr <> nil then begin
      PrintNumber(nPtr^.next);
      write(nPtr^.number:1, ' ');
      end; {if}
   end; {PrintNumber}


begin {PrintSymbols}
if ptr <> nil then begin
   PrintSymbols(ptr^.left);
   write(ptr^.symbol:16, '  ');
   PrintNumber(ptr^.lines);
   writeln;
   PrintSymbols(ptr^.right);
   end; {if}
end; {PrintSymbols}
```

```
begin
symbols := nil;                          {nothing in the symbol table}
lineNumber := 1;                         {first line}
ch := ' ';                               {initialize the scanner}
GetFileName(fName);                      {get the file name}
if length(fName) <> 0 then begin
   ReadFile(fName, fPtr, fLength);       {read the file}
   fPtr := pointer(ord4(fPtr)-1);        {scanner will skip to 1st char}
   if fPtr <> nil then begin
      repeat                             {find all of the symbols}
         GetToken;
         if length(token) <> 0 then
            Insert;
      until length(token) = 0;
      PrintSymbols(symbols);             {print the symbols}
      end; {if}
   end; {if}
end.
```

Solution to problem 16.4.

```
{ XREF                                                              }
{                                                                   }
{ This program generates a cross reference of a Pascal program,    }
{ showing where any symbol is used.  To use XREF, start by         }
{ selecting the shell window.  Type                                 }
{                                                                   }
{    xref filename                                                  }
{                                                                   }
{ where filename is the name of the program you want to cross-      }
{ reference.                                                        }

program ReadFile (input, output);

uses Common, ProDOS, MemoryMgr;

const
   fNameLength = 64;                     {max length of a file name}
   symbolLength = 80;                    {max length of a symbol}

type
   bytePtr = ^byte;                      {pointer into the file buffer}
   fNameType = string[fNameLength];      {type of a file name}

   tokenType = string[symbolLength];     {type of a symbol name}
```

```
   linePtr = ^lineRecord;                 {line number list}
   lineRecord = record
      next: linePtr;
      number: integer;
      end;

   symbolPtr = ^symbolRecord;             {symbol table entry}
   symbolRecord = record
      left,right: symbolPtr;
      symbol: string[symbolLength];
      reservedWord: boolean;
      lines: linePtr;
      end;

var
   ch: char;                              {current character}
   fLength: longint;                      {length of the file}
   fName: fNameType;                      {file name}
   fPtr: bytePtr;                         {file pointer}
   lineNumber: integer;                   {current line number}
   symbols: symbolPtr;                    {symbol table}
   token: tokenType;                      {current token}
   tokenLine: integer;                    {line number at start of token}


procedure GetFileName (var fName: fNameType);

{ Read a file name from the command line.  If none is given,   }
{ prompt for one.  Return a null string if no name is given    }
{ when a name is asked for.                                    }
{                                                              }
{ Parameters:                                                  }
{    fName - the file name is returned here                    }

var
   cline: string[255];                    {command line}
   i,j: integer;                          {string indices}


   procedure SkipBlanks;

   { Skip blanks in the command line                           }

   begin {SkipBlanks}
   while (i < length(cline)) and (cline[i] = ' ') do
      i := i+1;
   end; {SkipBlanks}
```

```
begin {GetFileName}
CommandLine(cline);                          {read the command line}
i := 1;
SkipBlanks;                                  {skip leading blanks}
while (i < length(cline))                    {skip the program name}
   and (cline[i] <> ' ') do
   i := i+1;
SkipBlanks;                                  {skip to the file name}
if i >= length(cline) then begin             {if needed, prompt for a name}
   write('File to cross reference: ');
   readln(cline);
   i := 1;
   SkipBlanks;
   end; {if}

if i <= length(cline) then begin
   j := 0;                                   {read the file name}
   while (j < fNameLength)
      and (i <= length(cline))
      and (cline[i] <> ' ') do begin
      j := j+1;
      fName[j] := cline[i];
      i := i+1;
      end; {while}
   fName[0] := chr(j);                       {set the length}
   SkipBlanks;                               {check for extra input}
   if i <= length(cline) then
      writeln('Extra input ignored');
   end {if}
else
   fName := '';                              {return a null string}
end; {GetFileName}


procedure ReadFile (var fName: fNameType; var ptr: bytePtr;
                    var flength: longint);

{ Read a file from disk                                              }
{                                                                    }
{ Parameters:                                                        }
{    fName - file name to load                                       }
{    ptr - return a pointer to the 1st char in the file              }
{    flength - length of the file, in characters (also bytes)        }

label 99;
```

```
var
   openRec: openDCB;                        {ProDOS records}
   eofRec: eofDCB;
   readRec: readWriteDCB;
   closeRec: closeDCB;
   fileHandle: handle;                      {file handle}


   procedure Error(err: integer; close: boolean);

   { Flag a ProDOS disk error                                    }
   {                                                             }
   { Parameters:                                                 }
   {    err - error number                                       }
   {    close - close the file?                                  }

   begin {Error}
   writeln('ProDOS error ', toolerror:1, ' processing ', fname);
   if close then
      P16Close(closeRec);
   goto 99;
   end; {Error}


begin {ReadFile}
ptr := nil;                                 {assume we will get an error}
flength := 0;
openRec.pathName := @fName;                 {open the file}
openRec.reserved := 0;
P16Open(openRec);
if toolerror = 0 then begin
   closeRec.refNum := openRec.refNum;       {get ready for a close}
   eofRec.refNum := openRec.refNum;         {find the length of the file}
   P16Get_EOF(eofRec);
   if toolerror <> 0 then
      Error(toolerror, true);
   flength := eofRec.fileSize;
   if flength <> 0 then begin
                                            {get some memory file the file}
      fileHandle := NewHandle(flength, UserID, $C000, nil);
      if toolerror <> 0 then
         Error(toolerror, true);
      readRec.refNum := eofRec.refNum;      {read the file}
      readRec.dataBuffer := pointer(fileHandle^);
      readRec.requestCount := flength;
      P16Read(readRec);
      if toolerror <> 0 then
         Error(toolerror, true);
      ptr := bytePtr(fileHandle^);          {set the file pointer}
      P16Close(closeRec);                   {close the file}
```

```
      end; {if}
   end {if}
else
   Error(toolerror, false);
99:
end; {ReadFile}


procedure GetToken;

{ Read a word from the source file                                    }
{                                                                     }
{ Variables:                                                          }
{    fPtr - pointer to the next character in the file                 }
{    fLength - number of characters left in the file                  }
{    lineNumber - current line number                                 }
{    token - string read                                              }
{    tokenLine - line number at the start of the token                }

var
   len: integer;                          {length of the string}
   tPtr: ^byte;                           {used for 1 char look-ahead}


   procedure NextCh;

   { Get the next character from the file, skipping comments      }


      procedure GetCh;

      { get the next character from the file                     }

      begin {GetCh}
      if fLength = 0 then
         ch := chr(0)
      else begin
         fPtr := pointer(ord4(fPtr)+1);
         fLength := fLength-1;
         ch := chr(fPtr^);
         end; {else}
      end; {GetCh}
```

```
   procedure SkipComment;

   { Skip comments in the program                              }

   begin {SkipComment}
   repeat
      GetCh;
      if ch = '*' then begin
         GetCh;
         if ch = ')' then
            ch := '}';
         end; {if}
   until ch in ['}', chr(0)];
   end; {SkipComment}


begin {NextCh}
GetCh;                                  {get the next character}
if ch = chr(13) then                    {handle the end of a line}
   lineNumber := lineNumber+1
else if ch = '{' then                   {skip comments}
   SkipComment
else if ch = '(' then begin
   tPtr := pointer(ord4(fPtr)+1);
   if chr(tPtr^) = '*' then begin
      GetCh;
      GetCh;
      SkipComment;
      end; {if}
   end; {else if}
end; {NextCh}
```

```
begin {GetToken}
len := 0;                                  {no token, yet}
if fLength <> 0 then begin
                                           {skip to the next token}
   while not (ch in ['a'..'z','A'..'Z','_',chr(0)]) do
      NextCh;
   tokenLine := lineNumber;                {record the line number}
                                           {record the token}
   while ch in ['a'..'z','A'..'Z','_','0'..'9'] do begin
      if len < symbolLength then begin
         len := len+1;
         token[len] := ch;
         end; {if}
      NextCh;
      end; {while}
   end; {if}
token[0] := chr(len);                      {set the length of the string}
end; {GetToken}


procedure Insert (reserved: boolean);

{ Insert a symbol use in the symbol table.  If the symbol does    }
{ not exist, create a new entry.                                  }
{                                                                 }
{ Parameters:                                                     }
{    reserved - is this a reserved word? (else a symbol)          }
{                                                                 }
{ Variables:                                                      }
{    tokenLine - line number at the start of the token            }
{    token - symbol to insert                                     }
{    symbols - pointer to the first entry in the symbol table     }

var
   lPtr: linePtr;                          {current line number pointer}
   sPtr: symbolPtr;                        {current symbol pointer}
```

```
function FindSymbol (var ptr: symbolPtr): symbolPtr;

{ Find a symbol; create one if none exists                    }
{                                                             }
{ Parameters:                                                 }
{    ptr - pointer to the next symbol to check                }
{                                                             }
{ Returns:                                                    }
{    a pointer to the symbol table entry                      }
{                                                             }
{ Variables:                                                  }
{    token - name of the symbol to find                       }

var
   test: integer;                    {result of the string compare}
   tPtr: symbolPtr;                  {symbol to return}


   function Compare (s1, s2: tokenType): integer;

   { Do a case insensitive compare of two strings             }
   {                                                          }
   { Parameters:                                              }
   {    s1,s2 - strings to compare                            }
   {                                                          }
   { Returns:                                                 }
   {    -1   if s1 < s2                                       }
   {     0   if s1 = s2                                       }
   {     1   if s1 > s2                                       }

   label 1;

   var
      c1, c2: char;                  {uppercase characters}
      i: integer;                    {loop variable}
      len: integer;                  {length of the smallest string}


      function ToUpper(ch: char): char;

      { Return the uppercase equivalent of ch                 }

      begin {ToUpper}
      if ch in ['a'..'z'] then
         ch := chr(ord(ch)-ord('a')+ord('A'));
      ToUpper := ch;
      end; {ToUpper}
```

```
   begin {Compare}
   len := length(s1);           {find the length of the shorter string}
   if length(s2) < len then
      len := length(s2);
   for i := 1 to len do begin          {check the characters}
      c1 := ToUpper(s1[i]);
      c2 := ToUpper(s2[i]);
      if c1 < c2 then begin
         Compare := -1;                {handle s1 < s2}
         goto 1;
         end {if}
      else if c1 > c2 then begin
         Compare := 1;                 {handle s1 > s2}
         goto 1;
         end; {else if}
      end; {for}
   if length(s1) < length(s2) then   {characters match - check lengths}
      Compare := -1
   else if length(s1) > length(s2) then
      Compare := 1
   else
      Compare := 0;
   1:
   end; {Compare}


begin {FindSymbol}
if ptr = nil then begin
   new(tPtr);                         {none exists: create a new entry}
   with tPtr^ do begin
      left := nil;
      right := nil;
      symbol := token;
      lines := nil;
      reservedWord := reserved;
      end; {with}
   ptr := tPtr;
   end {if}
else begin
                                      {compare the token to this entry}
   test := Compare(token, ptr^.symbol);
   if test < 0 then
      tPtr := FindSymbol(ptr^.left)    {too big: move left}
   else if test > 0 then
      tPtr := FindSymbol(ptr^.right)   {too small: move right}
   else
      tPtr := ptr;                     {else just right: return ptr}
   end; {else}
FindSymbol := tPtr;
end; {FindSymbol}
```

```
begin {Insert}
sPtr := FindSymbol(symbols);              {find or create a symbol}
if not reserved then begin                {for symbols, record the line}
   new(lPtr);                             {enter the line number}
   lPtr^.next := sPtr^.lines;
   sPtr^.lines := lPtr;
   lPtr^.number := tokenLine;
   end; {if}
end; {Insert}


procedure PrintSymbols (ptr: symbolPtr);

{ Print the symbols found and line numbers                        }
{                                                                 }
{ Parameters:                                                     }
{    ptr - pointer to the current entry in the symbol table       }


   procedure PrintNumber (nPtr: linePtr);

   { Recursively print the line numbers in reverse order          }
   {                                                              }
   { Parameters:                                                  }
   {    nPtr - pointer to the remainder of the line number list   }

   begin {PrintNumber}
   if nPtr <> nil then begin
      PrintNumber(nPtr^.next);
      write(nPtr^.number:1, ' ');
      end; {if}
   end; {PrintNumber}


begin {PrintSymbols}
if ptr <> nil then begin
   PrintSymbols(ptr^.left);
   if not ptr^.reservedWord then begin
      write(ptr^.symbol:16, '  ');
      PrintNumber(ptr^.lines);
      writeln;
      end; {if}
   PrintSymbols(ptr^.right);
   end; {if}
end; {PrintSymbols}
```

```
procedure InitReservedWords;

{ Record the reserved words in the symbol table.                }
{                                                               }
{ Note: The reserved words are entered in a strange order to    }
{ balance the binary tree used by Insert.                       }

begin {InitReservedWords}
token := 'nil';                 Insert(true);
token := 'file';                Insert(true);
token := 'div';                 Insert(true);
token := 'begin';               Insert(true);
token := 'array';               Insert(true);
token := 'and';                 Insert(true);
token := 'const';               Insert(true);
token := 'case';                Insert(true);
token := 'else';                Insert(true);
token := 'downto';              Insert(true);
token := 'do';                  Insert(true);
token := 'end';                 Insert(true);
token := 'implementation';      Insert(true);
token := 'goto';                Insert(true);
token := 'function';            Insert(true);
token := 'for';                 Insert(true);
token := 'if';                  Insert(true);
token := 'label';               Insert(true);
token := 'interface';           Insert(true);
token := 'in';                  Insert(true);
token := 'mod';                 Insert(true);
token := 'string';              Insert(true);
token := 'procedure';           Insert(true);
token := 'or';                  Insert(true);
token := 'of';                  Insert(true);
token := 'not';                 Insert(true);
token := 'packed';              Insert(true);
token := 'otherwise';           Insert(true);
token := 'repeat';              Insert(true);
token := 'record';              Insert(true);
token := 'program';             Insert(true);
token := 'set';                 Insert(true);
token := 'until';               Insert(true);
token := 'type';                Insert(true);
token := 'to';                  Insert(true);
token := 'then';                Insert(true);
token := 'univ';                Insert(true);
token := 'unit';                Insert(true);
token := 'while';               Insert(true);
token := 'var';                 Insert(true);
token := 'uses';                Insert(true);
token := 'with';                Insert(true);
```

```
end; {InitReservedWords}


begin
symbols := nil;                          {nothing in the symbol table}
lineNumber := 1;                         {first line}
ch := ' ';                               {initialize the scanner}
InitReservedWords;                       {record the reserved words}
GetFileName(fName);                      {get the file name}
if length(fName) <> 0 then begin
   ReadFile(fName, fPtr, fLength);       {read the file}
   fPtr := pointer(ord4(fPtr)-1);        {scanner will skip to 1st char}
   if fPtr <> nil then begin
      repeat                             {find all of the symbols}
         GetToken;
         if length(token) <> 0 then
            Insert(false);
      until length(token) = 0;
      PrintSymbols(symbols);             {print the symbols}
      end; {if}
   end; {if}
end.
```