Learn to Program in C

by Mike Westerfield

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 todays most popular programming languages, C. 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 C.

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. Pascal is an excellent language. It is very popular in educational settings, and has been used to write a large number of applications. C, however, is even more popular. C tends to give the programmer more freedom that Pascal by not being so strict with type checking, and always providing a way around the checks that exist. These features have made C very popular with professional programmers. Since the pros use C, it is only natural that everyone else wants to use it, too.

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/C. If you decide to use a different C, 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/C.

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/C. 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/C. 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/C 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 C 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/C 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/C 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/C program disk. As it is shipped, the ORCA/C Desktop System disk, which is the one you will have in the disk drive while writing programs, has about 140K 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/C 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/C disks, which is why you should never run from the original disks.

We will use ORCA/C 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/C 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/C 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/C 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 C 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/C 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 CC. If you pull it down again, you will see that CC has a check mark beside it. (It is traditional to call a C compiler CC, rather than just C.)

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.

```
#include <stdio.h>

void main(void)

{
printf("Hello, world.\n");
}
```

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.

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.

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.CC. Finally, click on the save button. Note that the name of the source window changed.

When ORCA/C runs your program, it will 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

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.

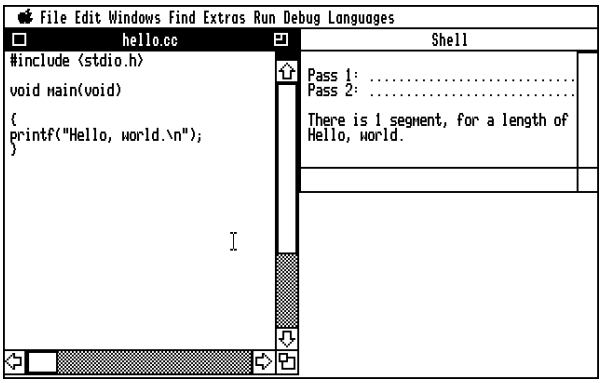
To move the window, you drag the title bar. Start by positioning the mouse cursor in the middle of the window's title bar, over the name of the window. Next, press and hold the mouse button, dragging the outline to the new location, and finally, let up on the mouse button.

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/C manual that describes the desktop editor to learn some of these shortcuts.

right hand side of your program window under the b in the Debug menu.



Well, the time has come to actually run your first C 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/C disk, you will see a message printed by the linker as it links the program. The compiler doesn't print anything unless it finds an error. 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!

Sizing the Program Window

If you forget to resize your program window, it may not look like anything worked. Once the program finishes executing, though, you can resize the program window to uncover the shell window. You will find that the program ran correctly, but that you just didn't see the text until you resized the window.

Program Didn't Run?

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/C 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/C and executed. If you encounter a problem, the chances are very, very high that you did something wrong. It isn't unheard of to find a bug in the development software, but we get far more bug reports that turn out to be programmer errors than bug reports that actually turn out to be a bug in ORCA/C.

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.

```
#include <stdio.h>

void main(void)

{
printf("Hello, world.\n");
}
```

One of the things about C that you will become very familiar with in this course is the C library. The C library is a huge collection of subroutines that do a wide variety of things for you. The size and standardization

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 C, or Pascal, 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 C, 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 Pascal instead of C, you need a different compiler.

One of the advantages of a language like C is that it does not depend on the computer you are using. On the Apple IIGS, you are using ORCA/C, which translates C programs into machine language programs the Apple IIGS can run. A Macintosh computer cannot run this program, but you could use MPW C on the Macintosh. MPW C 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.

of the C library is one of the major strengths of the C programming language.

The C compiler doesn't know about the C library. In our C programs, we have to tell the C compiler about the library. The first line of our program tells the compiler about the subroutines in a library called stdio, which is short for Standard Input/Output. This library contains most of the text formatting, console input and output, and disk input and output subroutines. In particular, it defines a subroutine called printf, which can be used to print text.

Technically, #include is part of the preprocessor. That's an idea you will become more familiar with a little later in the course. This preprocessor command tells the compiler to find another C program file, and to process that file first. The C compiler goes to the library folder and looks for a file called stdio.h; you can look at the file, too. To see what the compiler saw, pull down the File menu and select Open. Click on the disk button until you see the ORCA.C disk. Open the LIBRARIES folder, then open the ORCACDEFS folder that you find inside of the LIBRARIES folder. Inside of this folder, towards the bottom of a long list of files, you will find one called STDIO.H. Open that file. What you will see is an impressive collection of gibberish. There is no particular reason for you to understand it all at the moment. The point is that the #include preprocessor command tells the compiler to process a simple text file. You can put anything you like in the text file. Later, you will learn how to use this capability to split large programs up into several small, manageable files.

Right below the #include command is a blank line. The blank line is for our convenience; the C 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.

C programs are made up of a series of variable declarations, preprocessor commands and functions. These can be freely mixed, with just a few rules we will learn later. Our simple program doesn't have any variable declarations. It does, however, have a function.

```
void main(void)

{
printf("Hello, world.\n");
}
```

Each function in a C program has several distinct parts. The first part, which we generally put on a single line, tells the compiler the name of the function, what it returns, and what we need to pass to the function when it is called. The name of this function is main. Every C program must have a function called main; this is the first function executed when the program starts. This function doesn't return anything. We tell the compiler that it doesn't return anything by putting void right before the name of the function. Right after the name of the function, we put any parameters that will be passed to the function when it is called. Again, void is used to tell the compiler that there aren't any

parameters. This is the simplest kind of function: it does something, but no parameters are passed, and it doesn't return anything.

The last three lines,

```
{
printf("Hello, world.\n");
}
```

are the lines that tell the compiler what the function actually does. The { character and } character mark the start and end of the statements in the function. Between these characters is the only line in the program that actually does anything, the call to the printf function. printf is a library function, defined in the stdio library. The characters that we want the program to print are placed in a string constant. In C, string constants are enclosed in quote marks. You can put any character you like in a string constant, although there are some tricks to using a few of them; we'll look at the tricks in the next section. The string constant is a parameter that is passed to the printf function. Like all parameters, it is enclosed in parenthesis, and comes right after the name of the function. The line ends with a semicolon. All C statements end with a semicolon; this tells the compiler where the statement ends.

Escape Sequences

You may have noticed one slightly strange thing about the program. Supposedly, printf prints all of the characters in the string constant. On the other hand, you can see two characters, \ and n, at the end of the string constant – these were not printed.

The \ character has special meaning inside of a string constant, where it is used to mark the start of something called an escape sequence. Escape sequences give you a way to put characters into the program that you normally can't see, and in many cases, can't even type. The \n escape sequence puts a line feed character in the string. When the line feed character is printed, it starts a new line in the shell window.

To see how this works, make a small change to your program and run it again. The change is to replace the space between Hello and world with a \n escape sequence. Your program will look like this after you make the change:

```
#include <stdio.h>

void main(void)

{
printf("Hello,\nworld.\n");
}
```

Run this program, and look at the results in the shell window. This time, Hello and word appear on different lines.

There are quite a few escape sequences in C. As you go through the course, you will learn about most of the escape sequences, and it will make more sense to see them as they are used, rather than simply looking at a list of the various codes. You can find a list of the escape sequences in your compiler reference manual, though. C is a large language, and to use it effectively, you will need to learn to use your compiler reference manual well, too. Take a moment and find the escape sequences in the compiler reference manual. If you are curious (and I hope you are!) you should read the section of the reference manual that describes them. Some of the information may already make sense to you, while some will not; that's to be expected. Like I said, you will see most of the escape sequences in real programs in this course, where they will probably make a lot more sense.

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.CC.

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 .CC onto the end.

Reserved Words

Like English text, C programs are made up of words and punctuation marks. You have already seen a few of the punctuation marks, including the # character, which starts a preprocessor command; the ; character, which marks the end of a line; the (and) characters, which mark the start and end of a parameter list; and the { and } characters, which mark the start and end of the statements in a function. The words in the first program were include, void, main, and printf. You also saw a quoted string, which is called a string constant in C.

There is one major difference between words in English and words in C, though. Some words in C can

only be used for very specific purposes. These are called reserved words. In our first program, void was the only reserved word we used. As you learn to program in C, you will eventually encounter all of the reserved words, and find out what they are. You will also start to use words to define your own functions and variables, though, and you have to know what the reserved words are so you won't accidentally use one of the reserved words. While you don't need to memorize them, you do need to know they exist. If you run into a strange error trying to get a program to work, you can refer back to this list of reserved words to see if misusing a reserved word is the cause of the problem. (The same list is in your compiler reference manual, too.)

Reserved Words in ORCA/C

```
auto      asm       break     case
char      comp      const     continue
default   do        double    else
enum      extended  extern    float
for       goto      if        inline
int       long      pascal    register
return    segment   short     signed
sizeof    static    struct    switch
typedef   union     unsigned  void
volitile  while
```

Case Sensitivity

C is case sensitive. That means that there is a difference between two words that are spelled the same, but have different cases in the letters. For example, Void is not a reserved word in C, but void is. The name of the function that always gets executed first is main, with no capitol letters. Main is not the same thing, and the compiler will get confused if you try to use it for the name of the main function.

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.CC. When you type the program, pay special attention to the \ characters in the strings, and what follows. You already knew about the \n escape sequence; it starts a new line in the shell window. The \" escape sequence is new, though. We needed a quote mark in our string, but a quote mark is used to mark the end of a string, too. The \" escape sequence puts a quote mark in the string without marking the end of the string.

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

Listing 1.1

```
#include <stdio.h>

void main(void)

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

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 do, instead of what you want them to do. When your program does something different than you 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 do. As you learn more about C, 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 1.2. Write a program that prints your name and address. Print the address on separate lines, just as you would on an envelope.

Problem 1.3. 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, C 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.

```
#include <quickdraw.h>

void main(void)

{
SetPenMode(0);
SetSolidPenPat(0);
SetPenSize(3,1);
MoveTo(10,10);
LineTo(90,10);
LineTo(90,40);
LineTo(10,40);
LineTo(10,10);
}
```

Type in this program, and save it as SQAURE.CC, 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.)

You may have noticed that the #include statement changed in this program. In our text programs, we were using a library function called printf, which is defined in the stdio.h header file. We're not using printf in this program, so we don't need that header file. On the other hand, we are using several functions from the QuickDraw toolbox; these are defined in the

quickdraw.h header file. Remembering which header files you need can quickly become a daunting task. Later, we'll learn a few tricks to make the compiler help you out.

When you compile the program, it will take quite a bit longer than when you compiled the text programs. This may seem strange. After all, your program isn't much bigger than the ones you have written so far. The difference is caused by the quickdraw.h header file, which is much longer than the stdio.h header file.

Looking at the lines in the main function, the first three lines 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 1.4. 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 1.5. 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.

If you are having trouble with your program, be sure and use the debugger. That way, you can see the lines drawn one at a time. If only one line is out of place, this will help you nail the offending line.

Problem 1.6. Modify the program in problem 1.5 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 1.7. 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.

```
#include <stdio.h>

void main(void)

{
printf("Hello, Mike.\n");
}
```

Solution to problem 1.2.

```
#include <stdio.h>

void main(void)

{
printf("Mike Westerfield\n");
printf("4700 Irving Blvd. NW, Suite 207\n");
printf("Albuquerque, NM  87114\n");
}
```

Solution to problem 1.3.

```
#include <stdio.h>

void main(void)

{
printf("*   *  ***  *   * *****\n");
printf("** **   *   *  *  *    \n");
printf("* * *   *   * *   *    \n");
printf("*   *   *   **    ***  \n");
printf("*   *   *   * *   *    \n");
printf("*   *   *   *  *  *    \n");
printf("*   *  ***  *   * *****\n");
}
```

Solution to problem 1.4.

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.

Here is the program to draw the purple square:

```
#include <quickdraw.h>

void main(void)

{
SetPenMode(0);
SetSolidPenPat(1);
SetPenSize(3,1);
MoveTo(10,10);
LineTo(90,10);
LineTo(90,40);
LineTo(10,40);
LineTo(10,10);
}
```

Here is the program to draw the green square:

```
#include <quickdraw.h>

void main(void)

{
SetPenMode(0);
SetSolidPenPat(2);
SetPenSize(3,1);
MoveTo(10,10);
LineTo(90,10);
LineTo(90,40);
LineTo(10,40);
LineTo(10,10);
}
```

Here is the program to draw the white square:

```
#include <quickdraw.h>

void main(void)

{
SetPenMode(0);
SetSolidPenPat(3);
SetPenSize(3,1);
MoveTo(10,10);
LineTo(90,10);
LineTo(90,40);
LineTo(10,40);
LineTo(10,10);
}
```

Solution to problem 1.5.

```
#include <quickdraw.h>

void main(void)

{
SetPenMode(0);
SetSolidPenPat(0);
SetPenSize(3,1);
MoveTo(110,50);
LineTo(190,50);
LineTo(150,24);
LineTo(110,50);
}
```

Solution to problem 1.6.

```
#include <quickdraw.h>

void main(void)

{
SetPenMode(0);
SetSolidPenPat(2);
SetPenSize(6,2);
MoveTo(110,50);
LineTo(190,50);
LineTo(150,24);
LineTo(110,50);
MoveTo(110,33);
LineTo(190,33);
LineTo(150,59);
LineTo(110,33);
}
```

Solution to problem 1.7.

```
#include <quickdraw.h>

void main(void)

{
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);
}
```

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 shown in listing 2.1.

One of the first things you will see in our program is a comment. Comments start with a /* character sequence and end with a */ character sequence. It is very, very important to put the characters together. You can't put a space or any other character between the slash and asterisk. You can type anything you want except the */ character sequence inside of a comment, though. The compiler ignores comments 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.

Listing 2.1

```
/* This program prints a table of numbers and squares of the numbers */

#include <stdio.h>

int i,s;                        /* i is a number, s is its square */

void main (void)

{
i = 1;
s = i*i;
printf("%10d%10d\n", i, s);
i = i+1;
s = i*i;
printf("%10d%10d\n", i, s);
i = i+1;
s = i*i;
printf("%10d%10d\n", i, s);
i = i+1;
s = i*i;
printf("%10d%10d\n", i, s);
i = i+1;
s = i*i;
printf("%10d%10d\n", i, s);
}
```

Computers can work with a vast array of number formats, each of which has a special purpose. The two most common number formats are integer and floating-point. Integers are whole numbers, like 4, -100, or 1989. Floating-point numbers include the integers, plus all of 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 C, 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.

C is very flexible about where it lets you define things. In this program, we defined the two integers before the function main. Later, you will learn how to define variables inside of a function. Variables defined outside of a function, like the ones in our sample program, can be used from any function that comes after the variable. These are called global variables. Variables defined inside of a function can only be used from within the function; they are called local variables. You can mix variables and functions in any order, as long as you make sure you put the variable definitions before the first function that uses the variable. In all of our programs, we will adopt the common style of putting all variables before the start of the first function. That makes the variable definitions easier to find and change later on.

In C, when you define something, you start by telling the compiler what type of thing you are defining. We are defining two integers. In C, we use the abbreviation int to tell the compiler we want to define an integer variable. The variables are listed next. If there are more than one, like there are in our program, we can use commas to separate them. At the end of the definition, we type a semicolon to tell the compiler where the definition ends.

These variables are put to use in the function main. 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 = character is called the assignment operator. 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.

```
printf("%10d%10d\n", i, s);
```

The call to printf deserves a little more attention, since there are several new concepts here. We have already used the printf function to write characters to the shell window, but in this case we are writing two numbers. The printf function can actually print a wide variety of different data types. To write anything besides a string, though, there are a couple of new ideas to master. First, we still need to pass a string. The first parameter to printf is always a string, even if we only

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 printf call. 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 printf 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 printf 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.

want to print numbers. This string controls the format of the information that will be written to the shell window; it forms a sort of model. The % character starts something called a conversion specification. The conversion specification tells printf what to expect in the rest of the parameter list, as well as how to format the various variable values. In our example, we used %10d. The 10 tells printf to put the value in a ten character wide field, forcing the value to the right side of the field. When a 1 is printed, for example, it will be preceded by 9 spaces. The d tells printf that the value to print is an integer. Any characters that are not part of a conversion specification are printed as is, which is why we were able to use printf to write a simple string.

The parameters appear right after the format string. They are matched with the conversion specification from left to right; all three parameters, the format string and the two variables, are separated from each other with commas.

It is extremely important that you type the conversion specification exactly as you see it, with no spaces or other characters imbedded among the characters. You also have to be very careful with conversion specifications. As you will see, there are a lot of different kinds of variables in C, and many different kinds of conversion specifications, as well. You have to make sure that the number and kind of the conversion specifications match the number and kind of parameters that come after the format string exactly. If you don't, it could cause all sorts of trouble, including a crash. Don't worry – it will eventually happen to you, just like it does to every other C programmer. When it happens, reboot and try again.

I said there are a lot of conversion specification and data types – you may be wondering what they are. Eventually, we will cover most of them in the course, but peeking ahead is not only allowed, it is encouraged. Your first look at a technical description of conversion specifications may be scary, so be prepared. Your compiler reference manual has a complete list of them, though. Look up the printf function in the index, and flip to the proper place in the book. Don't panic. It's easier than the manual makes it seem.

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

In the example, we used a 10 for a width field. You can leave it out, using "%d%d" as the format string, but if you do, the two numbers will be crammed together. To separate them, just put a space between the two conversion specification. Remember, you can't put spaces inside of the conversion specification, but you can put anything you like around the conversion specification. As an example, try to figure out what this printf statement will do. After you think you know (or after you give up), try this format string in your program to see if you were right!

```
printf("%d squared is %d.\n", i, s);
```

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 below and run 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 the graphics window before running the program.

```
/* Draw a fan shape in the     */
/* graphics window             */

#include <quickdraw.h>

int i;        /* loop variable */

void main (void)

{
/* set up for graphics */
SetPenMode(0);
SetSolidPenPat(2);
SetPenSize(2,1);

/* draw the fan */
for (i = 1; i <= 25; ++i) {
   MoveTo(160, 70);
   LineTo(i*12-10, 10);
   }
}
```

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`. Right after this are three expressions, separated from each other by semicolons, and enclosed in parentheses. To understand what all of this means, let's step back and look at the for statement in a symbolic way, putting names in for the various expressions.

```
for (start; stop; loop)
   statement;
```

The for loop actually does quite a number of rather complicated things:

1. The first thing it does is to execute the start expression; that's the first one in the parentheses. You have seen an expression like this before; in our example program, the start expression stores a 1 in the variable i.

2. Next, the for statement executes the stop expression; that's the one in the middle. The stop condition looks a little odd. What it does is to compare i to 25. If i is less than or equal to 25, the for loop will keep going. If i is greater than 25, the for loop will stop, never having gone through the loop at all.

3. The third step is to execute the statement that comes after the closing parenthesis. The statement could be assigning a value to a variable, calling the printf function to write a value, or something a little more complicated. The for loop only executes one statement, though. We'll talk about this point more in a moment.

4. The next step is to execute the loop statement. In the example, you saw something else that is new; ++i. The ++ is actually an operator in C. An operator in a compiler isn't the person who answers the phone; it's a symbol that means "do something!" You have already used the + and * operators, which were probably familiar from math class. The ++ operator means to add one to the variable. The expression ++i means exactly the same thing as i = i+1, but it is a little easier to type. This sort of thing pops up time and again in C, so get used to it!

5. The last step is to go back to step 2, checking to see if we are finished with the loop.

You have to be careful with the for loop, since C is a trusting language. It gives you plenty of rope; it's up to you to hang yourself or build a rope bridge. For example, you can put an assignment in for the loop condition. Let's say you type

```
for (i = 1; i <= 10; i = 4)
   printf("Hello, world.\n");
```

This loop will never stop, since i is never greater than 10.

I mentioned earlier that the for loop 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 symbols, { and }, around the two statements. This tells the compiler to treat the group of lines between the { and } 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 { and } characters, like this:

```
for (i = 1; i <= 25; ++i)
   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 { and } characters. 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 { and } 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 C, the authors will talk about compound statements instead of continually saying "those statements grouped together with the characters { and }." 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.

There is one very common mistake to watch out for with the for loop. You are getting used to the fact that C statements end with a semicolon. So does the statement after the for loop. This is perfectly valid in C:

```
for (i = 1; i <= 10; ++i) ;
   printf("Hello, world.\n");
```

It loops 10 times, doing nothing, then calls printf once. The reason is that the ; after the for statement is a legal statement that does nothing. Watch out for those extra semicolons! Here's how to call the printf function 10 times:

```
for (i = 1; i <= 10; ++i)
   printf("Hello, world.\n");
```

The difference is small, but very, very important.

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:

```
/* draw the fan */
for (i = 1; i <= 25; ++i) {
   MoveTo(160, 70);
   LineTo(i*12-10, 10);
   }
```

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, these lines do exactly the same thing:

```
for (i = 1; i <= 25; ++i) {
MoveTo(160, 70); LineTo(i*12-10,
10);}
```

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

```
/* draw the fan */
for (i = 1; i <= 25; ++i) {
MoveTo(160, 70);
LineTo(i*12-10, 10);
}
```

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 { and } characters to show you that the for loop only executes one statement at a time, and how the { and } characters convince the compiler to treat two lines as a single statement, I wrote

```
for (i = 1; i <= 25; ++i)
   MoveTo(160, 70);
   LineTo(i*12-10, 10);
```

From the indenting, you would expect the second line 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.

There is more than one right way to do almost anything in programming, and indenting is one of those places where this is particularly true. I always use three spaces when I indent. I always put the { character 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; i <= 25; ++i)
   {
   MoveTo(160, 70);
   LineTo(i*12-10, 10);
   }

for (i = 1; i <= 25; ++i)
   {
      MoveTo(160, 70);
      LineTo(i*12-10, 10);
   }
```

```
for (i = 1; i <= 25; ++i)
{
        MoveTo(160, 70);
        LineTo(i*12-10, 10);
}
```

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 */

#include <stdio.h>

void main(void)

{
printf("%d\n", 1+2*3);
}
```

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, C uses the same rules as algebra teachers. 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.

Operator precedence is complicated in C because there are so many operators. In fact, there are 16 levels of precedence in C! Most C programmers don't even try to remember what all of the precedence rules are. Even if you could, your programs would confuse other people when they tried to read them. In C, remember that multiply and divide (* and /) have a higher precedence than add and subtract (+ and -). The ++ operator has an even higher precedence: it is done even before a multiply or divide. Beyond that, use parenthesis. For example, (1+2)*3 forces the compiler to do the addition first, giving 9. Parenthesis are a good way to make sure that you, the compiler, and anyone else who reads your program are all sure what the expression really means.

Of course, you may want to peek ahead to see what an awesome list of operators C really has, and see that complete, 16 level list of operator precedence. Go for it. You'll find the list in your language reference manual, as well as any decent C reference manual. We'll also keep you posted on the precedence of new operators as they are introduced in the course.

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 isn't Westerfiel, it's 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, and integers are no exception. Every C compiler imposes some upper limit on integers – some largest number that can be stored in an integer variable. On most microcomputer based Cs, this value is... but wait, there is a better way to find out.

```
/* Find the largest integer */

#include <stdio.h>
#include <limits.h>

void main(void)

{
printf("%d\n", INT_MAX);
}
```

INT_MAX is a constant you can find in the library of most C compilers; it is in limits.h. 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. It's name is MIN_INT. It is usually either -32767 or -32768. In ORCA/C, the value is -32767.

C 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.

Integers Come in Several Sizes

C actually has several different kinds of integers. The one we have been using so far is called int; that one is supposed to be the "natural" size for an integer on the machine you are using. There are two other types of integers, though, called long and short. You can define integers using int, long, short, or even the combinations short int or long int. In ORCA/C, short means exactly the same thing as int, while long integers give you a larger range of values. With ORCA/C, long integers can range from -2147483647 to 2147483647. The limits.h header file defines constants for the size of long and short integers, too. You can use these constants to find out how big integers are on any other C compiler you might someday use.

constant	type	value in ORCA/C
SHRT_MIN	short	-32767
SHRT_MAX	short	32767
INT_MIN	int	-32767
INT_MAX	int	32767
LONG_MIN	long	-2147483647
LONG_MAX	long	2147483647

When C was originally defined, there weren't any limits on the range of integers each of the types could hold. In fact, it was perfectly legal to use the same range for all three kinds of integer, although that was rarely done. Even today, with the ANSI C standard, it is still legal for a C compiler to make all three kinds of integers hold the same range of numbers, but the values shown in the table are now the minimum sizes. In other words, in any ANSI C compiler, you can count on the

fact that short and int variables can hold values from -32767 to 32767. The allowed range can be bigger, but not smaller.

Integers Can be Signed or Unsigned

Just to keep things interesting, C not only has three different sizes of integers (actually four; you'll learn about char variables later), it also has two different kinds of each of these sizes of integer. There are some cases where you need an integer, but you don't need one that can represent a negative value. In fact, we haven't used a negative value in any of our programs, yet. C defines a kind of integer called unsigned as an integer with a low value of 0. It turns out that there are two distinct advantages to unsigned integers on many computers, and this is especially true on the Apple IIGS. First, unsigned integers can hold larger values in the same amount of space. For example, unsigned int values can be as large as 65535. The other advantage is that many operations are faster with unsigned integers than they are with signed integers. Addition and subtraction take the same amount of time, at least on the Apple IIGS. Multiplication and division, though, are a little faster. The difference isn't usually enough to be important, but in some very time-critical applications, a slight improvement in speed could help a lot. The most important difference, though is for compares. Comparing two signed values takes a long time on an Apple IIGS, while comparing two unsigned integers is very easy, and very fast.

To define an unsigned integer, you put the word unsigned right before the normal definition, like this:

```
unsigned short s;
unsigned int i,j,k;
unsigned long l;
```

The most common size of integer in C programs is int, so C has a short cut for unsigned int variables. You can leave off the int, like this:

```
unsigned i,j,k;
```

This definition means the same thing as the definition of i, j and k in the original example, but it takes a few less keystrokes to type it in.

As with the signed integers, limits.h defines the size of unsigned integers, too. For unsigned integers, though, only the largest value is given, since the smallest value the variable can hold is always 0.

constant	type	value in ORCA/C
USHRT_MAX	short	65535u
UINT_MAX	int	65535u
ULONG_MAX	long	4294967295ul

The u after the number tells the compiler the number is unsigned. It's good practice to put the u on the end of unsigned numbers, although it won't make any difference if the number is smaller than the largest legal signed integer of the same type. You can also put an l after an integer constant to tell the compiler the value is long. If the integer is too big to fit in an integer variable, the compiler will promote it to long automatically, but there are a few places where leaving off something as simple as an l to indicate a long integer can actually cause a program to crash. Again, it is a good habit to use l when appropriate.

One of the reasons for using C instead of a safer language, like Pascal, is that there are several areas in the C language where a complicated idea like three sizes of integers and two kinds of integers can help you write smaller, faster programs. Standard Pascal, for example, only has one size and one kind of integer, although most real compilers have two sizes of integers. Getting used to which kind of integer to use can be a little intimidating, though. The basic rule of thumb is to use unsigned whenever you can. Use int variables if the values you will be using will fit in an int variable; int variables require half the room used by the same number of long variables, and calculations with int variables are two to four times faster than calculations with long variables. Use long or unsigned long when you need to handle values too big for int variables.

More Conversion Specifications

The printf subroutine is pretty picky about matching up format specifiers with the appropriate type of parameter. If you use the wrong number of parameters, or you tell printf that you are printing an integer by using the %d conversion specification, and then pass a long variable, the program could even crash. (In version 1.0 of ORCA/C, the program would crash or hang. In version 1.1, the program will not, unless you ask the compiler to optimize the program. You'll learn more about optimizations later.) Even if the values are the same size, you may not get the result you want.

For example, in this program, we obviously want to print 40000, which is a perfectly valid unsigned int value.

```
/* try printing a large */
/* unsigned int         */

#include <stdio.h>

void main (void)

{
printf("%d\n", 40000u);
}
```

When you run this program, it will print -25536; that's a far cry from 40000! The reason, of course, is that the conversion specification %d tells printf to expect an int value, but the program passes an unsigned int. By changing the letter to u, like this:

```
/* try printing a large */
/* unsigned int         */

#include <stdio.h>

void main (void)

{
printf("%u\n", 40000u);
}
```

you can tell printf to expect an unsigned int.

Long values are handled a little differently. To tell printf to expect a long value, rather than an int value, you put an l right before the d. An unsigned long is indicated with lu, rather than u.

```
/* sample of printf with long */
/* and unsigned long values   */

#include <stdio.h>

void main (void)

{
long l;
unsigned long ul;

l = 5000000l;
ul = 100ul;

printf("%lu\n", 123456lu);
printf("%ld\n", -100000l);
printf("%ld\n", l);
printf("%lu\n", ul);
}
```

Just to keep things interesting, short values use the same conversion specification as int values.

Here's a table that summarizes the conversion specifications for printf up to this point. Only the letter is shown in the table, but in all cases, the conversion specification starts with a % character. You can also use a number between the % character and the letter for the conversion specification to force the numbers into columns with any of the conversion specification.

variable type	conversion specification
short	d
unsigned short	u
int	d
unsigned int	u
long	ld
unsigned long	lu

Floating-Point 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 in listing 2.2 finds out how many months you will be paying.

Be sure you have typed the format string in the call to printf *exactly* as it is shown, then go ahead and run the program. 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 new preprocessor directive called define. The #define statement defines something called a macro. The word that comes right after #define is the name of the macro; the characters that come after the macro name are called the macro body. The compiler remembers this macro definition, and any time it finds the name of the macro in the program, it uses the characters from the macro body instead of the name of the macro. Macros are very powerful, but for now we will only use them to define constants. Constants have several uses. Like comments, they make a program easier to read, understand, and change. 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 shown in listing 2.3.

This program does exactly the same thing as the original, but the first is easier to change and understand. In addition, 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.

C is case sensitive, so you must type the name of the macro in uppercase if it is defined that way. As long as the case of the letters is the same, the compiler really doesn't care if you use uppercase letters or

Listing 2.2

```
/* Why I don't own a Porche */

#include <stdio.h>

#define COST 24000.0              /* initial cost of the car */
#define APR 10.0                  /* annual percentage rate */
#define PAYMENT 250.0             /* monthly payment */

unsigned month;                   /* number of months */
float principal;                  /* amount left to pay */

void main (void)

{
month = 0;                        /* no payments made, yet */
principal = COST;                 /* we start owing this much */
while (principal > 0.0) {       /* keep going until we're out of debt */
   ++month;                       /* it's a new month */
   principal += principal*        /* add in this month's interest */
      APR/100.0/12.0;
   principal -= PAYMENT;          /* make the payment */
                                  /* write the status */
   printf("month = %3u   principal = %.2f\n", month, principal);
   }
}
```

Listing 2.3

```
/* Why I don't own a Porche */

#include <stdio.h>

unsigned month;                  /* number of months */
float principal;                 /* amount left to pay */

void main (void)

{
month = 0;                       /* no payments made, yet */
principal = 24000.0;             /* we start owing this much */
while (principal > 0.0) {       /* keep going until we're out of debt */
   ++month;                      /* it's a new month */
   principal += principal*       /* add in this month's interest */
      10.0/100.0/12.0;
   principal -= 250.0;           /* make the payment */
                                 /* write the status */
   printf("month = %3u   principal = %.2f\n", month, principal);
   }
}
```

lowercase letters for your macro names, but it is traditional in C to use uppercase characters for macro names. We'll follow that tradition in this course.

C is full of short-cuts, and this program introduces a new one. It turns out that there are a lot of cases in programming where you want to add something to a variable, or multiply a variable by some value. For most of the operators you will learn called binary operators – that is, for all of the ones like +, -, * and / that have a value on each side – you can shorten an expression that does something to a variable and stores the result back in the variable. In our program, we needed to subtract PAYMENT from the principal. That's easy enough to do like this:

```
principal = principal - PAYMENT;
```

but you can also do exactly the same thing like this:

```
principal -= PAYMENT;
```

The program does the same thing either way, but the second statement takes less typing, and can reduce the chance of typing something incorrectly. In some compilers (although not in ORCA/C), the second statement might execute a bit faster, too.

The difference between floating-point numbers and integers is that floating-point numbers can have values other than whole numbers, but they don't have to. In our constant 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 floating-point 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 floating-point number, but 250 is an integer. In C, the decimal point can come at the beginning or end of a number, as long as there is a number on one side or the other. For example, all of these are legal constants in C:

```
3.14159     10.0        0.0
0.001       1.          .1
```

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)
```

the condition is that the principal must be greater than zero. 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, one of which you saw earlier in the for loop. 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 fact, you can define a for loop in terms of a while loop. Back when you first saw the for loop, I wrote it this way to make it easier to talk about what the for loop was doing:

```
for (start; stop; loop)
   statement;
```

We can do the same thing for the while loop.

```
start;
while (stop) {
   statement;
   loop;
   }
```

This for loop and while loop do *exactly* the same thing. In general, we use the for loop when we know in advance how many times we will execute the loop, or when the number of times can be expressed as a simple mathematical expression. The while loop is usually used when we want to loop while some condition is true, as we did here by looping while we still needed to pay on the car.

As with strings and integers, we can write a floating-point number by calling printf. Like the conversion specification for an integer, the conversion specification for a floating-point number starts with the % character. Instead of a d, though, we use f. The basic conversion specification for a floating-point number, then, is %f.

With integers, you put a number right after the % character to tell printf to put the numbers in a column. You can do the same thing with floating-point numbers, although we didn't in this case. With a floating-point number there is another value, though. Our program is supposed to print a dollar amount, so we want two digits after the decimal point. The %.2f conversion specification tells printf exactly that: the .2 tells printf to print exactly two digits after the decimal point, rounding the number if needed. To get a ten character wide column of dollar and cents amounts, we could use the conversion specification %10.2f. As with integers, if you leave the field width (the first number) off, the number is printed without leading spaces. If you don't tell the compiler how many digits to put after the decimal point, it will use a value of six.

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, since 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 lesson. 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 shown in listing 2.4, 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 begins to grow, rather than shrinking. The result is that the program will never stop. This is one form of the infamous infinite loop. Once you realize

Listing 2.4

```
/* I owe, I owe, ... */

#include <stdio.h>

#define COST 24000.0              /* initial cost of the car */
#define APR 15.0                  /* annual percentage rate */
#define PAYMENT 250.0             /* monthly payment */

unsigned month;                   /* number of months */
float principal;                  /* amount left to pay */

void main (void)

{
month = 0;                        /* no payments made, yet */
principal = COST;                 /* we start owing this much */
while (principal > 0.0) {       /* keep going until we're out of debt */
   ++month;                       /* it's a new month */
   principal += principal*        /* add in this month's interest */
      APR/100.0/12.0;
   principal -= PAYMENT;          /* make the payment */
                                  /* write the status */
   printf("month = %3u   principal = %.2f\n", month, principal);
   }
}
```

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. Floating-point numbers have limits, too, but the limits are of a slightly different nature. This is because floating-point 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 floating-point 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.

Floating-point numbers can get quite large and quite small, but there is a limit to the size. In ORCA/C, floating-point 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/C floating-point numbers have seven digits of accuracy. In other Cs, you would have to check the manual to find out how accurate the numbers are, but these are fairly common values.

If you need larger exponents or more digits of accuracy in your floating-point numbers, you can use variables with a type of double, instead of float. They work the same way – double numbers even use the same conversion specification as float numbers – but double variables take up twice the space as float variables, giving you exponents from 1e-308 to 1e308. Double variables can display seventeen digits accurately. Again, these are common values for C compilers, but there is nothing to prevent a particular compiler from making float and double variables the same size.

With integers, using a longer integer meant accepting a speed penalty. The same is true on some computers when dealing with floating-point numbers, but due to a peculiarity in the way floating-point calculations are done on the Apple IIGS (and most other microcomputers, as it turns out), calculations with float variables take about the same amount of time as calculations with double variables. The only thing float saves you is space.

The example in listing 2.5 shows how to use floating-point numbers to represent very large numbers. By using e instead of f in the conversion specification, we are telling the compiler to write the value in exponent format. This is also the format used by the debugger to display floating-point numbers.

If you look closely at this program, you will see something that looks a little odd. We want to print "At 3% growth...," but as you know, the % character marks the start of a conversion specification. The printf function has to use a special convention so that you can use the % character as a conversion specification, and still be able to print the character when you want to. Technically, % is another conversion specification that prints a single % character without requiring a parameter after the format string; I generally find it easier to just remember that % is a special character in a format string, so I need to write two of the % character whenever I want tp print one of them.

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?

Don't Panic!

By now, your head is probably spinning. With six types of integers and two types of floating-point numbers, plus a variety of conversion specification, some of which work with some values, and some of which don't, things may be getting a little confusing. Don't worry. It happens to everyone.

The important thing at this point isn't to memorize all of the number types, conversion specifications, and so forth, the important point is to realize that they exist, and why. You should know that unsigned integers exist because they can store larger numbers than signed

Listing 2.5

```
/*  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?                                             */

#include <stdio.h>

float people;                    /* number of people */
unsigned year;                   /* current year */

void main (void)

{
people = 5e9;                    /* 5e9 is 5000000000, or 5 billion */
for (year = 1989; year < 2090; ++year)
   people *= 1.03;
printf("At 3%% growth, there will be %e people in 2089.\n", people);
}
```

integers of the same size, and that calculations with unsigned values are faster than calculations with signed values. You should know that integers let you represent whole numbers, and that floating-point values are used when values between the whole numbers are needed, or when the values are very large or very small. You don't have to remember that lu is the conversion specification for unsigned long values – you can look that up in this lesson or in your compiler reference manual if you forget.

As the course progresses, you will see each of these number formats used again and again. Gradually, you will get used to them. By the end of the course, this lesson will seem pretty easy, instead of being the blur of facts it probably is now. As long as you keep the concepts in mind, you'll do fine.

Lesson Two
Solutions to Problems

Solution to problem 2.1.

```
/* Write out five Fibonacci numbers */

#include <stdio.h>

int last;                               /* last number */
int current;                            /* current number */
int next;                               /* new number */

void main(void)

{
last = 0;
current = 1;

next = last+current;
printf("%d\n", next);
last = current;
current = next;

next = last+current;
printf("%d\n", next);
last = current;
current = next;

next = last+current;
printf("%d\n", next);
last = current;
current = next;

next = last+current;
printf("%d\n", next);
last = current;
current = next;

next = last+current;
printf("%d\n", next);
last = current;
current = next;
}
```

Solution to problem 2.2.

```
/* write a table of squares */

#include <stdio.h>

int i;                          /* loop variable */
int s;                          /* square of i */

void main(void)

{
for (i = 1; i <= 5; ++i) {
   s = i*i;
   printf("%10d%10d\n", i, s);
   }
}
```

Solution to problem 2.3.

```
/* draw five concentric rectangles */

#include <quickdraw.h>

int i;                          /* loop variable */
int top, bottom, left, right;   /* sides of the rectangle */

void main(void)

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

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

```
for (i = 1; i <= 5; ++i) {
   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;
   }
}
```

Solution to problem 2.4.

```
/* Why I don't own a Porche */

#include <stdio.h>

#define COST 24000.0              /* initial cost of the car */
#define APR 10.0                  /* annual percentage rate */
#define PAYMENT 608.71            /* monthly payment */

unsigned month;                   /* number of months */
float principal;                  /* amount left to pay */

void main (void)

{
month = 0;                        /* no payments made, yet */
principal = COST;                 /* we start owing this much */
while (principal > 0.0) {         /* keep going until we're out of debt */
   ++month;                       /* it's a new month */
   principal += principal*        /* add in this month's interest */
      APR/100.0/12.0;
   principal -= PAYMENT;          /* make the payment */
                                  /* write the status */
   printf("month = %3u   principal = %.2f\n", month, principal);
   }
}
```

Solution to problem 2.5.

```
/* See how long it will take to reach a population of 50,000 */

#include <stdio.h>

#define CURRENTPOPULATION 30000.0     /* initial population of the city */
#define GROWTH 10.0                     /* population growth rate */

unsigned year;                          /* year */
float population;                       /* population during year */

void main(void)

{
year = 1990;                            /* current year */
population = CURRENTPOPULATION;         /* initialize the population */
while (population < 50000.0) {        /* keep going until we hit 50,000 */
   ++year;                              /* new year */
   population +=                        /* add in this year's growth */
      population*GROWTH/100.0;
                                        /* write the status */
   printf("The population in %u will be %.0f.\n", year, population);
   }
}
```

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 */

#include <stdio.h>

#define CURRENTCOST 1.00                /* current cost of the item */
#define INFLATION 4.0                   /* inflation rate */

unsigned year;                          /* number of years */
float cost;                             /* cost of the item */

void main(void)

{
cost = CURRENTCOST;
for (year = 1990; year <= 1999; ++year)
   cost += cost*INFLATION/100.0;
printf("Cost of a $%.2f item after 10 years is $%.2f.\n",
   CURRENTCOST, cost);
```

```
}
```

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

```
/* house cost at 4% inflation */

#include <stdio.h>

#define CURRENTCOST 100000.0            /* current cost of the item */
#define INFLATION 4.0                   /* inflation rate */

unsigned year;                          /* number of years */
float cost;                             /* cost of the item */

void main(void)

{
cost = CURRENTCOST;
for (year = 1990; year <= 1999; ++year)
   cost += cost*INFLATION/100.0;
printf("Cost of a $%.2f item after 10 years is $%.2f.\n",
   CURRENTCOST, cost);
}
```

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

```
/* house cost at 12% inflation */

#include <stdio.h>

#define CURRENTCOST 100000.0            /* current cost of the item */
#define INFLATION 12.0                  /* inflation rate */

unsigned year;                          /* number of years */
float cost;                             /* cost of the item */

void main(void)

{
cost = CURRENTCOST;
for (year = 1990; year <= 1999; ++year)
   cost += cost*INFLATION/100.0;
printf("Cost of a $%.2f item after 10 years is $%.2f.\n",
   CURRENTCOST, cost);
}
```

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?                         */

#include <stdio.h>

float germs;                            /* number of germs */
unsigned time;                          /* time loop counter */

void main(void)

{
germs = 1;
for (time = 1; time <= (24*3); ++time)
   germs *= 2.0;
printf("After 24 hours, there will be %e germs!\n", germs);
}
```

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 printf function to write a string to the shell window. You have already learned to write integers and real numbers using printf. The scanf library function is the counterpart to printf: it is used in much the same way to read numbers and strings. Like printf, scanf is defined in the stdio.h header file, so you don't have to include any new libraries to read values from the keyboard.

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.           */

#include <stdio.h>

int i;

void main (void)

{
scanf(" %d", &i);
printf("%d\n", i);
}
```

When the program gets to the scanf function call, it stops. In the shell window, you can see an inverse box, which 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 using printf.

Like printf, scanf is controlled with a format string, and the format string is still the first parameter. This time, though, the format string is used as a pattern for the characters that scanf reads. There are basically three things that you can put in scanf's format string: a conversion specifier, like the %d you see in the sample program; whitespace (more on that later); and other characters.

The conversion specifier is used when you want to read something, like a number, from the input stream. (In C, we talk about steams a lot. Later, you will learn more about streams, but for now, keep in mind that the phrase "input stream" just means the characters you type from the keyboard.) The conversion specifiers used by the scanf function work pretty much like the conversion specifiers used by printf, and generally mean the same thing. As you learn more about C, you will start to find out what all of the exceptions are, but all of the conversion specifiers you have learned so far work the same way in scanf and printf. The %d conversion specifier, for example, reads an integer value from the input stream, converts it into the internal format used to store numbers on a computer, and stuffs the result into the integer variable i. As with printf, each conversions specification needs exactly one parameter. With printf, though, the idea was to print a value, so you could put a wide variety of things in the parameter list for the function, including constant numbers like 1 or 500, variables like i or interest, and even expressions like interest/2. The scanf function, though, is reading a value and storing it someplace. As a result, the parameter must be a variable name. In addition, scanf needs to know where the variable is, not it's value. In C, when you want to tell a function where a variable is, you pass the address of the variable by putting an & character right before the name of the variable. What you are doing is passing a pointer to the variable. Over the next few lessons we will spend a great deal of time discussing pointers, what they are, and how they are used, but you need to know a lot more about programming in general, and C in particular, before they will make sense. For now, just remember to put a & character before all of the variables you pass to the scanf function.

When you use a pencil and paper to write text, columns of numbers, and so forth, you generally use blank space to help organize the information. You also use blank space to organize information when you type things from a keyboard, but as you may have noticed, there is more than one way to get blank space with a computer. For example, you can use the space bar to insert some spaces; or the TAB key to insert several spaces or a tab character, depending on the editor you

are using. When you type responses to a program, you generally expect to be able to use whichever method you feel like using at the moment, and you probably don't want the program to be sensitive to exactly how many spaces you type. C solves this problem by grouping several characters together, calling all of them whitespace characters. The idea is that the whitespace characters are the ones that look like spaces on the computer screen, although different printers, the operating system, and the toolbox may all treat these characters a little differently. The whitespace characters in C are the space, the end-of-line character, vertical and horizontal tabs, and form feeds. So far, you have only seen two of these characters in your C programs: the space bar, and the end-of-line character, which you put into a string using the \n character. Tabs are available, too, but you generally won't use them in desktop programs, since the Apple IIGS toolbox ignores tab characters.

The scanf function treats these whitespace characters in a special way to help you create programs that work the way the user wants to use the program without much effort on your part. In our sample program, the format string started with a space. Here's that string again:

```
" %d"
```

When scanf sees a whitespace character in the format string, it skips over all whitespace characters in the input stream, stopping when it gets to the first non-whitespace character. It doesn't matter how many whitespace characters are in the input, what kind of whitespace character they are, or even if they are mixed; all of them get skipped.

Finally, you can put other characters in the format string. If you do, the scanf function expects to find exactly the same character in the input stream.

If scanf hits something it doesn't expect, it stops. That's bad news, of course, since with some compilers and optimizations, the resulting C program might crash or hang if it gets input that it can't handle. Later in the course, you will learn how to write bullet-proof C programs that won't crash or hang with any compiler; for now, ORCA/C 1.1 will protect you from crashes when your input is bad as long as you don't optimize the code. We haven't covered how to use the optimizer yet – this is just a warning to those of you who are adventurous enough to be looking ahead in the ORCA/C manual and trying new things before they are covered in the course. (That's a practice I highly recommend, by the way.)

The number you type from the keyboard follows the same rules as numbers you type in a C program; namely, they must consist entirely of digits, and they must not have imbedded whitespace. You can also start the number with a + or - sign. scanf stops when it gets to the first character that is not a part of a legal number, returning whatever has been typed so far.

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, scanf will truncate the number in a way that doesn't make much sense unless you know a lot about how binary numbers are stored in the computer. The number printed by printf will definitely not be the same number you type, though!

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 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 Do-While Loop

One of the new things in our program is a new statement, called the do-while loop, or sometimes just the do loop. This is the third looping statement you have learned in C. The first two, of course, are the for

Listing 3.1

```
/* Guess a number */

#include <stdio.h>
#include <stdlib.h>

int i;                                  /* input value */
int value;                              /* the number to guess */

void main(void)

{
/* Introduce the game */
printf("In this game, you will try to\n");
printf("guess a number.  I need a hint to\n");
printf("help me pick numbers, though.\n\n");

/* get a seed for the random number generator */
printf("Please type a number between 1\nand 30000: ");
scanf(" %d", &i);
srand(i);

/* pick a number from 1 to 100 */
value = rand() % 100 + 1;

/* let the player guess the number */
do {
   printf("Your guess: ");           /* get the guess */
   scanf(" %d", &i);
   if (i > value)                    /* check for too high */
      printf("%d is to high.\n", i);
   if (i < value)                    /* check for too low */
      printf("%d is to low.\n", i);
   }
while (i != value);               /* if the guess was wrong then loop */
printf("%d is correct!\n", i);
}
```

loop and the while loop. The do loop is also the last looping statement in C! You're getting there...

Like the while loop, the do loop loops while 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 do 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 do loops from some of our programs, and compare the two.

In the last lesson, we wrote a program that showed how many payments were needed to buy a car. The source code is shown in listng 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

Listing 3.2

```
   while (principal > 0.0) {          /* keep going until we're out of debt
*/
      ++month;                        /* it's a new month */
      principal += principal*         /* add in this month's interest */
         APR/100.0/12.0;
      principal -= PAYMENT;           /* make the payment */
                                      /* write the status */
      printf("month = %3u   principal = %.2f\n", month, principal);
      }
```

Listing 3.3

```
   do {
      printf("Your guess: ");           /* get the guess */
      scanf(" %d", &i);
      if (i > value)                    /* check for too high */
         printf("%d is to high.\n", i);
      if (i < value)                    /* check for too low */
         printf("%d is to low.\n", i);
      }
   while (i != value);                  /* if the guess was wrong then loop */
```

Listing 3.4

```
   while (i != value) {
      printf("Your guess: ");           /* get the guess */
      scanf(" %d", &i);
      if (i > value)                    /* check for too high */
         printf("%d is to high.\n", i);
      if (i < value)                    /* check for too low */
         printf("%d is to low.\n", i);
      }
```

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

The do 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 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.

Listing 3.5

```
   i = value-1;                          /* make sure we get into the loop
*/
   while (i != value) {
      printf("Your guess: ");           /* get the guess */
      scanf(" %d", &i);
      if (i > value)                    /* check for too high */
         printf("%d is to high.\n", i);
      if (i < value)                    /* check for too low */
         printf("%d is to low.\n", i);
      }
```

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 switch that 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.

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 do 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 do 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 inside the loop. The test doesn't make sense until the number has been read at least one time, so we use a do loop.

How C Divides

Everyone knows that when you divide 3 by 2, the answer is 1.5. Unfortunately, the computer doesn't 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 shown in listing 3.6 will help us explore this, and many other curiosities about the / (divide) and % (mod) 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
```

```
   a       b       a/b       a%b
 -10       4        -2         2
  -9       4        -2         3
  -8       4        -2         0
  -7       4        -1         1
  -6       4        -1         2
  -5       4        -1         3
  -4       4        -1         0
  -3       4         0         1
  -2       4         0         2
  -1       4         0         3
   0       4         0         0
   1       4         0         1
   2       4         0         2
   3       4         0         3
   4       4         1         0
   5       4         1         1
   6       4         1         2
   7       4         1         3
   8       4         2         0
   9       4         2         1
  10       4         2         2
```

When you use the / operator to divide two integers, it always gives an integer result. Of course, that means that the answer isn't 1.25 when you divide 5 by 4, since 1.25 isn't an integer. Instead, C 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, the answer is -1.

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

Looking at the table from our sample program, you can see that the % operator behaves a little strangely when one number is negative. For example, (-1) % 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 % 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. If you

Listing 3.6

```
/* Exploring / and % */

#include <stdio.h>

int n;                              /* the "top" number in the division */
int d;                           /* the "bottom" number in the division */
int first,last;                     /* the first/last number to divide */

void main(void)

{
printf("first       :");           /* get the first/last numbers */
scanf(" %d", &first);
printf("last        :");
scanf(" %d", &last);
printf("denominator :");           /* get the number to divide by */
scanf(" %d", &d);
                                   /* write the table */
printf("\n     a      b       a/b       a%%b\n");
for (n = first; n <= last; ++n)
   printf("%6d %6d %9d %9d\n", n, d, n/d, n%d);
}
```

try to divide by 0 in C, the results are undefined, which means the compiler can return anything it wants to. Modular arithmetic also imposes an additional restriction on the % 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 % 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 % operator always returned a value from 0 to 3. In our number guessing game, we used the % operator to restrict the value returned by the random number generator to a small range of numbers.

```
value = ran() % 100 + 1;
```

The rand 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 rand, but it might get boring trying to track down a number in that range. The result of

```
rand() % 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 = rand % 500 + 1;
```

Empty Parameter Lists

The call to rand in our program looks a bit odd. You have seen function calls before, and you have seen parameter lists, but the call to rand is the first time you have seen a call to a function that doesn't need any parameters. In C, though, you have to put the (and) characters that mark the parameter list after the function name, even if the function doesn't need any parameters. The reason for this curious requirement is tied up in the way C defines functions and handles expressions. It turns out that C needs to see the (and) characters so it knows that the name is a call to a function, and not a reference to a variable. Later, when you learn a little more about how to create your own functions in C, and after you have learned more about something called scoping rules, we will return to this topic and explain why the (and) characters are needed. For now, just remember that you always have to have (and) characters after the function name for any function call, even if you aren't passing any parameters.

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 % operator in the table in the last section. The result of rand() % 100 will be zero when rand 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 rand 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 do loop can be used to do this. We haven't learned how to test for one condition *and* another yet, but by nesting two do loops, we can get the same effect.

```
/* pick a number from 1 to 100 */
do
   do
      value = rand();
   while (value <= 0);
while (value > 100);
```

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

```
do
   <get a number>
while (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,

```
   do
      value = rand();
   while (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 do loop – even another do loop. You can also put a for loop inside a while loop, a while loop in a do loop, a for loop inside of a while loop that is inside of a do 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/C, 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 four 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 do 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 do loops outlined above to pick the random number, rather than the % operator.

Next, add a do loop around the part of the program that picks a number and lets the player guess it. This do 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 do 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 do 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 do loop plays the game until the player wants to stop. Inside of this loop are two sets of nested do loops that make sure a number is in a particular range. There was already one do loop in the program, looping until the player guesses the right number. All together, that's six do 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; i <= 10000; ++i)
   for (j = 1; j <= 25; ++j)
      ;
```

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/C 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; i <= 100; ++i)
   for (j = 1; j <= 25; ++j)
      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.

```
/* A closer look at      */
/* pseudo-random numbers */

#include <stdio.h>
#include <stdlib.h>

unsigned i;  /* loop variable */
int seed;    /* seed */

void main(void)

{
printf("Random number seed = ");
scanf(" %d", &seed);
srand(seed);
for (i = 1; i <= 10; ++i)
   printf("%d\n", rand());
}
```

Type this program in 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 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 scanf

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

To look at how scanf reads more than one value at a time, we'll use another short program.

```
/* Multiple reads */

#include <stdio.h>

int n1,n2,n3,n4;

void main(void)

{
scanf(" %d %d", &n1, &n2);
scanf(" %d %d", &n3, &n4);
printf("%d %d %d %d\n",
   n1,n2,n3,n4);
}
```

At the beginning of this lesson, you learned that scanf skips all whitespace characters if the format string has a whitespace character. 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, since either is a whitespace character. If there are two numbers to read, as in our program, scanf 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. If there is more than one call to scanf, the second call picks right up with the first character that was not used by the first call to scanf.

You should try problem 3.7 to make sure you really understand all of the fine points and implications of what was just said.

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   3
4   5   6
```

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

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

fifth set:
```
1

2

3

4
```

Reading Floating-Point Numbers

So far, we have used the scanf function to read integers. It's just as easy to read floating-point numbers like 1.5, or 10.0. The only difference, in fact, is that the variable you are reading should be declared as a floating-point number, rather than an integer, and you use a different conversion specification.

When your program reads a floating-point 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 floating-point 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 floating-point 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-1
.1e1
```

There are two different kinds of floating-point numbers in standard C (and two more in Apple based implementations of C), and each type of floating-point number has a unique format specifier. This is because each of the floating-point number formats is stored a different way in memory, and you need to tell scanf which number format to use. Back in lesson 2, you used two difference conversion specifiers with floating-point numbers; f was used to print the number without an exponent, while e printed a value with an exponent. When you are reading numbers, you don't have to worry about the distinction. scanf will accept either conversion specifier and read any kind of floating-point number you choose to type. If you are reading a value into a double number, though, you need to put an l right before the f or e character. The l character is short for long float, which is an older way of telling C that you want a double value.

Problem 3.8. Modify the first sample program from this lesson to read floating-point numbers, rather than integers. To do this, all you have to do is change the variable i from an int to a float variable, and change the conversion specifiers in the scanf and printf calls. Run the program, giving it the numbers shown in the table, above.

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>)
   <statement>;
```

The if statement evaluates the same kind of condition that you have already used in the do and while statements. 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 { and } characters to group more that one statement into a compound statement, just as you do with the for loop, do 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 % operator you met earlier in the lesson, to write a program that can count change.

```
/* count change */

#include <stdio.h>

int change;

void main(void)

{
/* get the amount */
printf("How many cents? ");
scanf(" %d", &change);

/* write the header */
printf(
   "Your change consists of:\n");

/* count out the dollars */
if (change >= 100) {
   printf("%d dollars\n",
      change / 100);
   change %= 100;
   }

/* count out the quarters */
if (change >= 25) {
   printf("%d quarters\n",
      change / 25);
   change %= 25;
   }
```

```
/* count out the dimes */
if (change >= 10) {
   printf("%d dimes\n",
      change / 10);
   change %= 10;
   }

/* count out the nickles */
if (change >= 5) {
   printf("%d nickles\n",
      change / 5);
   change %= 5;
   }

/* count out the pennies */
if (change != 0)
   printf("%d pennies\n",
      change);
}
```

In this program, 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 { and } characters.

```
if (change >= 100) {
   printf("%d dollars\n",
      change / 100);
   change %= 100;
   }
```

If there are more than 100 pennies due, then we can start by giving out some number of dollars. Since the / 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 / 100, which is 5, as the number of dollars. The % operation then strips off the dollar amount: 536 % 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 an if-then-else statement, you can print something a bit prettier:

```
if (tries == 1)
   printf("You got it in 1 try!\n");
else
   printf("It took you %d tries.\n",
      tries);
```

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 C is a statement terminator, not a statement separator. In plain English, this means that the semicolon is used after every statement in a C program, even if the statement comes right before the else clause of an if statement. "Of course," you say, "how else would it work?" If you already know Pascal, Ada, Modula, or virtually any other language that uses semicolons, you know that these languages won't accept a semicolon before an else! It doesn't matter a whole lot which way the semicolon works, since both methods are in common use in various programming languages and both work just fine, but you should know that there is a difference.

Just for the record, the } character marks the end of a compound statement. What that means is that a semicolon is not needed after the } character.

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.

```
printf("That was your ");
if (try == 1)
   printf("1st");
else if (try == 2)
   printf("2nd");
else if (try == 3)
   printf("3rd");
else
   printf("%dth", try);
printf(" try!\n");
```

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

```
printf("That was your ");
if (try == 1)
   printf("1st");
else
   if (try == 2)
      printf("2nd");
   else
      if (try == 3)
         printf("3rd");
      else
         printf("%dth", try);
printf(" try!\n");
```

The second method shows more accurately what C 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>)" 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; i <= 1000; ++i) ;
```

Surprisingly, it really is important that you put the semicolon on the same line as the for statement. The C language really doesn't care, but in ORCA/C with debug code on, the debugger is called for each separate line to update the arrow in the program's window, and that takes quite a bit of time. If the semicolon is on the same line as the for statement, the debugger is not called.

Once the second step is complete and tested, it is time to move on to the third step. At this point, you should change the program so it asks the user for its inputs. The program should ask for a starting value for x and y, as well as starting values for three new variables, xSpeed, ySpeed and iter. iter is the number of times to move the ball; you should adjust the for loop accordingly. xSpeed and

ySpeed are the distance to move the ball in the x and y direction each time through the loop. You should change the statements that add one to x and y so xSpeed and ySpeed are added, instead. Be sure and get the program working before you move on to the next step.

At this point, stop and run the program with x and y set to 10, xSpeed set to 3, ySpeed set to 1, and iter set to 100. The ball will certainly move – it moves right off of the graphics window! The final step in developing the bounce program is to detect when this happens, and handle it. We'll look at how to do this for the x direction, and you can fill in the proper code for the y direction on your own.

```
if (x < 0) {
   x = 0;
   xSpeed = -xSpeed;
   }
else if (x > MAXX) {
   x = MAXX;
   xSpeed = -xSpeed;
   }
```

These statements should come right after the point where you update x by adding xSpeed. If the resulting value for x is less than zero, the you must have just moved the ball off of the left side of the graphics window. If the ball moved off of the left side of the graphics window, then of course, xSpeed must be negative. To bounce the ball back towards the middle of the graphics window, you need to set x to zero (putting the ball back on the screen) and reverse xSpeed (so the ball will move to the right). The next check does essentially the same thing, checking to see if x has moved past MAXX, which we set to 316 (the right side of the graphics window).

This program is probably more that you could have handled at this point if you had tried to write it all at once, and it probably wasn't easy to write even in small steps. This points out, once again, how important stepwise refinement is to the process of programming. Using stepwise refinement, you can break an impossible task up into smaller tasks that are merely difficult.

Lesson Three
Solutions to Problems

Solution to problem 3.1.

input	value read by scanf
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	Not what you typed! (17232)

Solution to problem 3.2.

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

```
/* Guess a number */

#include <stdio.h>
#include <stdlib.h>

int i;                                  /* input value */
int value;                              /* the number to guess */

void main(void)

{
/* Introduce the game */
printf("In this game, you will try to\n");
printf("guess a number.  I need a hint to\n");
printf("help me pick numbers, though.\n\n");

/* get a seed for the random number generator */
printf("Please type a number between 1\nand 30000: ");
scanf(" %d", &i);
srand(i);

/* pick a number from 1 to 100 */
do
   do
      value = rand();
   while (value <= 0);
while (value > 100);
```

```
/* let the player guess the number */
do {
   printf("Your guess: ");           /* get the guess */
   scanf(" %d", &i);
   if (i > value)                    /* check for too high */
      printf("%d is to high.\n", i);
   if (i < value)                    /* check for too low */
      printf("%d is to low.\n", i);
   }
while (i != value);                 /* if the guess was wrong then loop */
printf("%d is correct!\n", i);
}
```

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

```
/* Guess a number */

#include <stdio.h>
#include <stdlib.h>

int done;                           /* are we done, yet? */
int i;                              /* input value */
int value;                          /* the number to guess */

void main(void)

{
/* Introduce the game */
printf("In this game, you will try to\n");
printf("guess a number.  I need a hint to\n");
printf("help me pick numbers, though.\n\n");

/* get a seed for the random number generator */
printf("Please type a number between 1\nand 30000: ");
scanf(" %d", &i);
srand(i);
```

```
done = 3;
do {
   /* pick a number from 1 to 100 */
   do
      do
         value = rand();
      while (value <= 0);
   while (value > 100);

   /* let the player guess the number */
   do {
      printf("Your guess: ");           /* get the guess */
      scanf(" %d", &i);
      if (i > value)                    /* check for too high */
         printf("%d is to high.\n", i);
      if (i < value)                    /* check for too low */
         printf("%d is to low.\n", i);
      }
   while (i != value);              /* if the guess was wrong then loop */
   printf("%d is correct!\n", i);
   done -= 1;
   }
while (done != 0);
}
```

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

```
/* Guess a number */

#include <stdio.h>
#include <stdlib.h>

int done;                           /* are we done, yet? */
int i;                              /* input value */
int value;                          /* the number to guess */

void main(void)

{
/* Introduce the game */
printf("In this game, you will try to\n");
printf("guess a number.  I need a hint to\n");
printf("help me pick numbers, though.\n\n");

/* get a seed for the random number generator */
printf("Please type a number between 1\nand 30000: ");
scanf(" %d", &i);
srand(i);
```

```
do {
   /* pick a number from 1 to 100 */
   do
      do
         value = rand();
      while (value <= 0);
   while (value > 100);

   /* let the player guess the number */
   do {
      printf("Your guess: ");           /* get the guess */
      scanf(" %d", &i);
      if (i > value)                    /* check for too high */
         printf("%d is to high.\n", i);
      if (i < value)                    /* check for too low */
         printf("%d is to low.\n", i);
      }
   while (i != value);              /* if the guess was wrong then loop */
   printf("%d is correct!\n", i);

   do
      do {
         printf("Play again?  Enter 0\nfor no, 1 for yes:");
         scanf(" %d", &done);
         }
      while (done > 1);
   while (done < 0);
   }
while (done != 0);
}
```

Solution to problem 3.3.

```
/* Empty timing loop */

/* Note: be sure to run this program with debug turned off! */

#include <stdio.h>

unsigned i,j;                           /* loop variables */

void main(void)

{
printf("Start timer...\n");
for (i = 1; i <= 10000; ++i)
   for (j = 1; j <= 25; ++j)
      ;
printf("Stop timer.\n");
}
```

```
/* Time 250,000 integer additions */

/* Note: be sure to run this program with debug turned off! */

#include <stdio.h>

unsigned i,j;                           /* loop variables */
int a,b,c;                              /* test values */

void main(void)

{
a = 3;
b = 5;
printf("Start timer...\n");
for (i = 1; i <= 10000; ++i)
   for (j = 1; j <= 25; ++j)
      c = a+b;
printf("Stop timer.\n");
}


/* Time 250,000 floating-point additions */

/* Note: be sure to run this program with debug turned off! */

#include <stdio.h>

unsigned i,j;                           /* loop variables */
float a,b,c;                            /* test values */

void main(void)

{
a = 3.0;
b = 5.0;
printf("Start timer...\n");
for (i = 1; i <= 100; ++i)
   for (j = 1; j <= 25; ++j)
      c = a+b;
printf("Stop timer.\n");
}
```

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 411 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)	3.67	0.00001468	
3+5	5.35	0.00000672	
3-5	5.35	0.00000672	
3 / 5	64.30	0.00024252	
3*5	61.76	0.00023236	
3.0+5.0	694.00	0.00276132	411
3.0-5.0	695.00	0.00276532	412
3.0/5.0	1102.00	0.00439332	18
3.0*5.0	731.00	0.00290932	13

Solution to problem 3.4.

```
/* Throw two six-sided dice twenty times */

#include <stdio.h>
#include <stdlib.h>

#define SIDES 6                         /* # of sides on the dice */
#define NUMDICE 2                       /* # of dice to throw */

unsigned i,j;                           /* loop variables */
int seed;                               /* random number seed */

void main(void)

{
printf("Random number seed = ");
scanf(" %d", &seed);
srand(seed);

printf("\n");
for (i = 1; i <= 20; ++i) {
   for (j = 1; j <= NUMDICE; ++j)
      printf("%4d", rand() % SIDES + 1);
   printf("\n");
   }
}
```

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 */

#include <quickdraw.h>
#include <stdlib.h>

#define MAX 11651                       /* # of dots to fill */

unsigned i;                             /* loop counter */
unsigned x,y;                           /* coordinates of the point */

void main(void)

{
srand(5463);                            /* initialize for rand */
SetPenMode(0);                          /* set up for graphics */
SetSolidPenPat(0);
SetPenSize(1,1);

for (i = 1; i <= MAX; ++i) {
   x = rand() % 191 + 10;
   y = rand() % 60 + 10;
   MoveTo(x,y);
   LineTo(x,y);
   }
}
```

Solution to problem 3.6.

```
/* Gradually fill an area of the screen with colored dots */

#include <quickdraw.h>
#include <stdlib.h>

#define MAX 11651                       /* # of dots to fill */

unsigned i;                             /* loop counter */
unsigned x,y;                           /* coordinates of the point */

void main(void)

{
srand(5463);                            /* initialize for rand */
SetPenMode(0);                          /* set up for graphics */
SetPenSize(1,1);

for (i = 1; i <= MAX; ++i) {
   x = rand() % 191 + 10;
   y = rand() % 60 + 10;
   SetSolidPenPat(rand() % 4);
   MoveTo(x,y);
   LineTo(x,y);
   }
}
```

Solution to problem 3.7.

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

Solution to problem 3.8.

```
/* Read a float and write      */
/* it to the screen.           */

#include <stdio.h>

float i;

void main (void)

{
scanf(" %f", &i);
printf("%f\n", i);
}
```

Solution to problem 3.9.

```
/* Car Payment Calculator */

#include <stdio.h>

float APR;                              /* annual percentage rate */
float minPayment;                       /* minimum allowed payment */
unsigned month;                         /* number of months */
float payment;                          /* monthly payment */
float principal;                        /* amount left to pay */

void main (void)

{
month = 0;                              /* no payments made, yet */
printf("Cost of the car       ?");      /* get the cost of the car */
scanf(" %f", &principal);
printf("Annual percentage rate?");      /* get the interest rate */
scanf(" %f", &APR);
printf("Monthly payments      ?");      /* get the payment */
scanf(" %f", &payment);
minPayment = principal*APR/100.0/12.0;
if (payment <= minPayment)
   printf("Your minimum payment is %.2f\n", minPayment);
else
   while (principal > 0.0) {      /* keep going until we're out of debt */
      ++month;                          /* it's a new month */
      principal += principal*           /* add in this month's interest */
         APR/100.0/12.0;
      principal -= payment;             /* make the payment */
                                        /* write the status */
      printf("month = %3u   principal = %.2f\n", month, principal);
      }
}
```

Solution to problem 3.10.

step 1

```
#include <quickdraw.h>

#define MAXX 316                        /* graphics window size */
#define MAXY 83

unsigned i;                             /* loop counter */
unsigned x,y;                           /* coordinates of the point */

void main(void)

{
SetPenMode(0);                          /* set up for graphics */
SetPenSize(6,2);

x = 0;                                  /* initialize the ball position */
y = 0;

for (i = 0; i <= 50; ++i) {
   SetSolidPenPat(3);                   /* animate the ball */
   MoveTo(x,y);
   LineTo(x,y);
   ++x;
   ++y;
   SetSolidPenPat(0);
   MoveTo(x,y);
   LineTo(x,y);
   }
}
```

step 2

```
#include <quickdraw.h>

#define MAXX 316                        /* graphics window size */
#define MAXY 83

unsigned i,j;                           /* loop counters */
unsigned x,y;                           /* coordinates of the point */

void main(void)

{
SetPenMode(0);                          /* set up for graphics */
SetPenSize(6,2);

x = 0;                                  /* initialize the ball position */
y = 0;

for (i = 0; i <= 50; ++i) {
   SetSolidPenPat(3);                   /* animate the ball */
   MoveTo(x,y);
   LineTo(x,y);
   ++x;
   ++y;
   SetSolidPenPat(0);
   MoveTo(x,y);
   LineTo(x,y);
   for (j = 1; j <= 1000; ++j) ;         /* pause */
   }
}
```

step 3

```
#include <quickdraw.h>

#define MAXX 316                        /* graphics window size */
#define MAXY 83

unsigned i,j;                           /* loop counters */
int x,y;                                /* coordinates of the point */
int xSpeed,ySpeed;                      /* ball's speed */
unsigned iter;                          /* loop counter */

void main(void)

{
printf("Start x     :");                /* initialize the ball position */
scanf(" %d", &x);
printf("Start y     :");
scanf(" %d", &y);
printf("X speed     :");                /* initialize the ball speed */
scanf(" %d", &xSpeed);
printf("Y speed     :");
scanf(" %d", &ySpeed);
printf("iterations  :");                /* initialize the loop counter */
scanf(" %d", &iter);

SetPenMode(0);                          /* set up for graphics */
SetPenSize(6,2);

x = 0;                                  /* initialize the ball position */
y = 0;

for (i = 0; i <= iter; ++i) {
   SetSolidPenPat(3);                   /* animate the ball */
   MoveTo(x,y);
   LineTo(x,y);
   x += xSpeed;
   y += ySpeed;
   SetSolidPenPat(0);
   MoveTo(x,y);
   LineTo(x,y);
   for (j = 1; j <= 1000; ++j) ;         /* pause */
   }
}
```

step 4

```
#include <quickdraw.h>

#define MAXX 316                        /* graphics window size */
#define MAXY 83

unsigned i,j;                           /* loop counters */
int x,y;                                /* coordinates of the point */
int xSpeed,ySpeed;                      /* ball's speed */
unsigned iter;                          /* loop counter */

void main(void)

{
printf("Start x     :");                /* initialize the ball position */
scanf(" %d", &x);
printf("Start y     :");
scanf(" %d", &y);
printf("X speed     :");                /* initialize the ball speed */
scanf(" %d", &xSpeed);
printf("Y speed     :");
scanf(" %d", &ySpeed);
printf("iterations  :");                /* initialize the loop counter */
scanf(" %d", &iter);

SetPenMode(0);                          /* set up for graphics */
SetPenSize(6,2);

x = 0;                                  /* initialize the ball position */
y = 0;

for (i = 0; i <= iter; ++i) {
   SetSolidPenPat(3);                   /* animate the ball */
   MoveTo(x,y);
   LineTo(x,y);
   x += xSpeed;
   y += ySpeed;
   if (x < 0) {
      x = 0;
      xSpeed = -xSpeed;
      }
   else if (x > MAXX) {
      x = MAXX;
      xSpeed = -xSpeed;
      }
```

```
      if (y < 0) {
         y = 0;
         ySpeed = -ySpeed;
         }
      else if (y > MAXY) {
         y = MAXY;
         ySpeed = -ySpeed;
         }
      SetSolidPenPat(0);
      MoveTo(x,y);
      LineTo(x,y);
      for (j = 1; j <= 1000; ++j) ;          /* pause */
      }
   }
```

Lesson Four
Functions

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 a program that draws three rectangles on the graphics screen, filling each with a different color.

```
/* Draw three rectangles */

#include <quickdraw.h>

unsigned i;  /* loop variable */

void main(void)

{
/* set up for graphics */
SetPenMode(0);
SetSolidPenPat(0);
SetPenSize(3,1);

/* draw a black rectangle */
for (i = 10; i <= 60; ++i) {
   MoveTo(10,i);
   LineTo(250,i);
   }

/* draw a green rectangle */
SetSolidPenPat(1);
for (i = 31; i <= 49; ++i) {
   MoveTo(220,i);
   LineTo(270,i);
   }

/* outline it in black */
SetSolidPenPat(0);
MoveTo(220,30);
LineTo(220,50);
LineTo(270,50);
LineTo(270,30);
LineTo(220,30);

/* draw a purple rectangle */
SetSolidPenPat(2);
for (i = 41; i <= 79; ++i) {
   MoveTo(50,i);
   LineTo(300,i);
   }
/* outline it in black */
SetSolidPenPat(0);
MoveTo(50,40);
LineTo(50,80);
LineTo(300,80);
LineTo(300,40);
LineTo(50,40);
}
```

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 this:

```
/* draw a rectangle */
SetSolidPenPat(color);
for (i = top+1; i <= bottom-1;
   ++i) {
   MoveTo(left,i);
   LineTo(right,i);
   }
/* outline it in black */
SetSolidPenPat(0);
MoveTo(left,top);
LineTo(left,bottom);
LineTo(right,bottom);
LineTo(right,top);
LineTo(left,top);
```

Listing 4.1

```
/* Draw three rectangles */

#include <quickdraw.h>

void Rectangle (int left, int right, int top, int bottom, int color)

/* 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                   */

{
unsigned i;                     /* loop variable */

SetSolidPenPat(color);          /* draw a rectangle */
for (i = top+1; i <= bottom-1; ++i) {
   MoveTo(left,i);
   LineTo(right,i);
   }
SetSolidPenPat(0);              /* outline it in black */
MoveTo(left,top);
LineTo(left,bottom);
LineTo(right,bottom);
LineTo(right,top);
LineTo(left,top);
}

void main(void)

{
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 */
}
```

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, and then look at what is happening in detail.

The Structure of a Function

In one sense, the overall structure of the function Rectangle should look pretty familiar to you. After all, all of your programs have had at least one function, called main. There are a couple of new things here, though, so we'll take a very close look at this program.

Like the main function that you are familiar with, the Rectangle function doesn't return a value, so we start the definition with void. The name that comes next can be anything you like, so long as it isn't a reserved word, and so long as you haven't used the name in your program for something else. I like to capitalize the name of my functions like a title, but like all conventions, you can do anything you like. The one exception, of course, is main. Since C is a case sensitive language, I can't spell it as Main. Of course, whatever convention you use, you should be consistent.

The main function's parameter list is pretty simple. Since we don't pass any parameters to main, we use void between the parenthesis for the main function, and you would do that with your own subroutines if they did not have any parameters. The parameter list for Rectangle, though, is more complicated than the one in the main function, so we will need to spend a little more time talking about it. In the Rectangle function, the parameter list looks like this:

```
int left, int right, int top,
int bottom, int color
```

It is no accident that this looks suspiciously like a series of variable declarations. In fact, if you separate the variables with semicolons instead of commas, and moved the parameters to the top of the program, the compiler would be happy to take this parameter list as a series of variable declarations. What the parameter list actually does, in fact, is to define these variables within the function. Any statement within the function 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.

There is one major difference between the way that we define variables and the way that we write a parameter list. When you define variables, you know that you can define several at the same time with a single type, like this:

```
int left,right,top,bottom,color;
```

In a parameter list, the comma is used to separate the parameters themselves, so you have to specify the type again for each of the parameters.

The parameter list forms a sort of model that tells us how to call the function, as you can see by comparing the function declaration with a call from our sample program.

```
void Rectangle (int left,
   int right, int top, int bottom,
   int color)

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

On the top lines, you see the function declaration. The following lines show a call to the function, with spaces inserted to line up the matching components. C knows you are calling the function from the name itself. Since the compiler has already seen the definition of the function, it knows that Rectangle is a function, and that it needs five integer parameters. The compiler expects the values you pass as parameters to appear after the function 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 function 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 before the first statement of the function is executed:

```
/* in effect, what the */
/* compiler does       */
left = 10;
right = 250;
top = 10;
bottom = 60;
color = 0;
```

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

Local Variables

Right at the top of the Rectangle function, just after the { character that marks the start of the body of the function, you see the line

```
unsigned i;
```

In a sense, there is nothing new about this line; you have used lines just like it to define variables in all of your programs. The difference between this line and the variables you have defined in the past rests entirely with where the line is found. Instead of defining the variable before the start of the Rectangle function, this one is defined inside of the function itself. Variables defined inside of a function are called local variables.

There are several differences between local variables and global variables. Local variables are created when the function starts to execute, and the space that has been allocated for them goes away after the program leaves the function. Exactly how this is done is something we will cover later, but there is an important implication you should understand from the very beginning: if you assign a value to a local variable, the value will not necessarily be the same when you call the function again. (Later, you will learn how to get around this limitation.)

You might wonder why there are two different kinds of variables at all. The most important reason is that local variables give you more power in organizing your program into individual units. The local variable i in the Rectangle function belongs solely to that function; no other function sees it, no other function can change it, and other functions can have their own local variables, also called i, without interfering with Rectangle. A global variable, on the other hand, is available to all of the functions in your program. Global variables give you one way to share information between the various functions in the program.

In general, you should use local variables whenever you can, and global variables only when you have to. Global variables should only be used when a particular piece of information needs to be shared among several different functions. As your programming skills develop, the reason for this rule of thumb will become more and more clear.

How Functions are Executed

There are some subtle points about how functions 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 function 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 function (Figure 4.1). The next statement executed is the call to the function 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 function are executed just as they would be if they were in main. After the last statement is executed, you return to the first statement after the function call in the function main.

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

You can define variables in a function by declaring them as parameters or by declaring them right after the { character that marks the beginning of the function body. These variables are created when the function is called, and vanish when the function has finished executing. To see this, stop the program, or use Trace 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 global variables in main, you will find that you cannot look at a variable. You might try looking at the variable left, from the Rectangle function's parameter list, or i, defined with the function, to see this. The debugger will tell you that no such variable exists.

Continue stepping until you get to the first line of the function Rectangle. Watch the variables window as you enter the function: the name at the top of the variables window will change from main 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 function was called. It brings home the point that a variable should never be used before you assign a value to it.

While you are looking at the variables from any function, you cannot look at the global variables, or at the variables in another function. 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 global variables, or at the variables in another function, you click on the up arrow to move "up" one stack frame. To look at the variables from the function 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 function, so you are looking at the variables from that function. Continue stepping until you leave the function. As soon as you do, the variables window shifts back to main. The arrow can no longer be used to look at the variables from the function Rectangle. In fact, the variables literally do not exist anymore; the memory the compiler was using for the variables while the function was executing can be used for other purposes. When you step into the function 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 function 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.

Comments and Function Names

A program with one or two functions isn't likely to be too confusing, but as our programs use more and more functions, there are some 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 name of the function starts with a capitol letter. This helps remind me about the kind of identifier I am looking at when I read a program.

By far the most important convention, though, is how the function is commented. Right after the function declaration, and before the body of the function, I always put in a block comment. This block of comments tells what the function does. I use a very rigid format with up to four sections to comment a function. These sections are:

1. A description of the function, telling what the function 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 function declaration.

```
/* 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 function. 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 function 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.      */
```

As with the other formatting and commenting conventions mentioned in this course, there are many correct ways to comment a function 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.

Functions 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 function does, it is a lot easier to read the lines

```
Rectangle(10,250,10,60,0);
Rectangle(220,270,30,50,1);
Rectangle(50,300,40,90,2);
```

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);
SetSolidPenPat(0);
SetPenSize(3,1);
```

We can package these three lines into a function called InitGraphics, like this:

```
void InitGraphics(void)

/* Standard graphics initialization.                              */

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

With this new function, our program becomes even easier to read:

```
void main(void)

{
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 */
}
```

It may not be obvious yet, but there is still one more advantage to packaging even these three simple commands into a function. 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 function, you can easily update the function, 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 function 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 function 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; row <= 8; ++row)
   for (column = 1; column <= 8;
      ++column)
      <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 Functions

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 lots of functions. 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 the Rectangle function. 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 function. In our sample, for example, there are three calls to the Rectangle function. Having satisfied yourself that the function 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 function at full speed, returning you to single step mode after you get back to the main program.

Functions Can 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 functions, 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, so we need a way to get a value back from the function. To accomplish this, we need to do two things: declare a return type for the function, and return a value from the function. The program in listing 4.2 demonstrates this idea by packaging our random number generator.

There are really only two differences in the way you write a function that returns a value, and a function that returns void. The first shows up in the function declaration, which starts with the type unsigned, rather than the type void that you are used to. This type tells the compiler what it is that the function returns. You could tell the compiler that the function returns an int, a float, a double, a long int, or any other type you choose.

At some point, you need to specify what value the function should return. This is the second difference between a function that returns a value and a function that returns void. So far, all of our functions have exited at the bottom, after all of the statements in the function have been executed. To return a value from a function, though, you use the return statement. The return statement is followed by an expression that calculates the value you want to return. In our function, this is fairly complicated, but the value you return could simply be a variable, or even a constant.

You can put as many return statements in your function as you like, but you should be careful to create your function so that it will always exit via a return

Listing 4.2

```
/* This program rolls two dice 20 times.                              */

#include <stdio.h>
#include <stdlib.h>

#define SIDES 6                 /* sides on a die */
#define ROLLS 20                /* times to roll */
#define NUMDICE 2               /* number of dice to roll */


unsigned RandomValue (unsigned max)

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

{
return rand() % max + 1;
}


void main(void)

/* main program                                                       */

{
unsigned i,j;                   /* loop variables */
unsigned sum;                   /* number of spots showing */

srand(1234);                /* initialize the random number generator */
for (i = 1; i <= ROLLS; ++i) {
   sum = 0;
   for (j = 1; j <= NUMDICE; ++j)
      sum += RandomValue(SIDES);
   printf("%d\n", sum);
   }
}
```

statement. If you mess up, the function will still return after it executes the last statement in the function, but the value that is returned is completely unpredictable.

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

```
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 += 4;
```

Problem 4.2. You can use a function anywhere you can use a value in C. In particular, you can use a function to set the value of a parameter for another function call.

Use this idea 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 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) {
   temp = left;
   left = right;
   right = temp;
   }
```

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.

More about void and return

The return statement you just used in the Rectangle function is obviously necessary when you need to return a value from a function, but it can also be used in functions that return void. In fact, now is a good time to stop and think about what it means to "return void" from a function.

In many programming languages, there are two completely separate kinds of subroutines. In Pascal, for example, they are called functions and procedures; in FORTRRAN, they are subroutines and functions. C only has one kind of subroutine, the function. Conceptually, every C subroutine returns something. Pascal's procedures and FORTRAN's subroutines are the equivalent of a C function returning void; in C, then, void is used to tell the compiler that there really isn't anything to return.

That may seem a bit strange, and in a way it is, but there are some advantages to the way functions are defined in C. In C, you don't have to do anything with the value a function returns. For example, the printf function you have been using actually returns an integer; this integer is the number of characters actually printed by printf. You don't usually need to know this value, so in our programs, we don't assign the value returned by printf to anything. If we wanted to check to see how many characters had actually been written, though, we could. In a language like Pascal or FORTRAN, the language always checks to be sure we did something with the value returned by the function. In those languages, if you want an output subroutine that can return the number of characters written, but you also want an output subroutine that you can use without assigning the result to a variable, you would have to have two different subroutines. In C, we only need one.

In early versions of C, that's the way it was left. You could (and still can) choose to use the return statement with an expression, or without an expression. If you return from a function by dropping out of the bottom after executing the last statement, or from a return statement with no expression after it, the value returned by the function is unpredictable. If you return from a function by executing a return statement that is followed by an expression, the value of the expression itself is returned.

The idea of a function returning void is actually fairly new in the C language. There are two reasons for giving every function a type like this. First off, the compiler is able to do a few more checks that it could do before, preventing some programming errors. For example, if you try to assign the value returned by a void function to a variable, the compiler can flag an error. Calling the InitGraphics function from earlier like this:

```
i = InitGraphics;
```

would generate an error, since the compiler recognizes that you defined InitGraphics as returning void (or nothing), but clearly, your are trying to assign the value returned by the function to an integer variable.

A less important reason to use void, but one that is still crucial in some time-critical applications, is that ORCA/C must create some code to return something from a function. If you define a function as returning void, ORCA/C recognizes this fact, and doesn't create the machine language instructions to return the value. This may not be true on other compilers, and the difference in size and speed between a function

returning int and a function returning void isn't enormous, but there is a difference.

A First Look at Pointers

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.

To handle this situation, we will need to learn a whole new way of getting at a value. Up until now, all of the values we have used have been stored in variables. To define a variable, we list the type of the variable, and then its name. To use a variable in an expression, we type the name of the variable. To assign a value to a variable, we use it's name on the left-hand side of an equal sign. In all of these cases, we are being very specific about exactly which variable we want to define, use or set.

Pointers are used for many things, but one of them is to create subroutines that can set the value of several different variables, depending on what variable we pass to the function. The idea of a pointer is really fairly simple. Instead of accessing a variable by name, we access it by the address.

Let's try a simple program to see how this works.

```
#include <stdio.h>

void main(void)

{
int i, j, *ptr;

i = 4;
ptr = &i;
j = *ptr;
printf("%d %d\n", *ptr, j);
}
```

The first line in main defines a total of three variables, i, j and ptr. The first two, i and j, look just like the countless other int variables you have defined. The last one, ptr, is a bit different, though. The * character that comes right before the variable name tells us that we aren't really defining an int variable at all; instead, we are defining a *pointer to int.* ptr doesn't hold an integer value, like i and j do. Instead, ptr holds the address of an integer value.

The line

```
ptr = &i;
```

assigns a value to ptr. Since ptr is a pointer to an integer, we need to give it the address of an integer value. Remember the scanf function from the last lesson? In scanf, we also needed the address of a variable, since scanf wanted to store something at the address.

The next line assigns a value to j, which is an int variable, so we need to assign an integer value. The pointer ptr points to an integer, though; ptr is not itself an integer. To get at the integer ptr points to, we again use the * character right before the variable name, just like we did in the variable declaration. In fact, as with a function declaration, a pointer declaration is also a model showing you how to use the pointer in a statement. To access the pointer, whether you want to set it or copy it, you use the name of the variable. To access the value the variable points to, put the * character right before the variable name.

At this point, j is also 4, since we first set i to 4, then set ptr to point to i, then set j to whatever ptr pointed to. The printf statement confirms this, printing the value ptr points to and the value of j.

As the course progresses, you will learn to use pointers for more and more things, but the reason for bringing them up now is to show you how to write your own functions that, like scanf, change the value of a variable, instead of just using the value. You see, when you pass an int value like i to a function, the compiler actually passes the value in the variable i to the function. The function itself has no idea where it came from, and never changes the original value you passed. By passing the address, though, in the form of a pointer, the function can change the value the pointer points to. To see how this works, we'll redo the program from problem 3.10 that bounced a ball around the graphics window. In this program, though, we'll encapsulate the code to move a ball into a function called MoveBall, and pass this function the position and velocity of the ball. The function will then change these values and return.

Listing 4.3

```
/* Bounce a ball */

#include <quickdraw.h>
#include <stdio.h>

#define MAXX 316                    /* size of the screen */
#define MAXY 83


void MoveBall(int *x, int *y, int *vx, int *vy)

/* Move a ball in the graphics window.  If the ball   */
/* hits one of the sides (defined by the constants    */
/* MAXX and MAXY), the direction of the ball is       */
/* changed.                                           */
/*                                                    */
/* Parameters:                                        */
/*    x,y - position of the ball                      */
/*    vx,vy - velocity of the ball                    */

{
SetSolidPenPat(3);                  /* erase the ball */
MoveTo(*x,*y);
LineTo(*x,*y);
*x += *vx;                          /* move in the x direction */
if (*x < 0) {                       /* check for off the edge */
   *x = 0;
   *vx = -*vx;
   }
else if (*x > MAXX) {
   *x = MAXX;
   *vx = -*vx;
   }
*y += *vy;                          /* move in the y direction */
if (*y < 0) {                       /* check for off the edge */
   *y = 0;
   *vy = -*vy;
   }
else if (*y > MAXY) {
   *y = MAXY;
   *vy = -*vy;
   }
SetSolidPenPat(0);                  /* draw the ball in the new spot */
MoveTo(*x,*y);
LineTo(*x,*y);
}
```

```
void main(void)

/* main program                                        */

{
unsigned i,j;                       /* loop counters */
unsigned iter;                      /* # of movements of the ball */
int x,y;                            /* coordinates of the point */
int xSpeed,ySpeed;                  /* speed the ball */

SetPenMode(0);                      /* set up for graphics */
SetPenSize(6,2);

printf("Start x     :");           /* initialize the ball position */
scanf(" %d", &x);
printf("Start y     :");
scanf(" %d", &y);
printf("X speed     :");           /* initialize the ball speed */
scanf(" %d", &xSpeed);
printf("Y speed     :");
scanf(" %d", &ySpeed);
printf("iterations  :");           /* initialize the loop counter */
scanf(" %d", &iter);

if (x < 0)                         /* restrict the initial ball position */
   x = 0;
else if (x > MAXX)
   x = MAXX;
if (y < 0)
   y = 0;
else if (y > MAXY)
   y = MAXY;

for (i = 1; i <= iter; ++i) {      /* animate the ball */
   for (j = 1; j <= 1000; ++j) ;   /* pause for a bit */
   MoveBall(&x,&y,&xSpeed,&ySpeed); /* move the ball */
   }
}
```

In our program, the MoveBall function is used to update the position of the ball on the screen. We pass four values to the MoveBall function; 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 function, so they are all declared as pointers when MoveBall is declared, and we actually pass the address of the variable, not the variable itself, when MoveBall is called. Just as with scanf, when we need to pass the address of a variable, we put a & character right before the name of the variable. Inside of MoveBall, each time we use one of the values, we put a * before the name of the pointer, telling the compiler that we want the value the pointer points to, not the address contained in the pointer variable itself.

Run the program in step mode, and step up to the first line of the MoveBall function. Bring up the variables window, and take a look at the values for x, y, vx and vy in the function. You will see a strange value in the display. What you are seeing is the actual address passed as a parameter, shown in a form of notation called hexadecimal. (We'll look closer at

hexadecimal numbers later.) To see the value the pointer points to, place a ^ character after the name of the variable in the variables window. As you step through the function, changing the values for the parameters, switch back to the main program 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.

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.

Some Archaic Features of C

In this section, we will take a first look at some ancient C. If you were ever to write a new program using any of these techniques, and happened to do it in a Catholic school, some nun would take a swing at your knuckles with a yardstick. Like all other proponents of structured programming, I would sit in the back and cheer her on. Why, then, do you need to know all of this stuff? Why, in fact, does C allow it at all?

Let's look at the issue of why these things even exist in C first. While C's popularity is fairly recent, the language itself has been around for a long time, and the earliest versions of C were based on a language (B, believe it or not) that is even older. C was originally developed for computers that were very small by today's standards, too, so the original language had to be small. As C has grown in popularity, it has also grown in size, with many new features added fairly recently. On the other hand, there are a lot of programs laying around that are still very useful that were written with these early versions of C. The modern C language has been extended very carefully so that almost all of these old programs, assuming they were written well, can still be compiled from a modern C compiler. The older versions of C are collectively referred to as K&R C, taken from the authors of the first good reference manual for the C language, Kernighan and Ritchie. The modern C language is generally referred to as ANSI C, after the name of the standard for the language, published by ANSI (the American National Standards Institute).

In K&R C, there was no such thing as void. In other words, all functions returned some value. Just as with ANSI C, though, you could use a return statement with no value, or simply drop out of the bottom of the function without returning a value. And, just as with ANSI C, you didn't have to make use of the value that was returned. By convention, when a function didn't return anything, people defined the function as returning int. In fact, the type of a function is optional – if you leave the type off entirely, C compilers assume you want a function that returns int.

In K&R C, you also can't specify the types of the parameters in the parameter list, or even put void in the parameter list if there are no parameters. If there are parameters, you list the names of the parameters in the parameter list, then define the variables between the declaration of the function and the { character that starts the body of the function. For example, with the statements and comments removed, the MoveBall function would look like this in K&R C:

```
MoveBall(x, y, vx, vy)

int *x, *y, *vx, *vy;

{
/* statements go here */
}
```

Functions also don't have to be defined before they are used, as long as the function returns an int. In all of your programs that used printf, for example, you have included the header file stdio.h, which defines printf. The printf function returns an integer, though, so the program will actually work the same way if you leave the include out. Taking this all the way, our original program that wrote "Hello, world" to the screen looks like this:

```
main()

{
printf("Hello, world.\n");
}
```

Go ahead and type it in and try the program – even though it is written in the older, K&R C style, ORCA/C will still compile the program, just as the old C compilers did.

In some ways, all of this must seem a lot simpler than typing void everywhere, remembering to include the correct header files, and remembering to define functions before they are used. There are some serious

problems with K&R C, though. Take a look at the following program, but don't type it in or run it.

```
main()

{
foo(65536L);
}

foo(a,b)

int a,b;

{
printf("%d %d\n", a, b);
}
```

Although someone who writes good, structured code would shudder a bit reading this program, it is actually legal C, in the sense that any C compiler will accept it and create a program from it. On the vast majority of C compilers that, like ORCA/C, store integers in two bytes, and long values in four bytes, the program will even work, and do something almost sensible: it breaks the long integer into two int values, printing the two halves of the four-byte long integer as two two-byte integers.

Just because it works, though, doesn't mean it is a good program. What this program points out is that programs written with K&R style function declarations are dangerous. The compiler can't check to make sure you are passing the correct size value to a function, or that the function returns the correct type of result. In fact, the compiler can't even check to make sure you pass the correct *number* of parameters! And what happens if you make a mistake? According to all of the C standards, from the original description by Kernighan and Ritchie to the most recent ANSI C standard, the results are undefined. That's a polite way of saying that the compiler can do whatever it wants, including producing a program that crashes.

There are some good reasons why a very advanced system programmer might want to make use of some of these peculiarities of C. When they are needed, these freedoms can be used by a clever programmer to do things that literally can't be done in other programming languages. For example, C's printf function literally can't be written in Pascal, FORTRAN, or BASIC, but it can be written in C. On the other hand, for the vast majority of programming situations, it is comforting to have the compiler checking to make sure that we don't do stupid things, like passing the wrong number of parameters to a function. That's why ANSI C includes void and the ability to define the number and types of parameters explicitly. Unless you have a very, very good reason for using the freedom of K&R parameters, and know exactly what you are doing, you should always define the type of every function, declare the parameters explicitly in the parameter list, and define the function before you use it.

That is, of course, if you value your knuckles.

Lesson Four
Solutions to Problems

Solution to problem 4.1.

```
/* Draw a checker board                                          */

#include <quickdraw.h>

unsigned color;                         /* color of the square */


void InitGraphics(void)

/* Standard graphics initialization.                             */

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


void Rectangle (int left, int right, int top, int bottom, int color)

/* 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                   */

{
unsigned i;                             /* loop variable */

SetSolidPenPat(color);                  /* draw a rectangle */
for (i = top+1; i <= bottom-1; ++i) {
   MoveTo(left,i);
   LineTo(right,i);
   }
SetSolidPenPat(0);                      /* outline it in black */
MoveTo(left,top);
LineTo(left,bottom);
LineTo(right,bottom);
LineTo(right,top);
LineTo(left,top);
}
```

```
void SwapColor(void)

/* Changes the color between black and white.                    */
/*                                                               */
/* Variables:                                                    */
/*    color - color to change                                    */

{
if (color == 3)
   color = 0;
else
   color = 3;
}


void main(void)

{
unsigned row,column;                    /* position of the square */

InitGraphics;                           /* set up the graphics window */

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

Solution to problem 4.2.

```
/* Draw a random number of rectangles */

#include <quickdraw.h>
#include <stdlib.h>

#define MAXX 316                  /* size of the screen */
#define MAXY 83
```

```
unsigned RandomValue (unsigned max)

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

{
return rand() % max + 1;
}


void Rectangle (int left, int right, int top, int bottom, int color)

/* 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                   */

{
int i;                                  /* loop variable */

SetSolidPenPat(color);                  /* draw a rectangle */
for (i = top+1; i <= bottom-1; ++i) {
   MoveTo(left,i);
   LineTo(right,i);
   }
SetSolidPenPat(0);                      /* outline it in black */
MoveTo(left,top);
LineTo(left,bottom);
LineTo(right,bottom);
LineTo(right,top);
LineTo(left,top);
}


void InitGraphics(void)

/* Standard graphics initialization.                            */

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

```
void main(void)

{
unsigned numRects;                      /* # of rectangles to draw */
unsigned i;                             /* loop variable */
unsigned left,right,top,bottom;         /* sides of the rectangle */
unsigned temp;                          /* used to swap values */

InitGraphics;                           /* set up the graphics window */
srand(234);                  /* initialize the random number generator */

numRects = RandomValue(30);            /* decide how many rects to draw */
for (i = 1; i <= numRects; ++i) {
   left = RandomValue(MAXX);            /* get the left, right */
   right = RandomValue(MAXX);
   if (left > right) {
      temp = left;
      left = right;
      right = temp;
      }
   top = RandomValue(MAXY);             /* get the top, bottom */
   bottom = RandomValue(MAXY);
   if (top > bottom) {
      temp = top;
      top = bottom;
      bottom = temp;
      }
                                        /* draw the rectangle */
   Rectangle(left,right,top,bottom,RandomValue(4)-1);
   }
}
```

Solution to problem 4.3.

```
/* Bounce a ball */

#include <quickdraw.h>
#include <stdlib.h>

#define MAXX 316                    /* size of the screen */
#define MAXY 83


unsigned RandomValue (unsigned max)

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

{
return rand() % max + 1;
}


void MoveBall(int *x, int *y, int *vx, int *vy)

/* Move a ball in the graphics window.  If the ball   */
/* hits one of the sides (defined by the constants    */
/* MAXX and MAXY), the direction of the ball is       */
/* changed.                                           */
/*                                                    */
/* Parameters:                                        */
/*    x,y - position of the ball                      */
/*    vx,vy - velocity of the ball                    */

{
SetSolidPenPat(3);                  /* erase the ball */
MoveTo(*x,*y);
LineTo(*x,*y);
*x += *vx;                          /* move in the x direction */
if (*x < 0) {                       /* check for off the edge */
   *x = 0;
   *vx = -*vx;
   }
else if (*x > MAXX) {
   *x = MAXX;
   *vx = -*vx;
   }
```

```
*y += *vy;                          /* move in the y direction */
if (*y < 0) {                       /* check for off the edge */
   *y = 0;
   *vy = -*vy;
   }
else if (*y > MAXY) {
   *y = MAXY;
   *vy = -*vy;
   }
SetSolidPenPat(0);                  /* draw the ball in the new spot */
MoveTo(*x,*y);
LineTo(*x,*y);
}


void main(void)

/* main program                                        */

{
unsigned i,j;                       /* loop counters */
unsigned iter;                      /* # of movements of the ball */

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

SetPenMode(0);                      /* set up for graphics */
SetPenSize(6,2);
srand(1234);                  /* 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; i <= 100; ++i) {
   MoveBall(&x1,&y1,&vx1,&vy1);
   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);
   }
}
```

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. For example, an array can hold 1000 int values, or 200 float values, but an array cannot hold both int and float values at the same time. 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. In C, when you define an array, you specify how many things you want the array to hold – in this case, 11. The array is defined like this:

```
int totals[11];
```

This is very similar to the way we define integers. In fact, the only difference is that we have inclosed a number in brackets right after the name of the variable. As you might guess, this means that the array holds integers. It's just as easy to define an array that holds 100 double variables, though:

```
double values[100];
```

In the last lesson, you saw how to define a pointer to an integer by placing an * character before a variable name, and you even saw a definition where integers and pointers to int were mixed on the same line. You can also mix arrays of int with these other types, like this:

```
int totals[11], i, *iptr;
```

This line defines a total of three variables. The first, totals, is an array of 11 int values. The variable i is a simple int variable, like the ones you have used since the second lesson of the course, while the final variable is a pointer to int, like the ones you saw in the last lesson.

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. All arrays have indexes that start from 0 and count up to the number of elements in the array minus one. To set the first element of the array to zero, for example, we would write

```
totals[0] = 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 = 0; i < 11; ++i)
   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 function. There are very few cases, though, where you can use the entire array. You can't write an array using printf, for example. We will explore when and how you can use an entire array as we get to know arrays better.

Before you run the 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 nearly five minutes! We will look at the reasons in a moment.

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.                                                       */

#include <stdio.h>
#include <stdlib.h>

#define ROLLS 10000             /* times to roll */

int totals[11];                 /* number of spots showing */


unsigned RandomValue (unsigned max)

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

{
return rand() % max + 1;
}


void Simulation(void)

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

{
unsigned i;                         /* loop variable */
unsigned sum;                       /* # of spots on this roll */

for (i = 0; i < 11; ++i)            /* set the totals to zero */
   totals[i] = 0;
for (i = 1; i <= ROLLS; ++i) {
   sum = RandomValue(6)+RandomValue(6); /* roll the dice */
   ++totals[sum-2];                 /* increment the correct total */
   }
}
```

```
void WriteArray(void)

/* Write the results.                                                 */
/*                                                                    */
/* Variables:                                                         */
/*    totals - array holding the total number of rolls                */

{
unsigned i;                          /* loop variable */

printf("   spots   times\n");
for (i = 0; i < 11; ++i)
   printf("%8d%8d\n", i+2, totals[i]);
}


void main(void)

/* Main program.                                                      */

{
srand(1234);                   /* initialize the random number generator */
Simulation();                   /* do the dice simulation */
WriteArray();                   /* write the dice array */
}
```

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[0], totals[1], 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 nearly five minutes. In fact, it runs for 4 minutes, 48 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 10.06 seconds. That's quite a difference. In fact, that is over 28 times faster than 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.

Be Careful With Arrays!

Arrays are a wonderfully powerful way to handle large numbers of variables, but arrays do have some drawbacks in C. As an example, let's look at a very short program, and try to understand what it is doing. Don't run this program, though, until you have read the explanation of what it does.

```
/* Don't run this program yet! */

int a[2];

void main(void)

{
int i;

for (i = 0; i < 32767; ++i)
   a[i] = 0;
}
```

In this program, you are filling an array with zeros. The array a, defined at the top of the program, is made up of two int values; each of these requires two bytes of storage.

In the function main, the for loop starts off with i set to 0. This stores a 0 into the first two bytes that the compiler set aside for the array when it was defined. The next time through the loop, a[1] is set to zero, using the last two bytes the compiler set aside. Unfortunately, the loop doesn't stop. It goes right on to set a[2] to zero, changing the two bytes that come right after the array. These bytes were not reserved by the compiler for use in the array. In fact, they make up part of your program. As the program continues to run, you wipe out more and more of your program, perhaps along with some of the operating system, a desk accessory or two, the text that the development environment is displaying in your source window, and so forth. Eventually, the program will crash – probably by entering the monitor, which will display a confusing text screen that shows some information about the state of the 65816 processor the Apple IIGS uses, as well as a little about the contents of the memory; or by hanging, where the computer simply stops responding to you.

This is something that will happen to you over and over again as you program, so you might as well see it happen once now, while you are expecting it. Go ahead and type this program in and run it. Be sure you save anything on your desktop before you run it, though – once you crash, there is no practical way to recover, other than rebooting.

The point is that you can define an array to be whatever size you want, but the C language does not have a mechanism for making sure that you don't make mistakes. It's up to you to make sure that you keep the array subscript in the proper range. If you make a mistake, the results can be, well, dramatic.

The error you just saw was a pretty obvious one, but there is a similar mistake that, in practice, is made far more often. Let's take a look at an example:

```
int a[2];

void main(void)

{
int i;

for (i = 1; i <= 2; ++i)
   a[i] = 0;
}
```

To a BASIC programmer, this looks like a pretty reasonable program. You define an array with two elements, then fill in a[1] and a[2] with a zero. Remember, though, that C starts all arrays with a first index of 0, not 1! In addition, the 2 in a[2] is really the size of the array, not the value of the largest valid subscript. For this array, the proper way to initialize it looks like this:

```
int a[2];

void main(void)

{
int i;

for (i = 0; i < 2; ++i)
   a[i] = 0;
}
```

Take a close look at the for loop: it starts off with i set to 0, loops with i set to 1, and then stops, because, on the next loop, i would be 2, but in that case i is not less than 2. This is a very common C programming technique. It recognizes our natural tendency to want to use the size of the array in the condition that stops the for loop, but it also recognizes that a[2] does not exist. You can see this technique used in our dice program, for example, when the totals array is initialized:

```
for (i = 0; i < 11; ++i)
   totals[i] = 0;
```

Why Programmers are Humble – At Least, in Private!

By now, you probably feel like this issue is being beat to death, but it is a very common source of errors in C programs, so I would like for you to try one other thing. Here's the sample program that indexes one array element too far again:

```
int a[2];

void main(void)

{
int i;

for (i = 1; i <= 2; ++i)
   a[i] = 0;
}
```

At this point, I want you to type the program in and run it. Don't worry – you won't have to reboot again!

So what happened? Nothing, right? If this is such a bad error, why didn't anything happed?

The answer is simple, and it also shows just how careful you have to be as a programmer. The program did, in fact, wipe out two very important bytes in your program, but as it happened, the bytes had already been used, and were no longer needed. If, by chance, the bytes had been critical, the program would have crashed. If you were to add a call to printf right after initializing the array, for example, the program might crash.

The point this illustrates is that a bug doesn't always show up right after you make the mistake. In C, it is very possible for a bug like this one to stay hidden by chance arrangements of memory until you make an apparently innocent change to your program at some later date. The natural tendency of all beginning programmers (yes, I did it, too) is to look at the new code, correctly note that it looks just fine, and maybe even test it in a separate program. When the new code checks out, the claim is: "The computer made a mistake!" Or, perhaps one of the other variations: "The compiler/operating system has a bug!"

Right. The Apple IIGS operating system is not perfect, and neither is ORCA/C, but if your program crashes, the best place to start to look for the problem is in the mirror. This is a hard and humbling lesson for all new programmers to learn, but it is worth learning it quickly.

Strings are Arrays

You may have noticed 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 C are actually arrays of characters. Now that you are learning about arrays, you can also learn a great deal about strings.

In C, a string is simply an array of characters. Characters are a new data type; in C, you use the name char for characters to save some typing, just as you use int instead of integer. For example, the array

```
char name[10];
```

could be used to hold a name. It can hold Wirth, Jobs or Allred, but it is limited to names with 9 or fewer characters. This string can't hold Westerfield, since that name has 11 characters.

OK, you've learned your lesson about arrays, and you know that the array can only hold 10 char values, so why did I say that the longest string this array can hold is 9 characters long? The answer lies in how C represents a string internally. When printf prints a string, or when any of the functions you will eventually learn about use a string, they need to know how long the string is. The functions don't have any idea how big the array itself is, so C uses a special character called the null character to mark the end of the string. The null character is generally put into strings by C automatically, but you have to remember to leave room for it, which is why an array that holds ten char values is only long enough to hold a string with 9 characters. You have to leave one extra character so there is room for the null character that marks the end of the string.

Let's take a look at a string in a real program, as shown in listing 5.2.

Listing 5.2

```
/* This program reads in a string, reverses the order of the       */
/* characters, and writes the string back to the shell window.     */

#include <stdio.h>
#include <string.h>

#define MAX 256                 /* max length of the input string */

char inString[MAX],             /* input string */
     outString[MAX];            /* output string */
```

```
void Reverse()

/* Reverse a string                                    */
/*                                                     */
/* Variables:                                          */
/*    inString - string to reverse                     */
/*    outString - reversed string                      */

{
int i;                          /* loop variable */
int index;                      /* index into the reversed string */
int len;                        /* length of the input string */

len = strlen(inString);         /* find the length of the input string */
index = len;                    /* reverse the string's characters */
for (i = 0; i < len; ++i) {
   --index;
   outString[index] = inString[i];
   }
outString[len] = 0;             /* set the terminating null character */
}


void main(void)

/* Main program.                                       */

{
printf("String  :");            /* get a string */
scanf("%s", inString);
Reverse();                      /* reverse the string */
printf("Reversed:%s\n", outString); /* write the reversed string */
}
```

There are close to a whole bunch of new things in this program, but before we start to explore them, go ahead and run the program so you know what it does.

Back already? OK, let's start by looking at the main program. In particular, I'd like for you to take a close look at the scanf call:

```
scanf("%s", inString);
```

At first glance, this call looks innocent enough, but on closer inspection there are some peculiarities that point out some subtleties about the C language. The first is the absence of the & operator before the name of the string. By now, you know that all parameters to scanf, except for the original format string, of course, must be addresses, and you are used to using the & operator to get the address of a variable. In C, though, arrays and pointers have a unique relationship. This isn't something I'd like to confuse you with all at once, but the first thing that you should learn is that when you pass an array as a parameter to a function in C, the C language automatically converts the name of the array to the address of the array. In other words, you don't need the & operator before the name of an array, and in fact, it is a mistake to put it there. Fortunately, this is one mistake a C compiler can catch, so if you forget, you will be warned.

The other new things about this call are considerably more predictable. Since the program is reading a string, there must be a new conversion specifier, and in fact you see %s. The %s conversion specification also appears in the printf format string, when outString is written.

The last difference between this scanf call and the ones you have made before is that there is no whitespace character before the conversion specifier. This is because the %s conversion specifier actually skips whitespace on its own before it starts to read a string. When it is reading a string, scanf starts by skipping all whitespace characters, then it reads characters until it gets to the next whitespace character. In other words, it will read symbols from the input line one word at a time.

Inside the Reverse function is a call to a new library function, strlen. The strlen function is part of a very powerful string processing library called string.h. At the top of the program, you see a new include directive to include the header file that defines all of the functions in this library. This powerful, standardized string library makes C the best choice of any of the common high-level languages for string processing. You will see several functions from this library in the course.

The strlen function finds the length of a string and returns it. It works by counting all of the characters in the string up to the terminating null character. The string "hello", for example, would have a length of five, since there are five characters before the terminating null character that C puts at the end of all strings (yes, even at the end of string constants!).

Characters and Integers are Related

The characters you use in an array to form a string variable are more closely related to integers in C that in most languages. In fact, as far as the C language is concerned, a character *is* an integer. The only difference between a char variable and an int variable is that characters take up less room than integers, and therefore have a smaller range. In fact, char variables can hold integer values from -128 to 127. (This can change from compiler to compiler, but this range for characters is the most common one. The other common alternative is a range of 0 to 255, which you can get in ORCA/C by using the type unsigned char instead of char.) You can even use the %d conversion specification in printf to print the integer value of a character! To print an integer as a character, you can use the conversion specifier %c, which is the conversion specifier normally used to print char variables.

This special relationship between characters and integers really comes into play when you ask what it means to compare two characters. The integer value associated with each character changes from computer to computer, but some definition is essential. After all, you can get pretty good agreement from most people whether the character a is less than the character b, but things get a little less definite when you ask if the character ^ is less than the character *. On the Apple IIGS, and most other desktop computers, the integer that is associated with each character is defined by the ASCII character set. The ASCII character set 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, while 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 5.1 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.

Table 5.1

	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

Problem 5.1. In this section, two claims were made: first, that you could use %c to print the character that corresponds to an integer, and second, that the integer associated with the letter A is 65. Test these facts by writing a program that prints the integer value 65, but uses the conversion specifier for a character. (To save you from searching through the text, the conversion specifier for a character is %c.)

Problem 5.2. The ASCII character set consists of two kinds of characters, called the printing characters and special characters. The special characters are used for special purposes which you will learn about gradually. The printing characters are the characters starting with the space character (with an integer value of 32) and continuing through the ~ character (with an integer value of 126).

Write a program that prints the printing ASCII characters in the shell window. Your program should print sixteen characters on each line except the last line, which will have fifteen characters.

(Hint: Use two nested for loops, one of which ranges from 0 to 5, and the other of which loops from 0 to 15. Use an if statement to make sure you don't print the character for 127, which is not a printing character.)

A Bit About Memory

It might help you to understand some of the information about C's various variables if we stop and take a quick look at how the memory in your computer is organized and used. Basically, all of the memory in your computer is divided up into a series of bytes, each of which is made up of eight little electronic devices which can store either a one or a zero; these are called bits. The byte is the smallest unit of memory you can address, and the smallest unit of memory C can set aside for any particular use. Each of these bytes can hold 256 different values.

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 C 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, while a char variable only uses one byte. Float numbers and long integers each require four bytes of storage, while a double 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/C can access all of the memory you have, although there are a few tricks you have to know to access more than 64K.

Character Constants and String Constants

Way back in lesson 1, you learned that

```
"Hello, world.\n"
```

is a string constant. A string constant consists of zero 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 single quote marks. As with strings, to get the single quote mark as a character, we use an escape sequence; in this case, \'. All of the following are legal character constants.

```
'a'  ' '  '\''  '6'  '!'  'A'
```

From your standpoint, character constants and string constants are very similar, but you need to keep in mind that the C language does not share your viewpoint. In the C languages, character constants and strings are very, very different from each other. In fact, a character constant is really just another way of writing an integer value: the C compiler doesn't care one bit whether you write the character constant 'A' or the integer constant 65; the program the compiler creates will be exactly the same, either way.

A string constant, on the other hand, is treated just like an array by the compiler. When you use a string constant as a parameter to a function, for example, the compiler actually stores the string in your program, and passes the address of the array to the function – just as it would pass the address of an array to the function if you were to put an array name in the parameter list of a function.

Since character constants and string constants are so different in C, the C language has two different ways of representing these two kinds of constants. A character constant uses single quote marks around the character, while a string constant uses double quote marks. For example, 'A' is a character constant, but "A" is a string constant.

Another Look at strlen

C has a wide variety of functions in the standard library string.h that help you manipulate strings – so many, in fact, that we aren't going to look at all of them in this course. There are a few basic ones, though, that you will want to use again and again, so we'll stop and take a look at them now. For all of these string functions, you should include string.h at the start of your program.

You have already seen one string function, strlen. The strlen function finds out how many characters are in a string by counting the non-null characters that come before the terminating null character. The maximum length that strings can be varies from compiler to compiler, but in ANSI C, the maximum length is at least as big as a pointer, assuming you have that much memory in your computer. In other words, strings can be as big as the memory you have in your computer!

As it happens, that means that the length of a string is actually a long value on the Apple IIGS, so you need to make sure that when you print the result returned by strlen, you use the %ld conversion specification, not the %d conversion specification. When you assign the length of the string to an int value, though, C takes care of converting the value from a long integer to an int value for you. As long as the value is not too big to store in an int variable, you will get the correct answer. You will generally know if you are dealing with huge strings, since you have to allocate arrays big enough to hold them, though, so there is rarely any confusion about which size variable to use.

Copying Strings with strcpy and strncpy

The most basic thing you will want to do with a string is to assign one string to another. Strings are arrays, though, and you can't assign one array to another in C. The strcpy function lets you copy one string to another. Here's a quick example showing how it is used.

```
#include <stdio.h>
#include <string.h>

void main(void)

{
char string1[20], string2[20];

strcpy(string1,
   "Hello, world.\n");
strcpy(string2, string1);
printf(string1);
printf(string2);
}
```

As you can see, you can copy string constants to a string, or one string to another. The order of the strings is important – you put them in the same order you would for an assignment operation: the first string is the one you are setting, and the second string is the one you are copying into the first.

The strcpy function trusts you, so you need to be sure you know what you are doing. strcpy copies characters from the second string into the first until it hits the null character at the end of the second string. After copying the null character, strcpy stops. Counting the terminating null character, the string "Hello, world.\n" is a total of 15 characters long. In our sample program, the string arrays are 20 characters long, so they will hold the value with room to spare, but if you were to change the arrays to 10 character arrays, strcpy would copy right over top of the five bytes that follow the string in memory. In C, it's programmer beware, so you need to be sure your strings are long enough.

While the strcpy function is very useful, the fact that it can write past the end of an array makes it dangerous in some cases. For example, if you are writing a function that works on a string that is passed as a parameter, you may need to guard against the possibility that the string is longer that you are allowing for. You could use strlen to check, but there is also a variation of strcpy called strncpy that can be used. The strncpy function uses one more parameter, a maximum length, to make sure the array doesn't overflow. If the limit is hit, though, your string won't have a terminating null character, so you still have to check to see if there is one – or you can simply tell strncpy to use one fewer than the maximum size of the array, and fill in a null terminator on your own, as this sample program does.

```
#include <stdio.h>
#include <string.h>

void main(void)

{
char string1[10], string2[10];

string1[9] = 0;
string2[9] = 0;
strncpy(string1,
   "Hello, world.\n", 9);
strncpy(string2, string1, 9);
printf(string1);
printf(string2);
}
```

Incidentally, the null character in C is always the character whose integer value is zero; the first two statements in the program are simply making sure there is a null character in the last byte of the string.

Like most C library functions, strcpy and strncpy return something. In this case, they return a copy to the destination string – the one you copied things into.

Problem 5.3. Use the fact that you can print a character as an integer using the %d conversion specification to find out what integer value is used in C to mark the end of a line (the \n escape sequence) and the null character.

Hint: Use strcpy to copy the string "\n" into a char array, then print the first two elements of the array.

Putting Strings Together with strcat and strncat

Putting two strings together is accomplished with a function called strcat, for string concatenation. strcat takes two strings, and tacks the second one onto the first. As with strcpy, there is a close cousin of strcat called strncat that uses one additional parameter to prevent array overflows. Here's a sample of strcat and strncat in action.

```
#include <stdio.h>
#include <string.h>

void main(void)

{
char string[20];

strcpy(string, "Hello, ");
strcat(string, "world");
strncat(string, ".\n", 20);
printf(string);
}
```

Problem 5.4. As the examples have been showing, you can use a string variable as a format string for printf instead of a string constant. Use this fact to build a format string to write a character to the shell window, then change the format specifier and use it again to write the same value as an integer. You can use the value 65 for both printf statements.

While this is a pretty simple example, it points out one of the more powerful aspects of C's format string: you can change it on the fly to adapt a small subroutine to do a wide variety of different situations.

Comparing Strings with strcmp and strncmp

The strcmp function is used to compare two strings. Basically, strcmp compares the characters in each string, starting with the first, until it finds two characters that are not the same. If the character in the first string is less than the character in the second string, strcmp returns a negative int value; if it is greater, strcmp returns a positive int value. If all of the characters up to the terminating null character match exactly, strcmp returns zero. Finally, if one string is shorted than the other, but all of the characters up to the end of the shorter string match, strcmp acts like the shorter string is less than the longer string.

Combining all of this gibberish with the fact that the Apple IIGS uses the ASCII character set, in which letters are ordered right after each other, all of this means that strcmp works just the way you want it to. Assuming all of the characters are uppercase or lowercase, strcmp works the same way you would alphabetize strings. For example, "APPLES" is less than "ORANGES", while "TEDDY" is greater than "TED".

In fact, if you treat the ASCII character set as an extended alphabet, strcmp acts like strings are in alphabetical order for any string. For example, "100" is less than "120", since the character '0' is less than the character '2', and "apples" is greater than "Apples", since a lowercase 'a' character comes after 'A' in the ASCII character set.

Like most of the other string functions, strcmp has a cousin called strncmp. The strncmp function has one addition parameter, which is the maximum number of characters the function will compare.

Problem 5.5. The exact value strcmp will return when one string is less than another is negative in all C compilers, but it can be any negative number. The value returned when the first string is larger than the second is positive, but again, it can be any positive number.

Write a program that prints the actual values returned by both strcmp and strncmp when the first string is less than the second, and when the first string is greater than the second.

Problem 5.6. What will strcmp return for strcmp(s1, s2), when the strings in s1 and s2 are taken from the following table? Assume strcmp uses -1 for a negative value, and 1 for a positive value.

	s1	s2
a.	"APPLES"	"ORANGES"
b.	"50"	"200"
c.	"Mike"	"Michael"
d.	"Apple"	"Apple"
e.	" space"	"space"
f.	"happy"	"Happy"

Passing Strings as Parameters

You've used a lot of library functions that can handle strings; now it's time to find out how to do that for yourself. Listing 5.3 shows a program that implements one of the few functions C doesn't already have: it converts all of the characters in a string from uppercase to lowercase.

The main program should be old hat by now, so we won't worry too much about it. There are several interesting points about the strlower function that we need to look at, though. The first is that I have violated one of my formatting conventions. As I mentioned a few lessons ago, and as you have seen over and over, I use an uppercase letter for the first letter in my function names, but in this program, I violated that convention. You have to keep in mind that formatting conventions

Listing 5.3

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void strlower (char str[1])

/* Convert a string to lowercase                                    */
/*                                                                  */
/* Parameters:                                                      */
/*    str - string to convert                                       */

{
unsigned i;                     /* array index */

i = 0;
while (str[i] != 0) {
   str[i] = tolower(str[i]);
   ++i;
   }
}


void main(void)

/* Main program.                                                    */

{
char string[40];

strcpy(string, "Apples are red, aren't they?\n");
strlower(string);
printf(string);
}
```

are not rules, they are mental aids to help you program faster and make fewer mistakes. Since all of the string library functions start with str, I choose to make this function look as much like them as possible. For me, that will cause me to make fewer mistakes.

Going beyond the formatting, you can see that, to pass an array, you can simply declare the parameter as an array. The index may seem strange, but we'll deal with that issue in a moment. First, let's stop and think about what it means to pass an array. As you already know, C automatically converts an array to a pointer to the first element of the array (or the address of the array, which is another way of saying the same thing) when the array is passed as a parameter. Why, then, do we still declare the parameter as an array? Shouldn't it be declared as char *str, declaring the parameter as a pointer to a character?

Strangely enough, the answer is that it could be. In C, a function parameter that is an array can be treated in the C function as a pointer to an element of the array. If you pass a pointer to a character, you can also treat it as an array of characters in the function. In short, when you are passing a parameter, a pointer to a value and an array of values are completely interchangeable. In fact, later on you will learn how to use pointers to access values in arrays, and arrays to access values pointed to by a pointer. This is a powerful but strange concept that you will see developed gradually in the course.

For the most part, though, if we are passing an array, we declare the parameter as an array, and we use

the parameter as an array in the function. Even though C lets us do some tricks with pointers and arrays, it is usually easier to understand the program if you stick to one or the other. For now, just keep in mind that C knows what to do when you pass an array as a parameter, and declare the parameter as an array, even though the array is converted to a pointer in the mean time. We'll resolve all of the sticky questions about this issue later, after you learn a little more about expressions and pointers.

At the start of this lesson, you saw how dangerous it can be to index past the end of the array, and you saw that C doesn't check to make sure you don't do that in your programs. It may have seemed pretty crude of the language not to make a simple check to prevent such a disastrous error, but now you can see one place when this comes in handy. You see, the strlower function does not know, in advance, how long the string will be – and, since it can index past the end of the array, it doesn't care. We declare the array as having two characters, and then proceed to index into the array until we get to the terminating null character.

It may offend your sensibilities a little to do this. After all, the string we are passing has 40 characters, so why not define the parameter the same way? Go ahead: C doesn't care. But what happens if you call strlower again, in another place, with a string that is 80 characters long? Do you define the parameter with an index of 40, or 80? Sensibilities could be offended either way. In general, you might just pick some number to represent the size of an array parameter and stick with it, since you really can't tell in advance how large the array will be, anyway.

Of course, you might be wondering how much space C is using for this array. Keep in mind, though, that the parameter isn't really an array at all – it is a pointer to an array, so the pointer is the same size no matter how big the array is. The space for the array is allocated somewhere else.

All of this may seem a bit odd, but it gives quite a bit of power to C that doesn't exist in some other languages. The function we just defined can, in fact, be called with strings of several different lengths. You can't do that in Pascal. The trade off is that Pascal has checks to prevent array overflows. You have to be more careful in C, since C doesn't do that kind of checking, but the reward is that you can do some things in C that can't be done in Pascal.

While the standard C libraries don't have a function to convert a string to all uppercase or lowercase, they do have functions to convert characters from lowercase to uppercase, called tolower and toupper. The sample program uses tolower to convert the characters in the array to lowercase characters. Both functions take a character as a parameter, and return a character as a result. As you can see, they do not change characters unless they are, in fact, uppercase characters. C's character manipulation functions are contained in the ctype library; you can see the proper include for ctype.h at the top of the program.

Problem 5.7. Using the sample program in this section as a model, create a program to convert strings to all uppercase characters. You can use the character function toupper to help you do this. Test your function with two different string arrays, one of which is 20 characters long, and the other 30 characters long, to verify that you can, in fact, pass strings of different lengths.

Returning Strings as Function Results

The name of this section is a little misleading, in a way, since like most languages, C does not let you return a string as the result of a function. On the other hand, you can return a pointer to a char value, and as you are starting to discover, that's almost the same thing in C as returning an array. Listing 5.4 shows a reworked version of our last sample program that does just that.

The return statement sure looks like it's returning an array, but then, it looked like we were passing an array as a parameter, too, when in fact we were passing the address of the array. The main function shows why the string functions return a pointer to the array they change, too, instead of just changing the parameter. In this program, we can use printf to print the value returned by the function. Of course, the string has also been changed, as you saw in the last section.

Problem 5.8. The strlower function changes the string you pass it. In some cases, we don't want to do that. Create a function that prints the lowercase equivalent of the string without changing the string you pass to the function.

The program you create will have one serious drawback compared to the programs that change the string. What is the drawback?

The string.h and ctype.h Libraries

The standard C library is large and powerful. In fact, it is so large that we won't get a chance to cover all of the functions in the C library, nor do most C programmers know all of them. It isn't important that you remember all of them, either, but you should have a

Listing 5.4

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

char *strlower (char str[1])

/* Convert a string to lowercase                                    */
/*                                                                  */
/* Parameters:                                                      */
/*    str - string to convert                                       */

{
unsigned i;                     /* array index */

i = 0;
while (str[i] != 0) {
   str[i] = tolower(str[i]);
   ++i;
   }
return str;
}


void main(void)

/* Main program.                                                    */

{
char string[40];

strcpy(string, "Apples are red, aren't they?\n");
printf(strlower(string));
}
```

good idea of what is available, and a general idea of how to find descriptions of the functions when you need to. In the long run, it will save you quite a bit of work.

The tables below give you a quick summary of the functions in string.h and ctype.h, the string and character libraries we have used in this lesson. You know all that you need to to use any of these functions, and you know enough to understand the descriptions of them. Read through the list of functions and pick out a couple that interest you. Make sure you can find them in the ORCA/C reference manual, or whatever C reference manual you happen to prefer. You might even want to try a couple of them in a real program.

One thing you should notice about the table is that there are often several ways of doing the same thing. In all of these situations, the actual functions differ a bit. You have already seen some examples of this: strcpy and strncpy, for example, both copy a string into another string array, but they each have different options for doing so.

This may sound like a make-work exercise, but it really isn't. C is a big language, with a huge library, so very few people ever learn it completely. If you know how your reference manual is laid out, and basically where everything is, there's a good chance that you'll grab it to look up a function like strcpy when you need to. If you are not familiar with your manual, you might avoid it – and your C programs will be larger, less

efficient, and perhaps have more bugs than they would
if you had used the reference manual.

string.h

c2pstr	(ORCA/C specific.) Convert a C string to a p-string.
memchr	Find a byte.
memcmp	Compare memory.
memcpy	Copy memory.
memmove	Copy memory.
memset	Set memory to a value.
p2cstr	(ORCA/C specific.) Convert a p-string to a C string.
strcat	Concatenate strings.
strchr	Find a character.
strcmp	Compare strings.
strcpy	Copy a string.
strcspn	Scan for one of a set of characters.
strerror	Return a run-time error message.
strlen	Find the length of a string.
strncat	Concatenate strings.
strncmp	Compare strings.
strpbrk	Scan for one of a set of characters.
strpos	Find a character.
strrchr	Find a character.
strrpbrk	Scan for one of a set of characters.
strrpos	Find a character.
strspn	Scan for one of a set of characters.
strstr	Find one string in another.
strtok	Break a string into tokens.

ctype.h

isalnum	Is the character alpha-numeric?
isalpha	In the character alphabetic?
isascii	In the character in the ASCII character set?
iscntrl	In the character a control character?
iscsym	In the character one that can appear in a C symbol?
iscsymf	In the character one that can appear as the first character in a C symbol?
isdigit	In the character a digit?
isgraph	In the character a graphic character?
islower	In the character lowercase?
isodigit	In the character an octal digit?
isprint	In the character a printing character?
ispunct	In the character a punctuation character?
isspace	In the character a whitespace character?
isupper	In the character uppercase?
isxdigit	In the character a hexadecimal digit?
toascii	Convert the character to ASCII.
toint	Convert a digit to an integer value.
tolower	Convert a character to lowercase.
toupper	Convert a character to uppercase.
_tolower	Convert a character to lowercase.
_toupper	Convert a character to uppercase.

Lesson Five
Solutions to Problems

Solution to problem 5.1.

```
#include <stdio.h>

void main(void)

{
printf("%c\n", 65);
}
```

Solution to problem 5.2.

```
#include <stdio.h>

void main(void)

{
unsigned row, col;
unsigned ch;

ch = 32;
for (row = 0; row <= 5; ++row) {
   for (col = 0; col <= 15; ++col) {
      if (ch != 127)
         printf("%c", ch);
      ++ch;
      }
   printf("\n");
   }
}
```

Solution to problem 5.3.

```
#include <stdio.h>
#include <string.h>

void main(void)

{
char string[10];

strcpy(string, "\n");

printf("The \\n character is %d.\n", string[0]);
printf("The null character is %d.\n", string[1]);
}
```

Solution to problem 5.4.

```
#include <stdio.h>
#include <string.h>

void main(void)

{
char format[10];

strcpy(format, "%c\n");

printf(format, 65);
format[1] = 'd';
printf(format, 65);
}
```

Solution to problem 5.5.

```
#include <stdio.h>
#include <string.h>

void main(void)

{
printf("strcmp returns %d if the first string is less than the second.\n",
   strcmp("apple", "orange"));
printf("strcmp returns %d if the first string is greater than the second.\n",
   strcmp("orange", "apple"));
printf("strncmp returns %d if the first string is less than the second.\n",
   strncmp("apple", "orange", 10));
printf("strncmp  returns  %d  if  the  first  string  is  greater  than  the
second.\n",
   strncmp("orange", "apple", 10));
}
```

Solution to problem 5.6.

	s1	s2	
a.	"APPLES"	"ORANGES"	-1
b.	"50"	"200"	1
c.	"Mike"	"Michael"	1
d.	"Apple"	"Apple"	0
e.	" space"	"space"	-1
f.	"happy"	"Happy"	1

Solution to problem 5.7.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void strupper (char str[1])

/* Convert a string to uppercase                                    */
/*                                                                  */
/* Parameters:                                                      */
/*    str - string to convert                                       */

{
unsigned i;                     /* array index */

i = 0;
while (str[i] != 0) {
   str[i] = toupper(str[i]);
   ++i;
   }
}


void main(void)

/* Main program.                                                    */

{
char shortString[20], longString[30];

strcpy(shortString, "Short Test.\n");
strupper(shortString);
printf(shortString);
strcpy(longString, "A somewhat longer string.\n");
strupper(longString);
printf(longString);
}
```

Solution to problem 5.8.

The drawback in this program is that there is a maximum length beyond which the program cannot convert a string. In the solution, the longest string that can be handled is 79 characters. You could certainly extend this value, but using this technique, there will always be some maximum string length. The other versions of the subroutines were limited only by the size of the index variable, and even that could have been turned into a long value, so that the subroutines would only be limited by available memory.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

char *strlower (char str[1])

/* Convert a string to lowercase                                    */
/*                                                                  */
/* Parameters:                                                      */
/*    str - string to convert                                       */

{
unsigned i;                     /* array index */
char str2[80];                  /* string to return */

i = 0;
while (str[i] != 0) {
   str2[i] = tolower(str[i]);
   ++i;
   }
str2[i] = 0;
return str2;
}


void main(void)

/* Main program.                                                    */

{
char string[40];

strcpy(string, "Apples are red, aren't they?\n");
printf(strlower(string));
printf(string);
}
```

Lesson Six
More About Arrays

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. This kind of information is often stored in an array, the new way of organizing information you learned in the last lesson. In this section, you will learn how to sort an array so you can do things like put a mailing list in zip code order.

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
0	6
1	43
2	1
3	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
--->	0	6
	1	43
	2	1
	3	6

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

	index	value
	0	6
--->	1	43
	2	1
	3	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
	0	6
--->	1	1
	2	43
	3	6

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

	index	value
	0	6
	1	1
--->	2	6
	3	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
0	1
1	6
2	6
3	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. The normal way to do this in C is to set an int or unsigned value to 0 before you start through the loop, then set it to 1 (or some other non-zero value) if you make a swap. You can then test this value directly in C's do-while loop, since C treats an integer value of 0 as false, and any non-zero integer value as true. That's a pretty powerful concept, and we'll look at it more closely in a moment.

Listing 6.1

```
/* This program reads in an array of up to 100 float numbers.  It  */
/* then sorts the array, and prints the numbers in order.  Numbers */
/* are read until a zero is found.                                 */

#include <stdio.h>

#define MAX 100                 /* max # of floats to sort */

float numbers[MAX];             /* array to sort */
unsigned num;                   /* # of numbers actually read */


void ReadEm (void)

/* Read the list of numbers.                                       */
/*                                                                 */
/* Variables:                                                      */
/*    numbers - array of numbers read                              */
/*    num - number of numbers read                                 */

{
float rval;                     /* number read from the keyboard */

num = 0;
do {
   scanf(" %f", &rval);
   if (rval != 0.0) {
      numbers[num] = rval;
      ++num;
      }
   }
while (rval != 0.0);
}
```

```
void Sort (void)

/* Sort the list of numbers.                                          */
/*                                                                    */
/* Variables:                                                         */
/*    numbers - array of numbers read                                 */
/*    num - number of numbers read                                    */

{
float temp;                    /* temp variable; used for swapping */
unsigned swap;                 /* has a swap occurred? */
unsigned i;                    /* loop variable */

do {                           /* loop until the array is sorted */
   swap = 0;                   /* no swaps, yet */
   for (i = 0; i < num-1; ++i) { /* check each element but the last */
                                /*  if a swap is needed then...    */
      if (numbers[i+1] < numbers[i]) {
         swap = 1;             /* note that there was a swap */
         temp = numbers[i];    /* swap the entries */
         numbers[i] = numbers[i+1];
         numbers[i+1] = temp;
         }
      }
   }
while (swap);
}


void WriteEm (void)

/* Write the list of numbers.                                         */
/*                                                                    */
/* Variables:                                                         */
/*    numbers - array of numbers read                                 */
/*    num - number of numbers read                                    */

{
unsigned i;                    /* loop variable */

for (i = 0; i < num; ++i)
   printf("%f\n", numbers[i]);
}
```

```
void main (void)

/* main program                                                    */

{
ReadEm();                       /* read the list of numbers */
Sort();                         /* sort the numbers */
WriteEm();                      /* write the list of numbers */
}
```

Listing 6.1 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 swap 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 scanf function can read an integer and convert it to a real number.

Problem 6.1. As you learned in the last lesson, the comparison operators can work on characters as well as they can on integers or real numbers. Use this fact to modify the Reverse sample from lesson 5 so it will sort the characters, rather than printing them in reverse order.

Problem 6.2. 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.

Boolean Values

In the sort sample, you got a first look at the way C actually deals with values that are either true or false, something programmers call boolean values. In C, boolean values are handled using integers. When you test to see if a is less than 4 with the statement

```
if (a < 4)
   ...
```

C is actually comparing a to 4, and returning an integer result, just as a+4 would return an integer result. If a is, in fact, less than 4, C returns a non-zero value (1 in most implementations of C, including ORCA/C), and if a is greater than or equal to 4, the result is zero. In C, then, 0 is false, and anything else is true.

While the values are just integers, programmers often refer to expressions that are returning a true or false result as boolean expressions. To make things a little more obvious in the program, you may also see definitions like

```
#define BOOLEAN int
#define FALSE 0
#define TRUE 1
```

in a C program. This just gives you a slightly more mnemonic way of dealing with integers that are being used as boolean values; it's a mental crutch, like comments, that makes the program a little easier to read and understand.

Now we can put some of these ideas together to rapidly expand what we can do with the C language. For example,

```
numbers[i+1] < numbers[i]
```

is a boolean expression, returning an integer value, so we can assign it to an integer variable. If less is an int variable, then the assignment

```
less = numbers[i+1] < numbers[i];
```

works perfectly well in C. Also, as you saw briefly in the sort sample, you can use this int variable, or any other integer, for that matter, as the condition in a do-while loop, while loop, or if statement.

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. (Since C handles boolean values as integers, though, you can actually do this in C, but good programmers tend not to do this sort of thing.) 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 three operators that work on integers in a way that is consistent with the way we think about true and false. These operators are || (logical or), && (logical and) and ! (logical negation, often thought of as not). Note that the or operator is two | characters, not one, and the and operator is two & characters, not one. It is very important that you remember to use || and &&, since C also has & and | operators that do something else. The single-character operators, & and |, are called bitwise operators. You will learn about these later.

Like the familiar math operators +, -, * and /, the && and || operators work on two values. These values can actually be just about any type you have learned so far – any type, in fact, except for an array or a function. To be more specific, the types can be any of the signed or unsigned integers, float, double, char, or even a pointer. Each of these values is treated as false if it is 0 (0.0 for float and double, and NULL for pointers), and true if it is not zero. The && operator (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 Porsche 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 Porsche. The || operator (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 an election *or* if George Bush dies in office. Both the && and || operators return an int value that is 0 if the result is false, and non-zero if the result is true. Expressing these concepts in C, we get:

```
getAPorsche = rich && wifeApproves;
presidentQuale =
   QualeElected || BushDies;
```

The ! operator (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, and again, it also works on any of the types that the && or || operators support. I'm sure you have heard someone say, "that is not true!" Well, in C, as in English, not true is the same as false. For example, !a will be 0 if a started out as non-zero, and it will be some non-zero value if a started out as zero. The not operator is very handy in conditions.

```
if (!rich)
   printf("Keep the Datsun.\n");
```

As with math operators, boolean operators use operator precedence to determine the order they are applied to the arguments. The ! operator has the highest precedence, and is applied first. Next comes &&, followed by ||. The precedence of the operators, and their relationship with the other operators, can be found in the ORCA/C reference manual, or any other good C reference manual.

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 || beauty && 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 !, there are six other hybrid operators you are already familiar with that take numeric arguments and return a boolean result. The result of these operators can be used in an expression anywhere an int variable can be used, and, like the other boolean operators, the result is a zero for false, and some non-zero value for true. The operators, of course, are the six comparison operators:

```
<    >    <=   >=   !=   ==
```

I mentioned that ORCA/C, like most C compilers, will return 1 for the result of a boolean operator if the result is true, but you may have noticed that I was very careful to say "0 for false, and non-zero for true" in all of the places where I described the result of a boolean operator. This is one of those places that separates the good C programmers from the not-so-good programmers, and one of the reasons why C programs are only portable if you work at it a bit. While ANSI C requires a value of 1 for true, not all compilers are ANSI compilers. Historically, the value -1 has also been used a lot for true. In good programs, you should handle boolean values the same way C does: 0 is false, and anything else is true.

Problem 6.3. Back in lesson 3, we used two do-while loops to choose a random number, like this:

```
do
   do
      value = rand();
   while (value <= 0);
while (value > 100);
```

While this was a good way to learn a little about nested do-while loops, it is terrible C.

Use what you now know about boolean expressions to combine the two do-while loops into a single do-while loop. With this accomplished, write a program that selects and prints ten random integers in the range 1 to 100.

Problem 6.4. The printf library function doesn't have a way to write a boolean value as true or false. Write a function called PrintBool that will print the string false if you pass it a zero, and true if you pass anything but a zero. Test your function by calling it with the values 0, 1, and -1.

Arrays of Arrays

So far, we have looked at arrays of integers, arrays of reals, and arrays of characters, but you can actually define an array of any type in C, 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 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. C 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. For example, a string that can hold up to 10 characters is defined in C as

```
char str[11];
```

To define an array of 20 strings, each of which is able to hold 10 characters, we just add another array subscript before the first:

```
char strings[20][11];
```

You can access either the strings or the individual characters, depending on how many subscripts you specify. For example, to compare the third string in the array to "Mike", you could use

```
strcmp(strings[2], "Mike");
```

To print the fourth character of the second string, you would use

```
printf("%c", strings[1][3]);
```

These ideas are put to use in the program shown in listing 6.2. With these simple additions to your knowledge of arrays, there isn't anything really new about the program, but you should certainly take the time to use the debugger for a closer look at it. The first problem in this section will give you some things to try.

Listing 6.2

```
/* 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 consist of whitespace separated words.  Once you have   */
/* typed all of the words you want to sort, enter the string "#".  */

#include <stdio.h>
#include <string.h>

#define MAX 100                 /* max # of strings to sort */
#define SIZE 100                /* max size of a string */
```

```
char strings[MAX][SIZE+1];      /* strings to sort */
unsigned num;                   /* # of strings actually read */


void ReadEm (void)

/* Read the list of strings.                                    */
/*                                                              */
/* Variables:                                                   */
/*    strings - array of strings read                           */
/*    num - number of strings read                              */

{
char sval[SIZE+1];              /* string read from keyboard */

num = 0;
do {
   scanf("%s", sval);
   if (strcmp(sval, "#")) {
      strcpy(strings[num], sval);
      ++num;
      }
   }
while (strcmp(sval, "#"));
}


void Sort (void)

/* Sort the strings                                             */
/*                                                              */
/* Variables:                                                   */
/*    strings - array of strings to sort                        */
/*    num - number of strings in the array                      */

{
char temp[SIZE+1];              /* temp variable; used for swapping */
unsigned swap;                  /* has a swap occurred? */
unsigned i;                     /* loop variable */

do {                            /* loop until the array is sorted */
   swap = 0;                    /* no swaps, yet */
   for (i = 0; i < num-1; ++i) { /* check each element but the last */
                                 /*  if a swap is needed then...    */
      if (strcmp(strings[i+1], strings[i]) < 0) {
         swap = 1;              /* note that there was a swap */
         strcpy(temp, strings[i]); /* swap the entries */
         strcpy(strings[i], strings[i+1]);
         strcpy(strings[i+1], temp);
         }
```

```
      }
   }
while (swap);
}


void WriteEm (void)

/* Write the list of strings.                                    */
/*                                                               */
/* Variables:                                                    */
/*    strings - array of strings read                            */
/*    num - number of strings read                               */

{
unsigned i;                    /* loop variable */

for (i = 0; i < num; ++i)
   printf("%s\n", strings[i]);
}


void main (void)

/* Main program                                                  */

{
ReadEm();                      /* read the list of strings */
Sort();                        /* sort the strings */
WriteEm();                     /* write the list of strings */
}
```

Problem 6.5. 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.

```
Ran
Mike
123
Run
microphone
1212
```

Problem 6.6. In this problem, you will develop a program to play the game Hangman. This is a hard problem, easily the hardest so far in this course, and you should expect to take quite a bit of time on it. You will need to spend some time planning the program and thinking about what it will do before you start writing it. In addition, plan on reading this problem at least twice: once to get an idea about what is going on, and once more to fill in the details.

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 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 player 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. Use a #define to declare a constant called MAXWORDS to indicate how many words are in the array. Another global constant 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 use it to initialize the random number generator by calling the srand() function. Add the RandomValue subroutine from lesson 4 to your program. Test your work so far by calling RandomValue with a maximum value of MAXWORDS, and printing the corresponding word in the word list.

3. Add a new subroutine that plays hangman. You can try this on your own, or follow this basic outline:

    a. Declare an array of integers called found that is MAXCHARS long. You will use this array to keep track of which characters have actually been found by the player. Start by initializing each element of the array to false (0). The proper use of this array is central to getting your program to work, so we will take a closer look at it.

    To understand how the found array is used, let's play a short game of hangman, using the word "oops". The word will be in an array of strings called words, and we will assume that the particular word we want has an index of w. We'll also assume that we have already called strlen to find out how long the word is, and saved the result: in our example, though, we'll gloss over this by just dealing with the first four elements of word[w] and found. Once found is initialized, then, our arrays look like this:

```
found[0] = 0   word[w][0] = 'o'
found[1] = 0   word[w][1] = 'o'
found[2] = 0   word[w][2] = 'p'
found[3] = 0   word[w][3] = 's'
```

    Let's say the player guesses 'p'. We would scan the array, setting found[2] to 1, indicating that the character has been found. If the player guesses a letter that is not in the array, like 'e', we do nothing. If the player guesses a letter like 'o' that is in the array more than one time, we set all of the corresponding elements of found to the proper value; in this case, found[0] and found[1] would be set to 1.

    b. Declare a variable called wrong, and initialize it to zero. This variable is used to keep track of how many wrong guesses the player has made.

    c. Declare a variable called done, which you initialize to 0 (false). This variable controls a do-while loop that you should create; this is where the action takes place. The do-while loop should loop until the done variable is set to true.

    d. Choose a word, using the same ideas developed in the last step.

    e. At the start of the do-while loop, print the word. You will loop over the array containing the word, printing the letter in the word if the corresponding element of

the found array is true, and printing a '-' character if it is false. The effect is to print one character for each character in the word the player is trying to guess. If the player has already correctly guessed the letter, you print it; if not, you print a '-' as a place holder.

f. Get a character from the player. Scan the string to see if the character is in the word the player is trying to guess. If so, you need to set the proper elements of the found array to indicate that the character has been found. If not, you need to increment the variable wrong.

g. Scan your found array to see if there are any more characters to find. If not, the game is over: print the word, along with a congratulatory statement, and set done to true so the program will drop out of the loop.

h. If you incremented wrong, you need to print some sort of message. Do this with a series of if statements: if wrong is one, print something like "<ch> is not in the word: your head is in the noose!" and move on. If wrong is 6, the game is over. Print something like "Sorry, Jack Ketch got you!" followed by the correct word, then set done to true so you drop out of the loop.

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 function called PlayAgain that asks the player if he wants to play another game, and returns true (non-zero) or false (zero), 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 C. 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 three sections.

In all of the samples and problems in the rest of this lesson, 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, C 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 C 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 C, and back again. To convert from degrees to radians, we can multiply by π, and divide by 180. The opposite operation converts back. The following two subroutines package these ideas neatly; we can use them in our programs to do all of our conversions.

```
float DtoR(float degrees)

/* Convert from degrees to */
/* radians.                */

{
return 0.01745329 * degrees;
}
```

```
float RtoD(float radians)

/* Convert from radians to */
/* degrees.                */

{
return radians * 57.295780;
}
```

The basic trigonometry functions are sine, cosine, tangent, and the inverses of these functions, arcsine, arccosine, and arctangent. C gives you these functions, calling them sin, cos, tan, asin, acos and atan. There is even a special, two-argument form of the arctangent function that can handle the entire unit circle, rather than the 180 degrees you are limited to by a single argument. All of the functions are defined in the math.h header file, which has quite a few other trigonometric, logarithmic, and general use functions besides these basic ones.

The sin, cos and tan functions accept an angle, expressed in radians, as the argument. They return the sine, cosine or tangent of the argument. The following program makes use of the tan function 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 this program!

```
/* Find the height of a    */
/* building.               */

#include <stdio.h>
#include <math.h>

float DtoR(float degrees)

/* Convert from degrees to */
/* radians.                */

{
return 0.01745329 * degrees;
}

void main(void)

/* main program */

{
float distance; /* distance to */
                /*building     */
float angle;  /* angle between */
              /* base & top    */


printf("This program finds "
   "the height of\n"
   "a building.  You must supply"
   " the\n"
   "distance to the building and"
   "the\n"
   "angle between the base and"
   " top of\n"
   "the building (in degrees)."
   "\n\ndistance=");
scanf(" %f", &distance);
printf("angle=");
scanf(" %f", &angle);
printf("height = %f\n",
   tan(DtoR(angle))*distance);
}
```

There is one new programming trick shown here: in C, you can actually create a big string by coding a number of short strings one right after the other. The first printf statement shows this. You can use this idea as I have, to fit long strings into a limited amount of space, or you can use the idea to break a number of lines up on separate lines of your program while still making your program efficient. After all, it takes time to call the printf function, and by creating this long string, we only have to call printf once, instead of one time for each line.

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 C reference manual for details.

Converting Types

As you start to use float values more and more, you will occasionally come across situations where you need to convert from one type to another. The next

section presents one of these situations, so we are going to take a moment to look at type conversions now. For example, you may have a float value, and want to convert it to an integer. For the most part, C does this sort of thing for you automatically. For example, if f is a float variable, i is an integer variable, and l is a long variable, then all of these assignments are legal:

```
i = f;
f = l;
l = i;
i = l;
```

What C does in each of these cases is about what you would expect and hope. When you assign an integer value to another integer that is a different size, C quietly converts the integer from one type to another. If you assign a long integer to an integer, and the long integer is too big to fit in an integer, C can't do the conversion, but it does still do the assignment, lopping off part of the larger number. This can be a bit of a problem, but like all aspects of C, it is just something you have to be aware of, and check for if there is a possibility of a mistake.

When you assign an integer – long or normal sized – to a float value, C does an exact conversion, assuming that is possible. There are a few cases, of course, where it is not possible. For example, if you convert a long value of 123456789 to a float value, you will loose the last two digits, since float numbers are only accurate to seven significant figures.

When C converts a float value to a long or int value, the float value is rounded to the integer closest to the float value. Assigning 1.2 to an int would set the int to 1, while assigning 1.7 to an int would set it to 2. Again, of the number is too large to fit in the variable, the result is not valid.

In most situations, you can just let C take care of these details for you. There are a few programming situations, though, when you want to force the conversion for yourself. In these situations, you can use a type cast. A type cast is coded as a type, enclosed in parenthesis, appearing right before a value, like this:

```
(type) value
```

Type casts convert values using the same rules that C uses when you assign one type of value to another type of variable.

For example, to convert a float value to an integer, so you can print it using the %d conversion specification, you would write this:

```
printf("%d\n", (int) f);
```

There are three relatively common reasons for using a type cast to convert between number types, as well as a few other reasons for type casts of other types you will run across later. The first situation is to prevent a number overflow. As an example, let's assume you are going to multiply two int values, and store the results in a long value. If you write the program like this:

```
l = i*j;
```

the program will work fine if the result is small enough to fit in an integer, but C is doing an int multiply, so if i contained 10000, and j contained 10, the answer would not be correct. Casting one of the integers to a long, like this:

```
l = i * (long) j;
```

gives an entirely different result. This time, j is converted to a long value, and since one of the values is long, C uses a long multiply.

You might remember a problem from lesson 3, where you compared the speed of float math with the speed of int math, and found out how much faster integer calculations can be performed. This is the second reason to use type casting: to force the conversion from a float type to an int type quickly, so that the resulting calculations are also quicker. For example,

```
i = ((int) f) / 10;
```

is faster than letting C do the calculation with no guidance. In this statement, the float value f is converted to an integer right away, then an integer divide is used. Without the type cast, you would get the same answer, but it would take longer, since a floating-point divide would be performed, and the result converted to an integer. This isn't always the correct thing to do, of course, since not all calculations give the same answer when you use integer calculations, but the trick is worth keeping in mind for the situations where it does work. If you need some extra convincing, refer back to lesson 3 and check out the execution times again.

Finally, we come to the situation you will face in the next section. The toolbox .h files define the tool calls using K&R parameters, as discussed at the end of lesson 4. As a result, you must make sure that you pass exactly the type of parameters the toolbox calls expect. If you calculate a coordinate for the graphics screen using float variables, for example, these values must be

converted to int values before being used as parameters to the familiar graphics calls MoveTo and LineTo. If x and y are float variables, this would be the correct way to plot a point at x, y:

```
MoveTo ((int)x, (int)y);
LineTo ((int)x, (int)y);
```

Rotation

Over the past three lessons, you have learned a data type – character – 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 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 to 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.

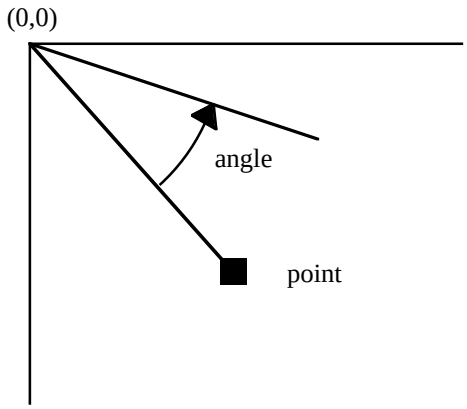


Figure 6.1: How a Point is Rotated

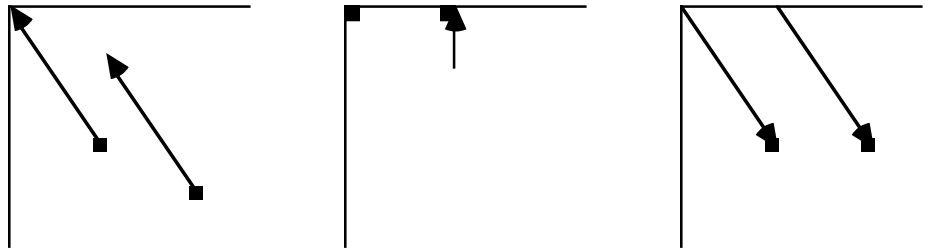
Once we know the coordinates of the point and the angle, we can figure out where the new point is using these formulas, shown here in C:

```
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 C will not do for us automatically.

These ideas are (finally!) put to use in the program in listing 6.3, which spins a square in the graphics window.



Step 1: Move the Points Step 2: Rotate Step 3: Move Them Back

Figure 6.2: Rotating About An Arbitrary Point

Listing 6.3

```
/* Rotate a square 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.                        */

#include <quickdraw.h>
#include <math.h>

#define XSCALE 86               /* x conversion factor */
#define YSCALE 33               /* y conversion factor */
#define pi 3.1415927            /* circumference of a circle */


void InitGraphics (void)

/* Standard graphics initialization           */

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

```
void Rotate (float *x, float *y, float angle, float ox, float oy)

/* 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          */

{
float cosAngle,sinAngle;        /* sin and cos of angle */
float nx;                       /* new x */

*x -= ox;                       /* move the point */
*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;
}


void RotateSquare (float x[4], float y[4])

/* Rotate the square 9 degrees                */
/*                                            */
/* Parameters:                                */
/*    x,y - coordinates of square             */

{
unsigned i;                     /* loop variable */

for (i = 0; i < 4; ++i)
   Rotate(&x[i], &y[i], pi/20.0, 1.5, 1.5);
}
```

```
void DrawSquare (int color, float x[4], float y[4])

/* Draw the square                              */
/*                                              */
/* Parameters:                                  */
/*    color - color to draw                     */
/*    x,y - coordinates of the square           */

{
SetSolidPenPat(color);       /* set the pen color */
                             /* draw the square */
MoveTo((int) (x[0]*XSCALE), (int) (y[0]*YSCALE));
LineTo((int) (x[1]*XSCALE), (int) (y[1]*YSCALE));
LineTo((int) (x[2]*XSCALE), (int) (y[2]*YSCALE));
LineTo((int) (x[3]*XSCALE), (int) (y[3]*YSCALE));
LineTo((int) (x[0]*XSCALE), (int) (y[0]*YSCALE));
}


void main(void)

/* Main program                                 */

{
float x[4],y[4],oldX[4],oldY[4]; /* points in the square */
unsigned i,j;                    /* loop variables */

InitGraphics();                  /* set up the graphics window */
x[0] = 1.0;   y[0] = 1.0;        /* initialize the square */
x[1] = 2.0;   y[1] = 1.0;
x[2] = 2.0;   y[2] = 2.0;
x[3] = 1.0;   y[3] = 2.0;
DrawSquare(0, x, y);             /* draw the square */

for (i = 0; i < 10; ++i) {
   for (j = 0; j < 4; ++j) {     /* save the current location */
      oldX[j] = x[j];
      oldY[j] = y[j];
      }
   RotateSquare(x, y);           /* rotate */
   DrawSquare(3, oldX, oldY);    /* erase the old square */
   DrawSquare(0, x, y);          /* draw the square */
   }
}
```

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 C. 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 packaged the idea of drawing and rotating a square into subroutines. We have also reused our InitGraphics function to get 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.

```
x[0] = 1.0;   y[0] = 1.0;
x[1] = 2.0;   y[1] = 1.0;
x[2] = 2.0;   y[2] = 2.0;
x[3] = 1.0;   y[3] = 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.

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 function that draws a circle using this method. The function 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; i <= 40; ++i) {
   Rotate(x,y,pi/20.0,1.5,1.5);
   lineto(x,y);
   }
```

Use this function 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 function from the last problem to draw a star, rather than a circle. Change the function to draw a five pointed star, with the points of the star 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.

```
/* This program reads in a string, sorts the characters, and       */
/* writes the string back to the shell window.                     */

#include <stdio.h>
#include <string.h>

#define MAX 255                /* max length of a string */

char inString[MAX];            /* input string */
char outString[MAX];           /* output string */


void Sort (void)

/* Sort the characters in a string                                 */
/*                                                                 */
/* Variables:                                                      */
/*    inString - string to sort                                    */
/*    outString - sorted string                                    */

{
unsigned swap;                 /* has a swap occurred? */
unsigned i;                    /* loop variable */
unsigned index;                /* length of the reversed string */
char temp;                     /* used to swap entries */

strcpy(outString,inString);    /* make a copy of the string */
index = strlen(outString);     /* get the length */
do {                           /* loop until the array is sorted */
   swap = 0;                   /* no swaps, yet */
   for (i = 0; i < index-1; ++i) { /* check each element but the last */
                                 /*  if a swap is needed then...    */
      if (outString[i+1] < outString[i]) {
         swap = 1;             /* note that there was a swap */
         temp = outString[i];  /* swap the entries */
         outString[i] = outString[i+1];
         outString[i+1] = temp;
         }
      }
   }
while (swap);
}
```

```
void main (void)

/* main program                                                    */

{
printf("String :");
scanf("%s", inString);
Sort();
printf("Sorted: %s\n", outString);
}
```

Solution to problem 6.2.

```
/* This program reads in an array of up to 100 float numbers.  It  */
/* then sorts the array, and prints the numbers in reverse order.  */
/* Numbers are read until a zero is found.                         */

#include <stdio.h>

#define MAX 100                 /* max # of floats to sort */

float numbers[MAX];             /* array to sort */
unsigned num;                   /* # of numbers actually read */


void ReadEm (void)

/* Read the list of numbers.                                       */
/*                                                                 */
/* Variables:                                                      */
/*    numbers - array of numbers read                              */
/*    num - number of numbers read                                 */

{
float rval;                     /* number read from the keyboard */

num = 0;
do {
   scanf(" %f", &rval);
   if (rval != 0.0) {
      numbers[num] = rval;
      ++num;
      }
   }
while (rval != 0.0);
}
```

```
void Sort (void)

/* Sort the list of numbers.                                          */
/*                                                                    */
/* Variables:                                                         */
/*    numbers - array of numbers read                                 */
/*    num - number of numbers read                                    */

{
float temp;                    /* temp variable; used for swapping */
unsigned swap;                 /* has a swap occurred? */
unsigned i;                    /* loop variable */

do {                           /* loop until the array is sorted */
   swap = 0;                   /* no swaps, yet */
   for (i = 0; i < num-1; ++i) { /* check each element but the last */
                                /*  if a swap is needed then...    */
      if (numbers[i+1] > numbers[i]) {
         swap = 1;             /* note that there was a swap */
         temp = numbers[i];    /* swap the entries */
         numbers[i] = numbers[i+1];
         numbers[i+1] = temp;
         }
      }
   }
while (swap);
}


void WriteEm (void)

/* Write the list of numbers.                                         */
/*                                                                    */
/* Variables:                                                         */
/*    numbers - array of numbers read                                 */
/*    num - number of numbers read                                    */

{
unsigned i;                    /* loop variable */

for (i = 0; i < num; ++i)
   printf("%f\n", numbers[i]);
}
```

```
void main (void)

/* main program                                                    */

{
ReadEm();                       /* read the list of numbers */
Sort();                         /* sort the numbers */
WriteEm();                      /* write the list of numbers */
}
```

Solution to problem 6.3.

```
#include <stdio.h>
#include <stdlib.h>

#define NUM 10                  /* # of integers to print */


unsigned RandomValue (unsigned max)

/* Return a pseudo-random number in the range 1..max.              */
/*                                                                 */
/* Parameters:                                                     */
/*    max - largest number to return                               */

{
int value;

do
   value = rand();
while ((value <= 0) || (value > max));
return value;
}


void main(void)

/* Main program.                                                   */

{
unsigned i;


srand(1234);                  /* initialize the random number generator */
for (i = 0; i < NUM; ++i)       /* print NUM random integers */
   printf("%d\n", RandomValue(100));
}
```

Solution to problem 6.4.

```
/* Test a function that prints boolean values */

#include <stdio.h>

void PrintBool (int bool)

/* Print a boolean string                                          */
/*                                                                 */
/* Parameters:                                                     */
/*    bool - boolean value                                         */

{
if (bool)
   printf("true");
else
   printf("false");
}


void main(void)

/* Main program.                                                   */

{
PrintBool(-1); printf("\n");
PrintBool( 0); printf("\n");
PrintBool( 1); printf("\n");
}
```

Solution to problem 6.5.

The list will be sorted to this order:

```
1212
123
Mike
Ran
Run
microphone
```

Solution to problem 6.6.

Part 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.                                      */

#include <stdio.h>
#include <string.h>

#define MAXWORDS 10             /* possible words */
#define MAXCHARS 8              /* number of characters in each word */

char words[MAXWORDS][MAXCHARS+1]; /* word array */


void FillArray (void)

/* Fill the word array.                                 */
/*                                                      */
/* Variables:                                           */
/*    words - word array                                */

{
strcpy(words[0], "computer");
strcpy(words[1], "whale");
strcpy(words[2], "megabyte");
strcpy(words[3], "modem");
strcpy(words[4], "chip");
strcpy(words[5], "online");
strcpy(words[6], "disk");
strcpy(words[7], "monitor");
strcpy(words[8], "window");
strcpy(words[9], "keyboard");
}
```

```
void PrintArray (void)

/* Print the word array                                */
/*                                                     */
/* Variables:                                          */
/*    words - word array                               */

{
unsigned i;                     /* loop variable */

for (i = 0; i < MAXWORDS; ++i)
   printf("%s\n", words[i]);
}


void main (void)

/* Main program                                        */

{
FillArray();                    /* fill the word array */
PrintArray();                   /* print the word array */
}
```

Part 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.                                  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAXWORDS 10             /* possible words */
#define MAXCHARS 8              /* number of characters in each word */

char words[MAXWORDS][MAXCHARS+1]; /* word array */
```

```
void FillArray (void)

/* Fill the word array.                                    */
/*                                                         */
/* Variables:                                              */
/*    words - word array                                   */

{
strcpy(words[0], "computer");
strcpy(words[1], "whale");
strcpy(words[2], "megabyte");
strcpy(words[3], "modem");
strcpy(words[4], "chip");
strcpy(words[5], "online");
strcpy(words[6], "disk");
strcpy(words[7], "monitor");
strcpy(words[8], "window");
strcpy(words[9], "keyboard");
}


void GetSeed (void)

/* Initialize the random number generator             */

{
int val;                        /* seed value */

printf("Please enter a random number seed:");
scanf(" %d", &val);
srand(val);
}


unsigned RandomValue (unsigned max)

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

{
return rand() % max + 1;
}
```

```
void main (void)

/* Main program                                        */

{
FillArray();                    /* fill the word array */
GetSeed();                    /* initialize the random number generator */

printf("%s\n", words[RandomValue(MAXWORDS)-1]); /* write a random word */
}
```

Part 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.                                  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAXWORDS 10             /* possible words */
#define MAXCHARS 8              /* number of characters in each word */

char words[MAXWORDS][MAXCHARS+1]; /* word array */
```

```
void FillArray (void)

/* Fill the word array.                                    */
/*                                                         */
/* Variables:                                              */
/*    words - word array                                   */

{
strcpy(words[0], "computer");
strcpy(words[1], "whale");
strcpy(words[2], "megabyte");
strcpy(words[3], "modem");
strcpy(words[4], "chip");
strcpy(words[5], "online");
strcpy(words[6], "disk");
strcpy(words[7], "monitor");
strcpy(words[8], "window");
strcpy(words[9], "keyboard");
}


void GetSeed (void)

/* Initialize the random number generator             */

{
int val;                        /* seed value */

printf("Please enter a random number seed:");
scanf(" %d", &val);
srand(val);
}


unsigned RandomValue (unsigned max)

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

{
return rand() % max + 1;
}
```

```
void Play (void)

/* Play a game of hangman.                              */
/*                                                      */
/* Variables:                                           */
/*    words - word array                                */

{
int allFound;                   /* used to test for unknown chars */
char ch;                        /* character from player */
int done;                       /* is the game over? */
char found[MAXCHARS];           /* characters found by player */
unsigned len;                   /* length of word; for efficiency */
unsigned i;                     /* loop variable */
int inString;                   /* is ch in the string? */
char word[MAXCHARS+1];          /* word to guess */
int wrong;                      /* number of wrong guesses */

for (i = 0; i < MAXCHARS; ++i)  /* no letters guessed, so far */
   found[i] = 0;
wrong = 0;                      /* no wrong guesses, yet */
done = 0;                       /* the game is not over, yet */
                                /* pick a word */
strcpy(word, words[RandomValue(MAXWORDS)]);
len = strlen(word);             /* record the length of the word */
do {
   printf("\nThe word is: \""); /* write the word */
   for (i = 0; i < len; ++i)
      if (found[i])
         printf("%c", word[i]);
      else
         printf("-");
   printf("\"\nGuess a character:"); /* get the player's choice */
   scanf(" %c", &ch);
   inString = 0;                /* see if ch is in the string */
   for (i = 0; i < len; ++i)
      if (word[i] == ch) {
         found[i] = 1;
         inString = 1;
         }
   if (inString)                /* handle a correct guess */
      printf("%c is in the string.\n", ch);

   else {                       /* handle an incorrect guess */
      printf("%c is not in the string.\n", ch);
      ++wrong;                  /* one more wrong answer... */
      printf("Your ");          /* tell the player how they are doing */
      if (wrong == 1)
         printf("head");
      else if (wrong == 2)
```

```
         printf("body");
      else if (wrong == 3)
         printf("left arm");
      else if (wrong == 4)
         printf("right arm");
      else if (wrong == 5)
         printf("left leg");
      else /* if (wrong == 6) */
         printf("right leg");
      printf(" is now in the noose!\n");
      }

   if (wrong == 6) {            /* see if the player is hung */
      printf("\n\nSorry, Jack Ketch got you!\nThe word was \"%s\".\n",
         word);
      done = 1;
      }
   allFound = 1;                /* check for unknown characters */
   for (i = 0; i < len; ++i)
      if (!found[i])
         allFound = 0;
   if (allFound) {              /* see if the player got the word */
      printf("You got it!  The word is \"%s\".\n", word);
      done = 1;
      }
   }
while (!done);
}


void main (void)

/* Main program                                          */

{
FillArray();                   /* fill the word array */
GetSeed();                   /* initialize the random number generator */
Play();                        /* play a game */
}
```

Part 4

```
/* 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.                                      */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAXWORDS 10             /* possible words */
#define MAXCHARS 8              /* number of characters in each word */

char words[MAXWORDS][MAXCHARS+1]; /* word array */


void FillArray (void)

/* Fill the word array.                                     */
/*                                                          */
/* Variables:                                               */
/*    words - word array                                    */

{
strcpy(words[0], "computer");
strcpy(words[1], "whale");
strcpy(words[2], "megabyte");
strcpy(words[3], "modem");
strcpy(words[4], "chip");
strcpy(words[5], "online");
strcpy(words[6], "disk");
strcpy(words[7], "monitor");
strcpy(words[8], "window");
strcpy(words[9], "keyboard");
}
```

```
void GetSeed (void)

/* Initialize the random number generator             */

{
int val;                        /* seed value */

printf("Please enter a random number seed:");
scanf(" %d", &val);
srand(val);
}


unsigned RandomValue (unsigned max)

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

{
return rand() % max + 1;
}


void Play (void)

/* Play a game of hangman.                            */
/*                                                    */
/* Variables:                                         */
/*    words - word array                              */

{
int allFound;                   /* used to test for unknown chars */
char ch;                        /* character from player */
int done;                       /* is the game over? */
char found[MAXCHARS];           /* characters found by player */
unsigned len;                   /* length of word; for efficiency */
unsigned i;                     /* loop variable */
int inString;                   /* is ch in the string? */
char word[MAXCHARS+1];          /* word to guess */
int wrong;                      /* number of wrong guesses */
```

```
for (i = 0; i < MAXCHARS; ++i)  /* no letters guessed, so far */
   found[i] = 0;
wrong = 0;                      /* no wrong guesses, yet */
done = 0;                       /* the game is not over, yet */
                                /* pick a word */
strcpy(word, words[RandomValue(MAXWORDS)]);
len = strlen(word);             /* record the length of the word */
do {
   printf("\nThe word is: \""); /* write the word */
   for (i = 0; i < len; ++i)
      if (found[i])
         printf("%c", word[i]);
      else
         printf("-");
   printf("\"\nGuess a character:"); /* get the player's choice */
   scanf(" %c", &ch);
   inString = 0;                /* see if ch is in the string */
   for (i = 0; i < len; ++i)
      if (word[i] == ch) {
         found[i] = 1;
         inString = 1;
         }
   if (inString)                /* handle a correct guess */
      printf("%c is in the string.\n", ch);

   else {                       /* handle an incorrect guess */
      printf("%c is not in the string.\n", ch);
      ++wrong;                  /* one more wrong answer... */
      printf("Your ");          /* tell the player how they are doing */
      if (wrong == 1)
         printf("head");
      else if (wrong == 2)
         printf("body");
      else if (wrong == 3)
         printf("left arm");
      else if (wrong == 4)
         printf("right arm");
      else if (wrong == 5)
         printf("left leg");
      else /* if (wrong == 6) */
         printf("right leg");
      printf(" is now in the noose!\n");
      }
```

```
   if (wrong == 6) {            /* see if the player is hung */
      printf("\n\nSorry, Jack Ketch got you!\nThe word was \"%s\".\n",
         word);
      done = 1;
      }
   allFound = 1;                /* check for unknown characters */
   for (i = 0; i < len; ++i)
      if (!found[i])
         allFound = 0;
   if (allFound) {              /* see if the player got the word */
      printf("You got it!  The word is \"%s\".\n", word);
      done = 1;
      }
   }
while (!done);
}


int PlayAgain (void)

/* See if the player wants to play another game.      */
/*                                                    */
/* Returns:                                           */
/*    True to play again, false to quit.              */

{
char ch;                        /* player's response */

printf("\n\n");
do {
   printf("Would you like to play again (y or n)?");
   scanf(" %c", &ch);
   }
while ((ch != 'y') && (ch != 'Y') && (ch != 'n') && (ch != 'N'));
return (ch == 'y') || (ch == 'Y');
}


void main (void)

/* Main program                                       */

{
FillArray();                    /* fill the word array */
GetSeed();                    /* initialize the random number generator */
do
   Play();                      /* play a game */
while (PlayAgain());            /* loop if he wants to play again */
}
```

Solution to problem 6.7.

```
/* Use rotation to draw a "flower"              */

#include <quickdraw.h>
#include <math.h>

#define XSCALE 86               /* x conversion factor */
#define YSCALE 33               /* y conversion factor */
#define pi 3.1415927            /* circumference of a circle */


void InitGraphics (void)

/* Standard graphics initialization            */

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


void Rotate (float *x, float *y, float angle, float ox, float oy)

/* 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          */

{
float cosAngle,sinAngle;        /* sin and cos of angle */
float nx;                       /* new x */

*x -= ox;                       /* move the point */
*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;
}
```

```
void RotateSquare (float x[4], float y[4])

/* Rotate the square 9 degrees                */
/*                                            */
/* Parameters:                                */
/*    x,y - coordinates of square             */

{
unsigned i;                     /* loop variable */

for (i = 0; i < 4; ++i)
   Rotate(&x[i], &y[i], pi/20.0, 1.25, 1.25);
}


void DrawSquare (int color, float x[4], float y[4])

/* Draw the square                            */
/*                                            */
/* Parameters:                                */
/*    color - color to draw                   */
/*    x,y - coordinates of the square         */

{
SetSolidPenPat(color);       /* set the pen color */
                             /* draw the square */
MoveTo((int) (x[0]*XSCALE), (int) (y[0]*YSCALE));
LineTo((int) (x[1]*XSCALE), (int) (y[1]*YSCALE));
LineTo((int) (x[2]*XSCALE), (int) (y[2]*YSCALE));
LineTo((int) (x[3]*XSCALE), (int) (y[3]*YSCALE));
LineTo((int) (x[0]*XSCALE), (int) (y[0]*YSCALE));
}


void main(void)

/* Main program                               */

{
float x[4],y[4],oldX[4],oldY[4]; /* points in the square */
unsigned i,j;                   /* loop variables */
unsigned color;

InitGraphics();                 /* set up the graphics window */
x[0] = 1.0;   y[0] = 1.0;       /* initialize the square */
x[1] = 2.0;   y[1] = 1.0;
x[2] = 2.0;   y[2] = 2.0;
x[3] = 1.0;   y[3] = 2.0;
color = 0;                      /* draw the square */
DrawSquare(color, x, y);
```

```
for (i = 0; i < 40; ++i) {
   for (j = 0; j < 4; ++j) {    /* save the current location */
      oldX[j] = x[j];
      oldY[j] = y[j];
      }
   RotateSquare(x, y);          /* rotate */
   color = (color + 1) % 3;     /* draw the square */
   DrawSquare(color, x, y);
   }
}
```

Solution to problem 6.8.

```
/* Create a circle using rotation                */

#include <quickdraw.h>
#include <math.h>

#define XSCALE 86               /* x conversion factor */
#define YSCALE 33               /* y conversion factor */
#define pi 3.1415927            /* circumference of a circle */


void InitGraphics (void)

/* Standard graphics initialization            */

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

```
void Rotate (float *x, float *y, float angle, float ox, float oy)

/* 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          */

{
float cosAngle,sinAngle;        /* sin and cos of angle */
float nx;                       /* new x */

*x -= ox;                       /* move the point */
*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;
}


void DrawCircle (float cx, float cy, float radius, int color)

/* Draw a circle                                      */
/*                                                    */
/* Parameters:                                        */
/*    color - color to draw                           */
/*    cx,cy - center of the circle                    */
/*    radius - radius of the circle                   */

{
float x,y;                      /* point on the circle */
unsigned i;                     /* loop variable */

SetSolidPenPat(color);          /* set the pen color */
x = cx;                         /* set the initial coordinates */
y = cy+radius;
                                /* move to the initial point */
MoveTo((int) (x*XSCALE), (int) (y*YSCALE));
for (i = 0; i < 40; ++i) {      /* draw the circle */
   Rotate(&x, &y, pi/20.0, cx, cy);
   LineTo((int) (x*XSCALE), (int) (y*YSCALE));
   }
}
```

```
void main(void)

/* Main program                                */

{
InitGraphics();                 /* set up the graphics window */

DrawCircle(1.5, 1.5, 0.5, 2);   /* draw the circle */
}
```

Solution to problem 6.9.

```
/* Create a star using rotation                */

#include <quickdraw.h>
#include <math.h>

#define XSCALE 86               /* x conversion factor */
#define YSCALE 33               /* y conversion factor */
#define pi 3.1415927            /* circumference of a circle */


void InitGraphics (void)

/* Standard graphics initialization            */

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


void Rotate (float *x, float *y, float angle, float ox, float oy)

/* 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          */
```

```
{
float cosAngle,sinAngle;        /* sin and cos of angle */
float nx;                       /* new x */

*x -= ox;                       /* move the point */
*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;
}


void DrawStar (float cx, float cy, float radius, int color)

/* Draw a star                                          */
/*                                                      */
/* Parameters:                                          */
/*    color - color to draw                             */
/*    cx,cy - center of the star                        */
/*    radius - distance from the center to a point      */

{
int h[5],v[5];                  /* points of the star; QD coordinates */
float x,y;                      /* point on the circle */
unsigned i;                     /* loop variable */

SetSolidPenPat(color);          /* set the pen color */
x = cx;                         /* set the top point */
y = cy-radius;
h[0] = (int) (x*XSCALE);        /* convert it to screen coordinates */
v[0] = (int) (y*YSCALE);
for (i = 1; i < 5; ++i) {       /* find the other 4 points */
   Rotate(&x, &y, pi*2.0/5.0, cx, cy);
   h[i] = (int) (x*XSCALE);
   v[i] = (int) (y*YSCALE);
   }
MoveTo(h[2], v[2]);             /* draw the star */
LineTo(h[0], v[0]);
LineTo(h[3], v[3]);
LineTo(h[1], v[1]);
LineTo(h[4], v[4]);
LineTo(h[2], v[2]);
}
```

```
void main(void)

/* Main program                                */

{
InitGraphics();                 /* set up the graphics window */

DrawStar(1.5, 1.5, 0.5, 1);     /* draw the star */
}
```

Lesson Seven
Types

Defining Types

You have already learned about several different types of variables. So far, we have used character, integer, and float variables. While you probably didn't think about it at the time, int, float, unsigned long, and so forth are all types. In fact, every variable in your C program has a type, although what C means by a type may not be exactly what you expect. To see how the C language thinks of a type, let's look at a variable declaration that looks a lot like some you have defined yourself:

```
char ch, str[40];
```

As you know, this line declares two variables, ch and str. The type of ch is easy enough; it is char. The type of str, though, is slightly more complicated. The type of string is array of char. Technically, you might say that it is an array of 40 char variables, but the fact that the array has 40 elements in only important to the C compiler in a few places. When the compiler reserves space for str in memory, for example, it needs to know how many characters are in the array so it can reserve the right amount of space. In other cases, the C language treats an array with 40 characters the same way it treats an array with 4 characters or 400 characters. As you have already learned from bitter experience, it is up to you to make sure your program doesn't index past the end of the array.

Defining an array this way is pretty clear, but you will soon start to use more complicated types. To use them effectively, and still write programs that are easy to read (which also tends to make them easier to debug, and less likely to have bugs in the first place), it helps to use something called a typedef that actually lets you define a new type that you can use the same way you use char or unsigned long. A typedef looks a lot like a variable declaration. In fact, the only real difference between the way you create a type and the way you create a variable is that a type starts with the reserved word typedef. Here's a couple of examples to get started with:

```
typedef unsigned boolean;
typedef char string[80];
```

In the last lesson, you learned that C uses integers to hold boolean values. (Boolean values are true or false, represented in C as non-zero numbers and zero, respectively.) In some strongly typed languages, there is a whole separate type for boolean variables, even though the computer stores them as integers. In C, where character variables are really just a short form of an integer, you would not expect to see a separate type for boolean variables, but your programs may still be easier to read if you use boolean for the type of a variable. The first typedef allows you to do just that, by defining a new type, called boolean. With this typedef in place, you can define boolean variables, arrays of boolean variables, and so forth, just like you define variables with any other type. For example, you can define the variable done, which we used as a loop terminator in an earlier program, like this:

```
boolean done;
```

Underneath it all, done is still an unsigned integer, and any C compiler would create exactly the same program if you defined it that way. On the other hand, when you look at the declaration of done, you can tell a little more about the variable and what it is used for. You can tell at a glance, for example, that done is used to hold a true or false value.

Our second type is even more useful in many respects, because it gives us something new, not just a new name for something that already exists. After the second typedef, string becomes a new type, just like boolean, but this time it defines an array of 80 characters. With this type, we can define things like

```
string name, friends[47];
```

With these more complicated types, you have to pay attention to keep track of the type of an individual variable. In one sense, the type of name is string, but as far as C is concerned, the type of name is array of char. The second variable, friends, is an array of array of char. Like the arrays of arrays you have already defined, you can specify a string by using one subscript, or an individual character by listing both subscripts.

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. Instead, you are going to learn about types the same way you have learned about for

loops, if statements, && 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 structures and pointers that makes C such a good choice as a programming language.

Problem 7.1. The sample program at the end of lesson 6 used an array of four float variables to keep track of the sides of a square. Modify this program by using a typedef to define a new type, called square, and declare the variables with this new type.

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. C, though, is designed for a broader range of programming jobs, so dealing with numbers and arrays of numbers just isn't enough. C 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 consists of one or more named items. These names become the values that can be assigned to to any variable you could assign an integer to; in fact, the enumeration names are really just another way of specifying an integer. For example, you can define an enumeration of weekdays for a calendar program like this:

```
enum weekDay {sunday, monday,
              teusday, wednesday,
              thursday, friday,
              saturday};
```

The integer values associated with each of the enumerated names start with zero, and increase by one with each name. For example,

```
printf("%d %d\n", sunday,
   teusday);
```

is perfectly legal, and will print 0 and 2.

While you can define an int or unsigned variable to hold the various weekdays, it is considered good form to define variables that are of the same type as the enumeration, like this:

```
enum weekDay dayOff, thanksgiving;
```

This brings up a subtle, but very important point. You have defined a new type, but the name of the type is not weekDay; the type is enum weekDay.

Values can be assigned to the variables just like they are assigned to the variables you are familiar with.

```
thanksgiving = thursday;
```

While it is legal in C to assign an integer value to an enum variable, and to assign an enum constant (like thursday) to an int or unsigned variable, doing so is considered very poor form. You may occasionally write programs that make use of the fact that enumerations are, in fact, integers, just as you occasionally make use of the fact that characters are integers (by comparing or incrementing them, for example), but your programs will be a lot easier to read and understand if you keep your enumerations separate from your integers.

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 in listing 7.1; these will be explained in detail in the next few sections.

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).                   */

#include <stdio.h>
#include <stdlib.h>

                                        /* suits of cards */
enum suit {spades, diamonds, clubs, hearts};
                                        /* face value of cards */
enum value {two, three, four, five, six, seven, eight, nine, ten,
            jack, queen, king, ace};

enum suit suitDeck[52];                 /* these two arrays define */
enum value valueDeck[52];               /* a deck of cards         */


unsigned RandomValue (unsigned max)

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

{
return rand() % max + 1;
}


void InitializeDeck (enum suit suits[52], enum value values[52])

/* Fills in the values to define a sorted deck of cards         */
/*                                                              */
/* Parameters:                                                  */
/*    suits - array of the card suits                           */
/*    values - array of the card values                         */

{
unsigned i;                             /* loop variable */
```

```
  for (i = 0; i < 13; ++i) {              /* initialize the suit array */
     suits[i] = spades;
     suits[i+13] = diamonds;
     suits[i+26] = clubs;
     suits[i+39] = hearts;
     }

  values[0] = two;                        /* initialize the first suit */
  values[1] = three;
  values[2] = four;
  values[3] = five;
  values[4] = six;
  values[5] = seven;
  values[6] = eight;
  values[7] = nine;
  values[8] = ten;
  values[9] = jack;
  values[10] = queen;
  values[11] = king;
  values[12] = ace;
  for (i = 13; i < 52; ++i)               /* copy the first suit to the */
     values[i] = values[i-13];            /* remaining suits            */
  }


  void Shuffle (enum suit suits[52], enum value values[52])

  /* Shuffles the deck of cards                                        */
  /*                                                                   */
  /* Parameters:                                                       */
  /*    suits - array of the card suits                                */
  /*    values - array of the card values                              */

  {
  unsigned i;                             /* loop variable */
  unsigned j;                           /* card to swap with current card */
  enum value tvalue;                      /* temp value; used for swap */
  enum suit tsuit;                        /* temp suit; used for swap */

  for (i = 0; i < 52-1; ++i) {
     j = RandomValue(52 - i) + i - 1;
     tsuit = suits[i];
     suits[i] = suits[j];
     suits[j] = tsuit;
     tvalue = values[i];
     values[i] = values[j];
     values[j] = tvalue;
     }
  }
```

```
void PrintValue (enum value v)

/* Print the value of a card                                    */
/*                                                              */
/* Parameters:                                                  */
/*    v - value of the card                                     */

{
if (v == two)
   printf("two");
else if (v == three)
   printf("three");
else if (v == four)
   printf("four");
else if (v == five)
   printf("five");
else if (v == six)
   printf("six");
else if (v == seven)
   printf("seven");
else if (v == eight)
   printf("eight");
else if (v == nine)
   printf("nine");
else if (v == ten)
   printf("ten");
else if (v == jack)
   printf("jack");
else if (v == queen)
   printf("queen");
else if (v == king)
   printf("king");
else if (v == ace)
   printf("ace");
}


void PrintSuit (enum suit s)

/* Print the suit of a card                                     */
/*                                                              */
/* Parameters:                                                  */
/*    s - suit of the card                                      */

{
if (s == spades)
   printf("spades");
else if (s == diamonds)
   printf("diamonds");
```

```
   else if (s == clubs)
      printf("clubs");
   else if (s == hearts)
      printf("hearts");
   }

   void PrintDeck (enum suit suits[52], enum value values[52])

   /* Prints the cards in order                                    */
   /*                                                              */
   /* Parameters:                                                  */
   /*    suits - array of the card suits                           */
   /*    values - array of the card values                         */

   {
   unsigned i;                              /* loop variable */

   for (i = 0; i < 52; ++i) {
      PrintValue(values[i]);
      printf(" of ");
      PrintSuit(suits[i]);
      printf("\n");
      }
   }


   void main (void)

   /* Main program                                                 */

   {
   srand(1234);                  /* initialize the random number generator */
   InitializeDeck(suitDeck, valueDeck);/* get a new (sorted) deck of cards */
   Shuffle(suitDeck, valueDeck);         /* shuffle the deck */
   PrintDeck(suitDeck, valueDeck);       /* print the shuffled deck */
   }
```

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 subroutines 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:

```
InitializeDeck(suitDeck,
   valueDeck);
Shuffle(suitDeck, valueDeck);
PrintDeck(suitDeck, valueDeck);
```

From a programming standpoint, there is nothing new in the InitializeDeck or Shuffle functions. 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 function in detail.

```
for (i = 0; i < 52-1; ++i) {
   j = RandomValue(52 - i) + i-1;
   tsuit = suits[i];
   suits[i] = suits[j];
   suits[j] = tsuit;
   tvalue = values[i];
   values[i] = values[j];
   values[j] = tvalue;
   }
```

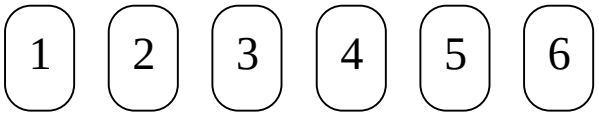
The idea is to shuffle the deck of cards, choosing each card randomly. To do this, we step through the deck of cards using the for loop. On each step, we want to pick one of the cards from the remaining ones, and swap it with the card in the current spot. Comparing this to a real deck of cards, what 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 0, we want to pick from 52 cards. The next time, when i is 1, we want to pick from 52-1 cards, and so forth. This explains the parameter to the RandomValue call.

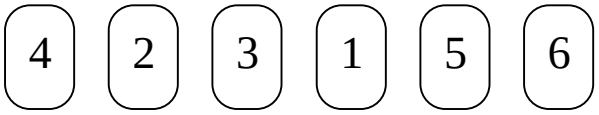
The card we pick must come from the remainder of the deck. When i is 0, Random will return a value from 1 to 52, but we need a value from 0 to 51, since we are using the value as an array index. After subtracting 1 to convert to an array index (remember, i is 0), we swap the card with the current card, and continue on. When i is 1, Random is returning a value from 1 to 51. In this case, adding i and subtracting 1 leave the value unchanged, so we pick a card from the remainder of the deck, leaving out the first card. The process continues through the rest of the deck.

Reading the explanation may not make much sense. This is pretty common the first time you see a new algorithm (a fancy way of saying a new process for doing something). To get a better handle on this program, we'll do something that you will find yourself doing over and over: trying something in real-life to see how a program works. What we will do is to use this algorithm to sort a short deck of cards, tracing through what happens by hand. In our example, though, instead of shuffling 52 cards, we will shuffle 6, and instead of using a random number generator, we'll use a standard, six-sided die.

To start the thought experiment, take a sheet of paper and tear it into six roughly equal pieces, numbering these 1 to 6. Lay these out on the table, like this:

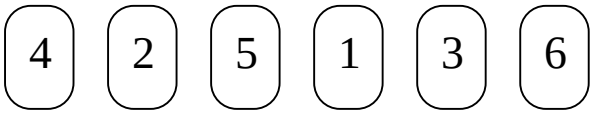


To shuffle the deck, we'll roll the dice to generate random numbers. We'll do this one time for each card except the last one, starting with the first card. Just in case you don't have a die handy, I used a short program to generate some appropriate random numbers. The first one is a four, so we'll switch the first card with the one four cards later (card number 4), with this result:

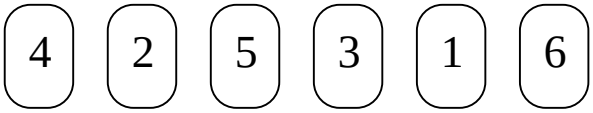


The next step is to move on to the second card. This time, my handy little program gives me a 1. This is an interesting case: we don't do anything at all, and simply move on. "But wait," you say, "the card should be switched with something!" It was, in a sense: it was switched with itself. This may seem a bit odd, but if you think about it, there is always a chance that a particular card will end up in its original place when you shuffle a deck of cards. In fact, the chances are pretty good that at least one card ends up back in its original location.

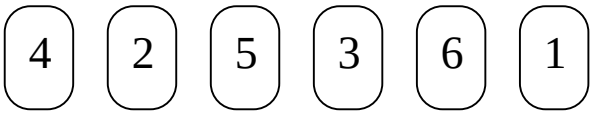
Moving on to the third card, our electronic die comes up with 3, so we switch the third card with card 5, ending up with this:



Next we get a 2, so we switch the fourth card with the one right after it. In this case, we're switching the 1 card with the 3 card. They've both been moved before, but that doesn't hurt anything.



Finally, we get to the fifth card, which is the last one we will switch. This time, though, the computer popped off with a 6, which we can't use. Our program will need a way to make sure that the numbers we get aren't too big, since we don't want to step past the end of an array in a C program. In our case, we just roll again – and Murphy strikes, giving a 3, which is about as useful as the 6. Finally, the computer coughs up a 2, and we switch the last two cards.



The result is a reasonably well-shuffled deck of six cards. You can also see why we only switched five of the cards, instead of six: there's nothing to swap the last card with, anyway, so there is no use in taking the time to do it.

This, in a nutshell, is what the program does, only it uses math to do it instead of using a human to keep track of which card is next, or a die to see what cards to switch. With this though experiment complete, go back and take another look at the program: you should be able to figure out what it does, now, since it is doing the same thing you just finished doing.

Problem 7.2. 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 function 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 its 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.

Structures 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 C 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 C, we do this with structures.

Structures are declared a lot like enumerations. Let's take a look at an example to see how they are actually declared, then discuss the example in detail.

```
struct card {
   enum suit s;
   enum value v;
   };
```

Like an enumeration, a structure creates a new type, but also like an enumeration, the name of the structure is not, itself, a type. In the structure we just defined, the type is struct card, not card. To define a variable of this type, we would do something like this:

```
struct card deck[52];
```

Again, just as with enumerations, the contents of the structure are surrounded by brackets. Inside of the brackets is a series of variable declarations, just like those you would use to declare variables anywhere else. These variables can be literally any type of variable you like, including enumerations, as shown, or even another structure.

When you use a struct variable, you can do almost anything with it that you would do with any other variable. For example, you can assign one structure to another, pass a structure to a function, return a structure as the result of a function, declare arrays of structures, or use structures within another structure. There are limits, though. You can't add two structures together, or do any other math operation on a structure. You also can't print a structure with printf, nor can you read a structure from the keyboard. There are also some limits on structures that don't appear in other parts of C. Structures are strongly typed, so you can't assign one structure to another unless they are the same type of structure, and you can't pass a structure as a parameter to a function that is expecting some other type of structure.

One of the things you need to be able to do, of course, is to use and set the various variables that are inside of the structure. You need a way to specify which field (variables inside a structure are called fields) of the structure you want to access, just as we specify which element of an array we want with an index. With structures, we use the name of the struct variable, followed by a dot, and the name of the variable within the structure. If we define myCard as a card, like this:

```
struct card myCard;
```

we can assign values to the structure, test the structure, and so forth, using the methods we have already learned, and by specifying the individual field of the structure using the dot operator. The examples below show how this is done.

```
myCard.s = diamonds;
myCard.v = ace;

if ((myCard.v >= jack)
   && (myCard.v <= king))
   printf("face card\n");

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 structures, you have to access an element of a structure that is in an array of structures. The first thing you have to name, then, is the array. We'll call it deck.

```
deck
```

Next, you want a specific card within the array, so you name the card by giving an array index.

```
deck[i]
```

Finally, you want a field within the structure. We'll assume you are after the value field. You end up with this:

```
deck[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.

Problem 7.3. 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 functions so you can pass an individual deck of cards, rather than two separate arrays.

Defining Variables Right Away with struct and enum

C is a pretty flexible language. If you look at what you have been doing, you may notice that the definition of an enumeration or structure, and the declaration of a variable for one of them, look a lot alike. You may also have noticed by trial and error that you can mix struct and enum definitions with variable declarations, defining an enum, then a variable, then another enum, for example. It turns out that you can combine them, too. Instead of defining an enumeration and then declaring a variable, like this:

```
enum weekDay {sunday, monday,
              teusday, wednesday,
              thursday, friday,
              saturday};
```

```
enum weekDay dayOff, thanksgiving;
```

you could put them together, like this:

```
enum weekDay {sunday, monday,
              teusday, wednesday,
              thursday, friday,
              saturday}
   dayOff, thanksgiving;
```

I personally think this makes a program a bit harder to read, but we can put this ability to very good use in combination with typedef, as you will see in the next section.

Using typedef with struct and enum

I've tried to make the point that the name of a struct or enum is not, by itself, a type. This is an important concept, since, if you forget to say struct or enum at just the right place, you drive the compiler nuts. The compiler, in turn, spits error messages at you instead of creating a useful program, and you get frustrated.

Well, you wouldn't be alone. As it turns out, though, there is a surprising convention that you can use to get around this problem. You probably know by now that the C compiler doesn't like it if you define two variables that have the same name. As it happens, though, the C language has several different classes of variables, called overloading classes. Within certain limits, you can actually use the same name for two different purposes. In general, that's a rotten idea, but in the case of a struct or enum definition, it comes in handy. To see how, let's redefine our first enum and struct using typedef. As with the other types you have defined with typedef, defining a struct or enum type looks just like a variable declaration, but with a typedef out front.

```
typedef struct card {
   enum suit s;
   enum value v;
   } card;
```

At this point, though, card is both the name of a structure and a new type. You can still use struct card to define variables, but you can also use card as a type, which is a lot easier, and more natural.

```
card deck[52];
```

If the idea of having card declared twice offends your sensibilities, you can also leave off the name of the structure entirely, creating a nameless structure, but defining a type that you can use, instead. This statement defines a type called card; the type is a structure containing two enum variables, but the structure itself has no name.

```
typedef struct {
   enum suit s;
   enum value v;
   } card;
```

While this last form is probably closest to the way other languages use structures, the form with card defined both as a structure and as a type is more common among C programmers, so that is how we will define structures and enumerations in the sample programs and solutions.

Problem 7.4. Try these ideas out by writing a short program that defines an enumeration for the days of the week, but declare the enumeration as both an enum name and as a type. Put the enumeration to use by writing a function that will print a the day you pass it; in other words, if you pass the enum value sunday, your function should write the string "Sunday" using a printf statement. Test your function using a loop in the main function to print each of the days in the week. Naturally, the loop variable and the parameter to your function that prints the day of the week should both be declared as the enumeration type, not using enum or as an integer.

The switch Statement

While this lesson has dealt with types, you may have noticed a few places 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.

```
void PrintValue (enum value v)

/* Print the value of a card           */
/*                                     */
/* Parameters:                         */
/*    v - value of the card            */

{
if (v == two)
   printf("two");
else if (v == three)
   printf("three");
else if (v == four)
   printf("four");
else if (v == five)
   printf("five");
else if (v == six)
   printf("six");
else if (v == seven)
   printf("seven");
else if (v == eight)
   printf("eight");
else if (v == nine)
   printf("nine");
else if (v == ten)
   printf("ten");
else if (v == jack)
   printf("jack");
else if (v == queen)
   printf("queen");
else if (v == king)
   printf("king");
else if (v == ace)
   printf("ace");
}
```

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.

C has a special statement, called the switch statement, that is used in situations like this. The switch statement is like a multiple branch. You give it an expression that can be any integer type, including char or enum, and then list the various values the expression can take on, followed by a colon, and the statement to execute. After evaluating the condition, the switch statement jumps to something called a case label, which must appear in the statement (usually a compound statement) that follows the switch condition. Using a switch statement, the PrintValue function becomes

```
void PrintValue (enum value v)

/* Print the value of a card           */
/*                                     */
/* Parameters:                         */
/*    v - value of the card            */

{
switch (v) {
   case two:   printf("two");
               break;
   case three: printf("three");
               break;
   case four:  printf("four");
               break;
   case five:  printf("five");
               break;
   case six:   printf("six");
               break;
   case seven: printf("seven");
               break;
   case eight: printf("eight");
               break;
   case nine:  printf("nine");
               break;
   case ten:   printf("ten");
               break;
   case jack:  printf("jack");
               break;
   case queen: printf("queen");
               break;
   case king:  printf("king");
               break;
   case ace:   printf("ace");
               break;
   }
}
```

There is another new thing in the switch statement that you probably noticed, but to understand why it is there, you need to know a little more about how the switch statement works. We'll use this short program to explore the details of the switch statement:

```
#include <stdio.h>

void main (void)

{
unsigned i;

for (i = 1; i <= 5; ++i) {
   switch (i) {
      case 1: printf("*");
      case 2: printf("**");
      case 3: printf("***");
      case 4: printf("****");
      case 5: printf("*****");
      }
   printf("\n");
   }
}
```

A simple look at this program might lead you to believe that it would print this:

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

After all, the idea behind the switch statement is to evaluate the switch condition, and branch to the appropriate line in the switch statement. Try running it, though. What really gets printed is this:

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

To see why, fire up the debugger, and trace through the program. The first loop tells the whole story. When i is 1, the program does, in fact, execute the printf that prints a single * character, but then it continues right on, executing the next four printf statements, too.

Now add break statements after each printf statement, like this, and try the program again. Once again, be sure you try the debugger.

```
#include <stdio.h>

void main (void)

{
unsigned i;

for (i = 1; i <= 5; ++i) {
   switch (i) {
      case 1: printf("*");
              break;
      case 2: printf("**");
              break;
      case 3: printf("***");
              break;
      case 4: printf("****");
              break;
      case 5: printf("*****");
      }
   printf("\n");
   }
}
```

This time, right after each printf, the program hits a break statement. The break statement tells the compiler to get out of the switch statement right away.

If you look closely at the program, you may notice that there is no break statement right after the last printf statement. If you were to put one in, it would not cause any harm, but the program ends up doing the same thing either way. In one case, the break statement jumps out of the switch; without the break statement, the program falls through the end of the switch statement. Either way, the program ends up executing the printf that writes the \n character as the next statement.

In some compilers, the program will be a tiny bit shorter and faster if you leave out that last break, while other compilers will create the same program either way. Even if the compiler happens to create a slightly less efficient program, though, the difference is rarely important. You may want to put a break after every series of statements in the switch statement, just as an anti-bug habit. Frankly, I do.

If there is a label for the switch value, it is easy to see what the switch statement does, but what if there isn't? What if, for example, you looped from 1 to 6 in the sample program, instead of 1 to 5? If this happens, the program bypasses all of the statements in the switch statement, almost as if a break were executed right away.

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. In this situation, you can use a special switch statement label called default. It looks just like the other switch statement labels, but statement after the default 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 switch statement, shows how the default label is used.

```
for (i = 1; i < 10; ++i)
   if (i == 1)
      printf("1st\n");
   else if (i == 2)
      printf("2nd\n");
   else if (i == 3)
      printf("3rd\n");
   else
      printf("%dth\n", i);

for (i = 1; i < 10; ++i)
   switch (i) {
      case 1: printf("1st\n");
              break;
      case 2: printf("2nd\n");
              break;
      case 3: printf("3rd\n");
              break;
     default: printf("%dth\n",
                     i);
      }
```

Problem 7.5. Change the card shuffle sample to use switch statements instead of repeated if statements.

Problem 7.6. The fact that you have to remember to put a break statement after each and every series of statements in a switch loop has caused endless bugs down though the ages, but the way C's switch statement works can be put to good use, too. Rewrite the simple sample from this section that prints a series of asterisks on separate lines so that it still prints one, then two, and so forth asterisks, but figure out a way to do it without break statements. The idea you discover can occasionally save you a lot of code in a C program.

Lesson Seven
Solutions to Problems

Solution to problem 7.1.

```
/* Rotate a square 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.                        */

#include <quickdraw.h>
#include <math.h>

#define XSCALE 86               /* x conversion factor */
#define YSCALE 33               /* y conversion factor */
#define pi 3.1415927            /* circumference of a circle */

typedef float square[4];        /* type for a square */


void InitGraphics (void)

/* Standard graphics initialization            */

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


void Rotate (float *x, float *y, float angle, float ox, float oy)

/* 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          */
```

```
{
float cosAngle,sinAngle;        /* sin and cos of angle */
float nx;                       /* new x */

*x -= ox;                       /* move the point */
*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;
}


void RotateSquare (square x, square y)

/* Rotate the square 9 degrees                */
/*                                            */
/* Parameters:                                */
/*    x,y - coordinates of square             */

{
unsigned i;                     /* loop variable */

for (i = 0; i < 4; ++i)
   Rotate(&x[i], &y[i], pi/20.0, 1.5, 1.5);
}


void DrawSquare (int color, square x, square y)

/* Draw the square                            */
/*                                            */
/* Parameters:                                */
/*    color - color to draw                   */
/*    x,y - coordinates of the square         */

{
SetSolidPenPat(color);       /* set the pen color */
                             /* draw the square */
MoveTo((int) (x[0]*XSCALE), (int) (y[0]*YSCALE));
LineTo((int) (x[1]*XSCALE), (int) (y[1]*YSCALE));
LineTo((int) (x[2]*XSCALE), (int) (y[2]*YSCALE));
LineTo((int) (x[3]*XSCALE), (int) (y[3]*YSCALE));
LineTo((int) (x[0]*XSCALE), (int) (y[0]*YSCALE));
}
```

```
void main(void)

/* Main program                                */

{
square x,y,oldX,oldY;          /* points in the square */
unsigned i,j;                  /* loop variables */

InitGraphics();                /* set up the graphics window */
x[0] = 1.0;   y[0] = 1.0;      /* initialize the square */
x[1] = 2.0;   y[1] = 1.0;
x[2] = 2.0;   y[2] = 2.0;
x[3] = 1.0;   y[3] = 2.0;
DrawSquare(0, x, y);           /* draw the square */

for (i = 0; i < 10; ++i) {
   for (j = 0; j < 4; ++j) {   /* save the current location */
      oldX[j] = x[j];
      oldY[j] = y[j];
      }
   RotateSquare(x, y);         /* rotate */
   DrawSquare(3, oldX, oldY);  /* erase the old square */
   DrawSquare(0, x, y);        /* draw the square */
   }
}
```

Solution to problem 7.2.

```
/* 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).                       */

#include <stdio.h>
#include <stdlib.h>
```

```
                                        /* suits of cards */
enum suit {spades, diamonds, clubs, hearts};
                                        /* face value of cards */
enum value {two, three, four, five, six, seven, eight, nine, ten,
            jack, queen, king, ace};

typedef unsigned boolean;               /* set up boolean logic */
#define TRUE 1
#define FALSE 0

typedef enum suit suitDeck[52];         /* these two arrays define */
typedef enum value valueDeck[52];       /* a deck of cards         */

suitDeck suits;                         /* our deck of cards */
valueDeck values;
boolean done;                           /* is the game over? */
unsigned hands;                      /* # of hands played from the deck */
float money;                            /* amount of money left */
unsigned nextCard;                      /* next card in the deck */


unsigned RandomValue (unsigned max)

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

{
return rand() % max + 1;
}


void InitializeDeck (suitDeck suits, valueDeck values)

/* Fills in the values to define a sorted deck of cards         */
/*                                                              */
/* Parameters:                                                  */
/*    suits - array of the card suits                           */
/*    values - array of the card values                         */

{
unsigned i;                             /* loop variable */
```

```
for (i = 0; i < 13; ++i) {              /* initialize the suit array */
   suits[i] = spades;
   suits[i+13] = diamonds;
   suits[i+26] = clubs;
   suits[i+39] = hearts;
   }

values[0] = two;                        /* initialize the first suit */
values[1] = three;
values[2] = four;
values[3] = five;
values[4] = six;
values[5] = seven;
values[6] = eight;
values[7] = nine;
values[8] = ten;
values[9] = jack;
values[10] = queen;
values[11] = king;
values[12] = ace;
for (i = 13; i < 52; ++i)               /* copy the first suit to the */
   values[i] = values[i-13];            /* remaining suits            */
}


void Shuffle(suitDeck suits, valueDeck values)

/* Shuffles the deck of cards                                        */
/*                                                                   */
/* Parameters:                                                       */
/*    suits - array of the card suits                                */
/*    values - array of the card values                              */

{
unsigned i;                             /* loop variable */
unsigned j;                           /* card to swap with current card */
enum value tvalue;                      /* temp value; for swap */
enum suit tsuit;                        /* temp suit; for swap */

for (i = 0; i < 52-1; ++i) {
   j = RandomValue(52 - i) + i - 1;
   tvalue = values[i];
   values[i] = values[j];
   values[j] = tvalue;
   tsuit = suits[i];
   suits[i] = suits[j];
   suits[j] = tsuit;
   }
}
```

```
void PrintValue (enum value v)

/* Print the value of a card                                  */
/*                                                            */
/* Parameters:                                                */
/*    v - value of the card                                   */

{
if (v == two)
   printf("two");
else if (v == three)
   printf("three");
else if (v == four)
   printf("four");
else if (v == five)
   printf("five");
else if (v == six)
   printf("six");
else if (v == seven)
   printf("seven");
else if (v == eight)
   printf("eight");
else if (v == nine)
   printf("nine");
else if (v == ten)
   printf("ten");
else if (v == jack)
   printf("jack");
else if (v == queen)
   printf("queen");
else if (v == king)
   printf("king");
else if (v == ace)
   printf("ace");
}


void PrintSuit (enum suit s)

/* Print the suit of a card                                   */
/*                                                            */
/* Parameters:                                                */
/*    s - suit of the card                                    */
```

```
{
if (s == spades)
   printf("spades");
else if (s == diamonds)
   printf("diamonds");
else if (s == clubs)
   printf("clubs");
else if (s == hearts)
   printf("hearts");
}

void PrintCard (enum suit s, enum value v)

/* Print a card                                                    */
/*                                                                 */
/* Parameters:                                                     */
/*    s - suit                                                     */
/*    v - values                                                   */

{
PrintValue(v);
printf(" of ");
PrintSuit(s);
printf("\n");
}


void PlayHand (void)

/* 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                                 */

{
float bet;                              /* player's bet */
enum value v1,v2,v3;                    /* value of the three cards */

printf("\nI draw:\n");
v1 = values[nextCard];                  /* draw the first card */
PrintCard(suits[nextCard], v1);
++nextCard;
v2 = values[nextCard];                  /* draw the second card */
PrintCard(suits[nextCard], v2);
++nextCard;
```

```
   if (v2 < v1) {                          /* sort the values */
      v3 = v2;
      v2 = v1;
      v1 = v3;
      }
   do {                                    /* get the bet */
      printf("You have %.2f left.\nYour bet:", money);
      scanf(" %f", &bet);
      if (bet < 0.0)
         done = TRUE;
      else if (bet > money)
         printf("Sorry, you don't have that much.\n");
      }
   while (bet > money);
   if (! done) {
      v3 = values[nextCard];               /* draw the third card */
      printf("Your card is:\n");
      PrintCard(suits[nextCard], v3);
      ++nextCard;
      if ((v1 < v3) && (v3 < v2)) {
         money += bet;                     /* player wins */
         printf("You win!\n");
         }
      else {
         money -= bet;                     /* player looses */
         printf("Sorry, you loose.\n");
         if (money <= 0.0) {               /* see if he's broke */
            printf("You are out of money.  So long!\n");
            done = TRUE;
            }
         }
      }
   }


   void GetSeed (void)

   /* Initialize the random number generator                        */

   {
   int i;                                  /* integer from keyboard */

   printf("Please enter a number from\n1000 to 30000:");
   scanf(" %d", &i);
   srand(i);
   }
```

```
void main (void)

/* Main program                                                    */

{
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 */
do {
   if (hands = 17) {             /* reshuffle after 17 hands */
      Shuffle(suits, values);    /* shuffle the deck */
      hands = 0;                 /* no hands played from the deck */
      nextCard = 1;              /* next card to draw */
      }
   PlayHand();                   /* play one hand of Acey Ducey */
   ++hands;                      /* update the # of hands played */
   }
while (!done);
}
```

Solution to problem 7.3.

```
/* 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).                   */

#include <stdio.h>
#include <stdlib.h>

                                        /* suits of cards */
enum suit {spades, diamonds, clubs, hearts};
                                        /* face value of cards */
enum value {two, three, four, five, six, seven, eight, nine, ten,
            jack, queen, king, ace};

struct card {                           /* one card in a deck */
             enum value v;
             enum suit s;
            };

struct card deck[52];                   /* our deck of cards */
```

```
unsigned RandomValue (unsigned max)

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

{
return rand() % max + 1;
}


void InitializeDeck (struct card deck[52])

/* Fills in the values to define a sorted deck of cards         */
/*                                                              */
/* Parameters:                                                  */
/*    deck - array of cards                                     */

{
unsigned i;                             /* loop variable */

for (i = 0; i < 13; ++i) {              /* initialize the suit array */
   deck[i].s = spades;
   deck[i+13].s = diamonds;
   deck[i+26].s = clubs;
   deck[i+39].s = hearts;
   }

deck[0].v = two;                        /* initialize the first suit */
deck[1].v = three;
deck[2].v = four;
deck[3].v = five;
deck[4].v = six;
deck[5].v = seven;
deck[6].v = eight;
deck[7].v = nine;
deck[8].v = ten;
deck[9].v = jack;
deck[10].v = queen;
deck[11].v = king;
deck[12].v = ace;
for (i = 13; i < 52; ++i)               /* copy the first suit to the */
   deck[i].v = deck[i-13].v;            /* remaining suits            */
}
```

```
void Shuffle(struct card deck[52])

/* Shuffles the deck of cards                                    */
/*                                                               */
/* Parameters:                                                   */
/*    deck - array of cards                                      */

{
unsigned i;                             /* loop variable */
unsigned j;                            /* card to swap with current card */
struct card tcard;                      /* temp card; for swap */

for (i = 0; i < 52-1; ++i) {
   j = RandomValue(52 - i) + i - 1;
   tcard = deck[i];
   deck[i] = deck[j];
   deck[j] = tcard;
   }
}


void PrintValue (enum value v)

/* Print the value of a card                                  */
/*                                                            */
/* Parameters:                                                */
/*    v - value of the card                                   */

{
if (v == two)
   printf("two");
else if (v == three)
   printf("three");
else if (v == four)
   printf("four");
else if (v == five)
   printf("five");
else if (v == six)
   printf("six");
else if (v == seven)
   printf("seven");
else if (v == eight)
   printf("eight");
else if (v == nine)
   printf("nine");
else if (v == ten)
   printf("ten");
else if (v == jack)
   printf("jack");
else if (v == queen)
```

```
   printf("queen");
else if (v == king)
   printf("king");
else if (v == ace)
   printf("ace");
}


void PrintSuit (enum suit s)

/* Print the suit of a card                                    */
/*                                                             */
/* Parameters:                                                 */
/*    s - suit of the card                                     */

{
if (s == spades)
   printf("spades");
else if (s == diamonds)
   printf("diamonds");
else if (s == clubs)
   printf("clubs");
else if (s == hearts)
   printf("hearts");
}

void PrintDeck (struct card deck[52])

/* Prints the cards in order                                      */
/*                                                                */
/* Parameters:                                                    */
/*    deck - array of cards                                       */

{
unsigned i;                              /* loop variable */

for (i = 0; i < 52; ++i) {
   PrintValue(deck[i].v);
   printf(" of ");
   PrintSuit(deck[i].s);
   printf("\n");
   }
}
```

```
void main (void)

/* Main program                                                  */

{
srand(1234);                  /* initialize the random number generator */
InitializeDeck(deck);           /* get a new (sorted) deck of cards */
Shuffle(deck);                  /* shuffle the deck */
PrintDeck(deck);                /* print the shuffled deck */
}
```

Solution to problem 7.4.

```
/* This program demonstrates typedef used with enum */

#include <stdio.h>

typedef enum weekDay {
                       sunday, monday, teusday, wednesday, thursday,
                       friday, saturday
                      } weekDay;

void PrintDay (weekDay day)

/* Print the day of the week                                  */
/*                                                            */
/* Parameters:                                                */
/*    day - day of the week                                   */

{
if (day == sunday)
   printf("Sunday");
else if (day == monday)
   printf("Monday");
else if (day == teusday)
   printf("Teusday");
else if (day == wednesday)
   printf("Wednesday");
else if (day == thursday)
   printf("Thursday");
else if (day == friday)
   printf("Friday");
else if (day == saturday)
   printf("Saturday");
}
```

```
void main(void)

{
weekDay day;

for (day = sunday; day <= saturday; ++day) {
   PrintDay(day);
   printf("\n");
   }
}
```

Solution to problem 7.5.

```
/* 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).                   */

#include <stdio.h>
#include <stdlib.h>

                                        /* suits of cards */
enum suit {spades, diamonds, clubs, hearts};
                                        /* face value of cards */
enum value {two, three, four, five, six, seven, eight, nine, ten,
            jack, queen, king, ace};

enum suit suitDeck[52];                 /* these two arrays define */
enum value valueDeck[52];               /* a deck of cards         */


unsigned RandomValue (unsigned max)

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

{
return rand() % max + 1;
}
```

```
void InitializeDeck (enum suit suits[52], enum value values[52])

/* Fills in the values to define a sorted deck of cards         */
/*                                                              */
/* Parameters:                                                  */
/*    suits - array of the card suits                           */
/*    values - array of the card values                         */

{
unsigned i;                             /* loop variable */

for (i = 0; i < 13; ++i) {              /* initialize the suit array */
   suits[i] = spades;
   suits[i+13] = diamonds;
   suits[i+26] = clubs;
   suits[i+39] = hearts;
   }

values[0] = two;                        /* initialize the first suit */
values[1] = three;
values[2] = four;
values[3] = five;
values[4] = six;
values[5] = seven;
values[6] = eight;
values[7] = nine;
values[8] = ten;
values[9] = jack;
values[10] = queen;
values[11] = king;
values[12] = ace;
for (i = 13; i < 52; ++i)               /* copy the first suit to the */
   values[i] = values[i-13];            /* remaining suits            */
}


void Shuffle (enum suit suits[52], enum value values[52])

/* Shuffles the deck of cards                                   */
/*                                                              */
/* Parameters:                                                  */
/*    suits - array of the card suits                           */
/*    values - array of the card values                         */

{
unsigned i;                               /* loop variable */
unsigned j;                             /* card to swap with current card */
enum value tvalue;                        /* temp value; used for swap */
enum suit tsuit;                          /* temp suit; used for swap */
```

```
   for (i = 0; i < 52-1; ++i) {
      j = RandomValue(52 - i) + i - 1;
      tsuit = suits[i];
      suits[i] = suits[j];
      suits[j] = tsuit;
      tvalue = values[i];
      values[i] = values[j];
      values[j] = tvalue;
      }
   }


   void PrintValue (enum value v)

   /* Print the value of a card                                    */
   /*                                                              */
   /* Parameters:                                                  */
   /*    v - value of the card                                     */

   {
   switch (v) {
      case two:   printf("two");
                  break;
      case three: printf("three");
                  break;
      case four:  printf("four");
                  break;
      case five:  printf("five");
                  break;
      case six:   printf("six");
                  break;
      case seven: printf("seven");
                  break;
      case eight: printf("eight");
                  break;
      case nine:  printf("nine");
                  break;
      case ten:   printf("ten");
                  break;
      case jack:  printf("jack");
                  break;
      case queen: printf("queen");
                  break;
      case king:  printf("king");
                  break;
      case ace:   printf("ace");
                  break;
      }
   }
```

```
void PrintSuit (enum suit s)

/* Print the suit of a card                                    */
/*                                                            */
/* Parameters:                                                */
/*    s - suit of the card                                    */

{
switch (s) {
   case spades:    printf("spades");
                   break;
   case diamonds:  printf("diamonds");
                   break;
   case clubs:     printf("clubs");
                   break;
   case hearts:    printf("hearts");
                   break;
   }
}

void PrintDeck (enum suit suits[52], enum value values[52])

/* Prints the cards in order                                  */
/*                                                            */
/* Parameters:                                                */
/*    suits - array of the card suits                         */
/*    values - array of the card values                       */

{
unsigned i;                              /* loop variable */

for (i = 0; i < 52; ++i) {
   PrintValue(values[i]);
   printf(" of ");
   PrintSuit(suits[i]);
   printf("\n");
   }
}


void main (void)

/* Main program                                               */

{
srand(1234);                  /* initialize the random number generator */
InitializeDeck(suitDeck, valueDeck);/* get a new (sorted) deck of cards */
Shuffle(suitDeck, valueDeck);         /* shuffle the deck */
PrintDeck(suitDeck, valueDeck);       /* print the shuffled deck */
```

```
}
```

Solution to problem 7.6.

```
/* Demonstrates that leaving out the break statement    */
/* can occasionally result in shorter programs.         */

#include <stdio.h>

void main (void)

{
unsigned i;

for (i = 1; i <= 5; ++i) {
   switch (i) {
      case 5: printf("*");
      case 4: printf("*");
      case 3: printf("*");
      case 2: printf("*");
      case 1: printf("*");
      }
   printf("\n");
   }
}
```

Lesson Eight
Pointers and Lists

What is a Pointer?

By now, you have used two very powerful techniques to organize information in C. Arrays are use to handle a large amount of information when all of the pieces are the same type, while structures 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. One 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.

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 structure 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 is one of the most common uses for the pointer, a data type you have already seen used to pass variables to functions. A pointer *points to* a memory location. In terms of the C program, a pointer points to a variable. The variable can be a simple variable, like an integer or a floating-point number; a structure; 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. 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, the * operator, which is easy to confuse with the multiplication operator. 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 C, 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 C that handles all of those mucky details for you!

A realistic example of how pointers and dynamic memory 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 C 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!

You have already had one brief look at pointers, back when we discussed how to use them to pass a value to a function in such a way that the function could change the variable we passed. In that case, though, you only defined a pointer as a parameter. It turns out that you can use the same idea to declare pointers as either global or local variables. No matter where you define a pointer, though, and no matter what it points to, the general mechanism is the same: you put an * character right before the name of the variable. The variable is then a pointer to the type that it would have been without the * character in place. For example, to define a pointer to a char, you could use the declaration

```
char *cptr;
```

A pointer is a strange, hybrid type. The type of the variable cptr is pointer to char, and a pointer is a very real thing, so you can set a pointer to point to a variable, set a pointer to point to some dynamically allocated piece of memory, set a pointer to point to nothing in particular, or even set a pointer to point to the same thing as some other pointer. In all of these cases, you use the name of the pointer, just as you would use the name of a variable to set or access the variable. In the end, though, you will want to get at the thing the pointer points to, not the pointer itself. To do that, you put the * character right before the variable name, just as you did when the pointer was declared. Any time the * comes before the variable name, C will load or store to the value the pointer points at, not to 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!)

```
#include <stdio.h>

void main(void)

{
int i, j, *ip;

j = 4;
*ip = j;
i = *ip;
printf("%d\n", i);
}
```

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.

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?

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. You have used the address operator in the past to set the value of a pointer, but we need to learn a new way to set up a pointer, now. The C standard library gives us a function called malloc to get some new memory. When you are finished with the memory, the function free can be used to get rid of the memory. Both functions are declared in the header file stdlib.h, but you have to be careful here: some older C compilers declare malloc in other header files, and a few don't declare it at all. If you are moving your program from system to system, you need to be sure you read the documentation for each

compiler to see how malloc is implemented in the other compilers. If the compiler follows the ANSI C standard, though, malloc and free will be in stdlib.h.

We can change our program from the last section into a safe one using these function. This program is one you can run!

```
#include <stdio.h>
#include <stdlib.h>

void main (void)

{
int i, j, *ip;

ip = (int *) malloc (sizeof(int));
j = 4;
*ip = j;
i = *ip;
printf("%d\n", i);
free(ip);
}
```

When this program runs, it starts by making a call to malloc. This simple call shows a lot of new concepts, so we need to stop and spend some time looking at it closely. The malloc function needs to know how much memory you want, in bytes. Since you are allocating a pointer to an integer, you need to allocate two bytes, but the size of an integer can vary from one C compiler to another. Worse yet, later you will be allocating memory for structures, and it can be pretty easy to make a mistake on the size of a structure, even if you know your compiler well. C solves this problem with a special operator called sizeof. The sizeof operator returns the size of any type in the same units malloc uses (bytes, in our case). If you move this program to another compiler that uses a different size of integer, it will still work, because the new compiler's sizeof function will know how big integers are in that implementation.

The sizeof operator is something that has changed a bit over the years, so it is also something you should be careful of if you are moving your programs between computers or typing in C programs from another book. In older implementations of C, the value returned by sizeof was generally an int or unsigned value. The problem with this is that you could never use sizeof on a really big structure or array, since this effectively limits the value returned by sizeof to 64K-1 on most computers, and as you know, you have a lot more memory at your disposal on the Apple IIGS. In ANSI C implementations, the value returned is of type size_t, which is an unsigned integer big enough to hold a pointer value. In ORCA/C, this means the integer is an unsigned long, not an unsigned int. The main place this comes up is when you want to print the result of the sizeof function: many people try to use the %d conversion specifier to do this, and that is simply not correct, since the %d conversion specifier is only valid with two-byte values in ORCA/C. Instead, you need to use the %ld conversion specifier. On other compilers, though, this could be different.

The malloc function performs some advanced magic and, after the call, two bytes of memory have

How malloc and free 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 C 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 malloc 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 free 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.

been obtained. The exact process involved in getting this memory is a bit involved, and not particularly important to you, the C 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. The type of the pointer that is returned is then cast to the type of the pointer we want, just to keep things neat. This isn't actually required in ORCA/C, but neatness never hurts, and this type cast may be necessary in other implementations of C. To cast one pointer to another type, you just put the type in parenthesis in front of the pointer value.

Just before the program ends, you see a call to free. This function goes through a complicated mechanism that gets rid of the two bytes of memory. After calling free, 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 free, 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/C, 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. (In the variables window, you look at the value of a pointer by placing a ^ after the name, instead of a * before the name. The effect is the same.) 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, or, to the debugger, ip^. Stepping through the program, you can see malloc allocate new memory, and you can see the assignment setting the value being pointed to. Free 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 floating-point 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 floating-point 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 printf call.

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 malloc, 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 free to deallocate the memory areas reserved by the calls to malloc.

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 simple 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 malloc 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 structures. Each of the structures in the linked list contains, among other things, a pointer. The pointer points to another structure in the list. A single pointer variable in the program points to the first structure in the linked list.

For our first look at a linked list, we will create a list of integers. The structure, then, must have a pointer to the next record, and an integer. It looks like this:

```
typedef struct listElement {
   struct listElement *next;
   int i;
   } listElement;

listElement *list, *temp;
```

With these definitions, we can start to create a linked list. For each element in the list, we will need to call malloc to get space for a new record, and then place a value into the integer, like this:

```
temp = (listElement *)
   malloc(sizeof listElement);
(*temp).i = 4;
```

The assignment that places a 4 in the integer variable looks a bit odd. You might not think about putting parenthesis in a C expression to tell the compiler in what order to apply the * and . operators, but it actually does work. As with the other examples of * and . that you have seen, the * operator applied to temp tells the compiler to look at the thing temp points to, not the pointer temp. Applying the . operator and the variable name i, you are telling the compiler to look at a variable i, contained within a struct pointed at by the pointer temp. That's a mouthful, but if you look at it closely and apply the old principals you already know one after the other, it makes some sense.

The assignment is, however, a bit confusing, not to mention hard to type, and it turns out that you need to do this sort of thing a lot in real programs. C has another operator, ->, which does the job of the * and . operators put together, and makes the line a little easier to read at the same time:

```
temp->i = 4;
```

The -> operator, used instead of the . operator, tells the compiler that the variable temp is a pointer to a structure, rather than a structure, and to do the right things on its own.

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 out 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 NULL, defined in stddef.h. NULL has a type of void *, which means a pointer to some unspecified object, and so is type compatible with any pointer. You can set a pointer to NULL or compare a pointer to NULL. By convention, NULL is used to mean that the pointer doesn't point to anything, and that is how we mark the end of our list. By initializing list to NULL at the start of the program, and checking to see if list is NULL 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.

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.                                                     */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
```

```
typedef struct listElement {
   struct listElement *next;
   int i;
   } listElement;

listElement *list;


void GetList (void)

/* Read a list from the keyboard.                                  */
/*                                                                 */
/* Variables:                                                      */
/*    list - pointer to the head of the list                       */

{
int i;                              /* variable read from the keyboard */
listElement *temp;                     /* work pointer */

list = NULL;                           /* initialize the list pointer */

do {
   scanf(" %d", &i);                   /* read a value */
   if (i != 0) {                    /* if not at the end of the list... */
                                       /* allocate a structure */
      temp = (listElement *) malloc (sizeof(listElement));
      temp->i = i;                     /* place i in the structure */
      temp->next = list;              /* put the structure in the list */
      list = temp;
      }
   }
while (i != 0);
}


void PrintList (void)

/* Print a list.                                                   */
/*                                                                 */
/* Parameters:                                                     */
/*    list - pointer to the head of the list                       */

{
listElement *temp;                     /* work pointer */

while (list != NULL) {
   temp = list;                        /* remove an item from the list */
   list = temp->next;
   printf("%d\n", temp->i);            /* write the value */
   free(temp);                         /* free the memory */
```

```
    }
}


void main (void)

{
GetList();                              /* read a list */
PrintList();                            /* print a list */
}
```

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 structure, saves the number in the structure, and puts the structure in the list.

PrintList loops for as long as there are entries left in the list. Each time through the loop, the top structure in the list is removed from the list, the value is printed, and the memory used by the structure 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 unused memory for the malloc 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:

```
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 warn 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, 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 used in such diverse applications as burglar alarms, data 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.

Change the sample program from this section by adding a new function called ReverseList that reverses the list. Use this function to reverse the list before it is printed.

Problem 8.4. In some applications, we read in a list, then scan the list repeatedly, looking for structures with certain characteristics. For example, in a burglar alarm, we might use one function 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. Scan the list, incrementing a counter in an array when you find a value in the range 1 to 5, and ignoring all other entries. 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 0
```

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 NULL. 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 structures 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 NULL, 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 structure is always set to NULL. From then on, we add a new structure by setting the next pointer in the structure pointed to by last to point to the new structure, and then set last to point to the new structure. In C code, then, we set the list up like this:

```
list = NULL;
last = NULL;
```

To add a record to the end of the list, we check to see if the structure 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 == NULL) {
   list = temp;
   last = temp;
   }
else {
   last->next = temp;
   last = temp;
   }
```

Of course, since 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 == NULL)
   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 NULL when we initialize the list is also unnecessary.

Problem 8.5. 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, malloc returns NULL instead of a pointer to the new memory. You can – and should – check for this possibility in your program, and handle the situation if it arises. Here's a sample program that allocates chunks of 10,000 bytes of memory at a time. It will give you some idea about how much memory you actually have available when your program runs, and give you a good example of handling an out of memory situation, too. Of course, the amount of memory you have available can change due to a number of factors. The memory you can allocate will go up if you add more memory, make any RAM disk smaller, or run the program from the Finder or text shell instead of from the desktop development environment. The amount of memory you can allocate can go up or down as the version of the operating system or compiler change, and even as you add, delete, or change the desk accessories you use.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

void main (void)

{
unsigned count;
int *p;

count = 0;

do {
   p = (int *) malloc (10000);
   ++count;
   }
while (p != NULL);
printf("%d 10000 byte areas"
   " allocated.\n", count-1);
}
```

One interesting point about this program is that we aren't using free to dispose of the memory we allocate. In this rare case, that does no harm, since the operating system will free the memory once the program stops running.

The & Operator Returns an Address

It may have struck you as a little odd that we went through all of those contortions with malloc to allocate space for an integer value when you already know how to create an integer by simply declaring it. Well, the biggest reason was to prepare you for using malloc to create lists, but as it turns out, you can actually get the address of a normal variable, and use that address to set the pointer. The method used is exactly the same as when you pass the address of a variable as a parameter to functions like scanf. &i, for example, is the address of i, and you can use this to set a pointer.

Problem 8.6. The very first sample program in this section was:

```
#include <stdio.h>

void main(void)

{
int i, j, *ip;

j = 4;
*ip = j;
i = *ip;
printf("%d\n", i);
}
```

Change this program by setting ip to point to j, rather than setting the value ip points to to the value of j, then run the program. Be sure to use the debugger to see what is happening.

The Special Relationship Between Pointers and Arrays

In C, there is a very close relationship between arrays and pointers. In fact, if a is an array of integers, the type of a in an expression isn't really array at all – it is a pointer! Let's use a simple program to get a first look at how this works.

```
#include <stdio.h>

void main (void)

{
int a[2], *ip;

a[0] = 4;
ip = a;
printf("%d\n", *ip);
}
```

The first line is simple enough – it assigns 4 to the first element of the array. The next line is where the neat stuff happens. Here, we assign a to ip. What does this mean? Well, since a is an array of integers, C treats it as a pointer to an integer in this case, and ip ends up pointing to the first element of the array a. When you print *ip, the value printed is 4 – the same value you placed at the start of the array.

It is this special relationship that we put to use a few lessons back to pass an array as a parameter to a function. C didn't really pass the array at all – instead, C passed a pointer to the first element in the array.

The relationship works the other way around, too. You can actually use an array subscript after a pointer to access the value the pointer points to. For example, you can look at the value ip points to using *ip, as you normally would, or by using ip[0], which means exactly the same thing in C. That's why you were able to use array subscripts inside of the subroutine to access values in the array that C passed as a pointer. You can even index past the value ip currently points to; ip[1] is the integer after the one *ip points to, for example.

Problem 8.7. Use the idea that array indexes can be used with a pointer to modify the sample program from this section. Start by declaring an array with three values, and set these values to 1, 2 and 3. Next, set a pointer to the first element of the array. Finally, print the values in the array by indexing 0, 1 and 2 past the pointer.

Pointer Math

The fact that arrays and pointers enjoy such a close relationship in C becomes a powerful new programming technique when combined with another factor. You can actually do some math with pointers in C, too. The math is limited to the things that make sense with a pointer, which basically means that you can add or subtract an integer value, or increment or decrement a pointer. C handles all of this very gracefully, too. When you add one to a pointer, the result is a pointer to the next item of the same type after the original pointer!

```
#include <stdio.h>

void main (void)

{
int a[5], *ip, i;

for (i = 0; i < 5; ++i)
   a[i] = i+1;

ip = a;
i = 0;
for (i = 0; i < 5; ++i) {
   printf("%d\n", *ip);
   ++ip;
   }
}
```

This program shows how this ability is used most often in C. You see, it takes a fair amount of time to calculate the location of a value in an array, like a[4]. It takes even longer if the array is subscripted multiple times, like a[i][j][k]. This sort of multiple indexing may seem a bit strange, but it comes up a lot in some kinds of engineering programs. While it takes a long time to compute array indices, though, incrementing the value of a pointer is a very fast operation by comparison, and accessing the value a pointer points to is also a very fast operation. In short, you can use this idea of pointer math to create some very short, fast programs in C – programs whose speed is much greater than a similar program in other high-level languages.

Let's explore this concept further by writing a function that will work like strcmp, but will do a case insensitive compare – i.e., one where the characters 'A' and 'a' are treated as being equal. Using arrays, our function looks like this:

```
int strcmp2 (char str1[1],
             char str2[1])

{
unsigned i;

i = 0;
while ((str1[i] != 0)
       && (str2[i] != 0)) {
   if (toupper(str1[i]) !=
       toupper(str2[i])) {
      if (toupper(str1[i]) <
          toupper(str2[i]))
         return -1;
      return 1;
      }
   ++i;
   }
if (str1[i] != 0)
   return 1;
if (str2[i] != 0)
   return -1;
return 0;
}
```

This subroutine has one distinct advantage: you can see the rules that strcmp uses to compare strings layed out very clearly. The subroutine scans the strings, checking to see if it has reached the null character in either string, and stopping if it has. Assuming both strings still have characters, the subroutine enters the while loop, and checks to see if the uppercase equivalent of these character is the same; if not, it checks to see which is smaller, and returns -1 or 1, as appropriate. If the characters match, the subroutine increments i and continues on to the next position in the strings.

Assuming the end of one or the other string is reached without finding a mismatch, the subroutine drops out of the do loop, and checks to see if one string is shorter than the other. If so, an appropriate value is returned. If not, the strings are the same, and a 0 is returned.

C doesn't care if you declare the parameters to this function as arrays, as we have done, or pointers: the compiler will let you call the function with the same sort of arguments either way. Using this fact, we can convert this subroutine to use pointers instead of arrays. The first effect is that we no longer need the variable i.

```
int strcmp2 (char *p1, char *p2)

{
while ((*p1 != 0) && (*p2 != 0))
   {
   if (toupper(*p1) !=
       toupper(*p2)) {
      if (toupper(*p1) <
          toupper(*p2))
         return -1;
      return 1;
      }
   ++p1;
   ++p2;
   }
if (*p1 != 0)
   return 1;
if (*p2 != 0)
   return -1;
return 0;
}
```

This seems a little shorter, and in fact, it is faster, too. We'll continue to refine this program, using our knowledge of C to continue improving on the size and, as it turns out, the speed of this subroutine. This process of iterative refinement of a subroutine is a very common one. The next step is to recognize that, in C, 0 is false and anything else is true, so all of those comparisons to 0 are really unnecessary.

```
int strcmp2 (char *p1, char *p2)

{
while (*p1 && *p2)
   {
   if (toupper(*p1) !=
       toupper(*p2)) {
      if (toupper(*p1) <
          toupper(*p2))
         return -1;
      return 1;
      }
   ++p1;
   ++p2;
   }
if (*p1)
   return 1;
if (*p2)
   return -1;
return 0;
}
```

We could carry this process even further, but it would take some ideas about the C language that you haven't seen yet. In any case, the process can be taken to the point where the subroutine is very efficient, but also very hard to follow. (Maybe you think this is already true!) This brings up an interesting philosophical point. In many cases, like the sample programs in this course, it is very important that programs be easy to follow and understand. This helps a lot when you are debugging a program, too. In other situations, you need to milk as much speed as possible from your computer, and that sometimes means going over code again and again to make it smaller and more efficient. The result may be very difficult for anyone, even yourself, to decipher later. Some people will get stuck in one more or the other. A teacher, for example, might get stuck in the "write it clearly" mode, while a professional programmer writing tight animation code in C might get caught up in obscure techniques to cut a few percent from the execution time, producing unreadable code. I hope you can be flexible enough to see that both views are correct, and both have a place. My advice is to start by writing clear, easy to understand subroutines. If the program is too slow, go back and change individual subroutines that are taking up too much time to get more efficiency.

Problem 8.8. Write a short program to test the last version of strcmp2. Write your program so that it passes each of the following at least one time:

a. A string constant.
b. A character array.
c. A pointer to a character. (The pointer can be initialized to point to the character array.)

Lesson Eight
Solutions to Problems

Solution to problem 8.1.

```
#include <stdio.h>
#include <stdlib.h>

void main (void)

{
float *fp;

fp = (float *) malloc (sizeof(float));
*fp = 1.2;
printf("%f\n", *fp);
free(fp);
}
```

Solution to problem 8.2.

```
#include <stdio.h>
#include <stdlib.h>

void main (void)

{
int *ip, *jp, *kp;

ip = (int *) malloc (sizeof(int));
jp = (int *) malloc (sizeof(int));
kp = (int *) malloc (sizeof(int));

*ip = 4;
*jp = 6;
*kp = *ip + *jp;

printf("%d\n", *kp);

free(ip);
free(jp);
free(kp);
}
```

Solution to problem 8.3.

```
/* This program reads in a list of integers, and then prints    */
/* them.  The program stops when a zero value is read.          */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

typedef struct listElement {
   struct listElement *next;
   int i;
   } listElement;

listElement *list;


void GetList (void)

/* Read a list from the keyboard.                               */
/*                                                              */
/* Variables:                                                   */
/*    list - pointer to the head of the list                    */

{
int i;                              /* variable read from the keyboard */
listElement *temp;                     /* work pointer */

list = NULL;                           /* initialize the list pointer */

do {
   scanf(" %d", &i);                   /* read a value */
   if (i != 0) {                    /* if not at the end of the list... */
                                       /* allocate a structure */
      temp = (listElement *) malloc (sizeof(listElement));
      temp->i = i;                     /* place i in the structure */
      temp->next = list;               /* put the record in the list */
      list = temp;
      }
   }
while (i != 0);
}
```

```
void ReverseList (void)

/* Reverse a list                                                */
/*                                                               */
/* Parameters:                                                   */
/*    list - pointer to the head of the list                     */

{
listElement *temp;                    /* work pointer */
listElement *nlist;                   /* new list pointer */

nlist = NULL;                         /* start out with an empty list */

while (list != NULL) {
   temp = list;                       /* remove an item from the list */
   list = temp->next;
   temp->next = nlist;                /* add it to the new list */
   nlist = temp;
   }
list = nlist;                         /* switch to the new list */
}


void PrintList (void)

/* Print a list.                                                 */
/*                                                               */
/* Parameters:                                                   */
/*    list - pointer to the head of the list                     */

{
listElement *temp;                    /* work pointer */

while (list != NULL) {
   temp = list;                       /* remove an item from the list */
   list = temp->next;
   printf("%d\n", temp->i);           /* write the value */
   free(temp);                        /* free the memory */
   }
}


void main (void)

{
GetList();                            /* read a list */
ReverseList();                        /* reverse the list */
PrintList();                          /* print a list */
}
```

Solution to problem 8.4.

```
/* This program reads in a list of integers, then scans the     */
/* list, printing the number of times each of the integers 1 to */
/* 5 appear in the list.                                        */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

typedef struct listElement {
   struct listElement *next;
   int i;
   } listElement;

listElement *list;


void GetList (void)

/* Read a list from the keyboard.                          */
/*                                                         */
/* Variables:                                              */
/*    list - pointer to the head of the list               */

{
int i;                              /* variable read from the keyboard */
listElement *temp;                     /* work pointer */

list = NULL;                           /* initialize the list pointer */

do {
   scanf(" %d", &i);                   /* read a value */
   if (i != 0) {                    /* if not at the end of the list... */
                                       /* allocate a structure */
      temp = (listElement *) malloc (sizeof(listElement));
      temp->i = i;                     /* place i in the structure */
      temp->next = list;               /* put the record in the list */
      list = temp;
      }
   }
while (i != 0);
}
```

```
void CountList (void)

/* Count the elements in the list                          */
/*                                                         */
/* Variables:                                              */
/*    list - pointer to the head of the list               */

{
listElement *temp;                    /* work pointer */
unsigned counts[5];                   /* array of frequency counts */
unsigned i;                           /* loop counter */

for (i = 0; i < 5; ++i)               /* initialize the array */
   counts[i] = 0;
temp = list;                          /* do the count */
while (temp != NULL) {
   if ((temp->i >= 1) && (temp->i <= 5))
      ++counts[temp->i - 1];
   temp = temp->next;
   }
for (i = 0; i < 5; ++i)               /* print the results */
   printf("%8d%8d\n", i+1, counts[i]);
}


void main (void)

{
GetList();                            /* read a list */
CountList();                          /* count the items in the list */
}
```

Solution to problem 8.5.

```
/* This program reads in a list of integers, placing them in a  */
/* queue, then prints the numbers in the resulting linked list. */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

typedef struct listElement {
   struct listElement *next;
   int i;
   } listElement;

listElement *list;
```

```
void GetList (void)

/* Read a list from the keyboard.                                    */
/*                                                                   */
/* Variables:                                                        */
/*    list - pointer to the head of the list                         */

{
int i;                                /* variable read from the keyboard */
listElement *temp;                       /* work pointer */
listElement *last;                       /* ptr to the end of the list */

list = NULL;                             /* initialize the list pointer */
last = NULL;                             /* no end item, yet */

do {
   scanf(" %d", &i);                     /* read a value */
   if (i != 0) {                      /* if not at the end of the list... */
                                         /* allocate a structure */
      temp = (listElement *) malloc (sizeof(listElement));
      temp->i = i;                       /* record the value */
      temp->next = NULL;                 /* nothing comes after it */
      if (list == NULL)                  /* add the struct to the list */
         list = temp;
      else
         last->next = temp;
      last = temp;                       /* record the new end of list */
      }
   }
while (i != 0);
}


void PrintList (void)

/* Print a list.                                                     */
/*                                                                   */
/* Parameters:                                                       */
/*    list - pointer to the head of the list                         */

{
listElement *temp;                       /* work pointer */

while (list != NULL) {
   temp = list;                          /* remove an item from the list */
   list = temp->next;
   printf("%d\n", temp->i);              /* write the value */
   free(temp);                           /* free the memory */
   }
}
```

```
void main (void)

{
GetList();                              /* read a list */
PrintList();                            /* print a list */
}
```

Solution to problem 8.6.

```
#include <stdio.h>

void main (void)

{
int i, j, *ip;

j = 4;
ip = &j;
i = *ip;
printf("%d\n", i);
}
```

Solution to problem 8.7.

```
#include <stdio.h>

void main (void)

{
int a[3], *ip;

a[0] = 1;
a[1] = 2;
a[2] = 3;
ip = a;
printf("%d\n%d\n%d\n", ip[0], ip[1], ip[2]);
}
```

Solution to problem 8.8.

```
/* Test strcmp2 */

#include <ctype.h>
#include <stdio.h>
#include <string.h>

int strcmp2 (char *p1, char *p2)

{
while (*p1 && *p2)
   {
   if (toupper(*p1) !=
       toupper(*p2)) {
      if (toupper(*p1) <
          toupper(*p2))
         return -1;
      return 1;
      }
   ++p1;
   ++p2;
   }
if (*p1)
   return 1;
if (*p2)
   return -1;
return 0;
}


void main (void)

{
char string1[20], *cp;

strcpy(string1, "HO");
cp = string1;

printf("%d %d\n",  1, strcmp2("ho", "HI"));
printf("%d %d\n", -1, strcmp2("hi", string1));
printf("%d %d\n",  0, strcmp2("ho", cp));
printf("%d %d\n",  1, strcmp2("hoh", "HO"));
printf("%d %d\n", -1, strcmp2("ho", "HOH"));
}
```

Lesson Nine
Files

The Nature of Files in C

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/C itself – all of these programs read and write files. This lesson introduces files as used in the C language.

C has an enormous variety of different ways to deal with disk files, all of which are accessed through functions defined in the stdio.h header file you already use for reading the keyboard and writing information to the shell window. The functions you will learn are actually designed to work with a wide variety of output devices. For example, the fprintf function works almost exactly like the printf function you are used to, but it writes to a file. With the proper device drivers installed in the operating system, the fprintf function could also be used to write to modems, printers, the shell window, or even tape drives. There is also an fscanf function that can be used to read from files. In other words, one of the methods for accessing files is going to look very similar to the way you access the keyboard and shell window. In fact, C can even treat the keyboard and shell window as a file.

Some of the functions in stdio.h will seem a little dated on the Apple IIGS. The rewind function is a good example: it was included in the library from the early days of C programming, when the most common form of mass storage was the tape drive, not the floppy disk. With all of these functions lying around, stdio.h is a very large (and very capable) library. We won't even try to cover all of these features of stdio.h; instead, we will concentrate on a few of the basic functions that are useful for general purpose file input and output. As you read this lesson, though, I would recommend that you keep the ORCA/C reference manual handy, and refer to it often to learn about functions that we are not covering, and to read about the many features that we won't cover in some of the functions we do use.

What is a File?

The way we normally think of files, a file is basically a collection of information stored somewhere. For floppy disks and hard disks, which is what we will deal with in this lesson, that definition fits very well. As I mentioned a moment ago, though, the file access functions in C use a looser definition of the term. With this looser definition, a file becomes a place that you can send information to, or a place that you can get information from. You can't always read what you write, nor can you always back up and read something a second time. In fact, some files can be read, but not written to, and other can be written to, but not read. Two good examples of this are stdin and stdout, two files that you have been using all along without realizing it. The file stdin is the standard input device – in our case, the keyboard. You can read the keyboard, but you can't back up and reread a line if you missed something. Of course, you can't write to the keyboard; that just doesn't make any sense. stdout is the standard output device. If you are using the PRIZM desktop development environment, the standard output device is the shell window; from the text development environment, the standard output device is the text screen. Once again, it is easy to see why you can write to the stdout file, but you can't read from it.

The Four Basic Operations

As you deal with files, there are basically four operations that you will learn. There are a lot of different functions defined in stdio.h, but most of them do one of these four basic operations. Before you can read from a file or write to a file, you must open it. Opening a file, then, is the first of the four basic operations. Opening a file is a lot like opening a file folder in a file cabinet: it tells C to find the file, creating a new one in some circumstances, and to get the file ready for reading, writing, or both. Once you open a file, you will either read from it or write to it; these are the second and third of the basic operations. Finally, when you are finished with a file, you need to close it. This cleans up, taking care of any operating system dependent tasks needed to get the file ready for the next program that will open the file.

File Variables

The four basic operations are scattered across a number of different function calls. In our first few examples, we will use fopen to open a file, fprintf and fscanf to write to and read from a file, and fclose to close the file when we are finished with it. C can, of course, deal with more than one open file at a time, so we need a way to tell all of these functions exactly what file we want the function to work on. This is done with a file variable, defined like this:

```
FILE *f;
```

The type FILE is defined in stdio.h; file variables are always a pointer to this type. There is nothing special about the variable name. You pick it just like you would for any other variable.

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 printer, a tape drive, or anything else that GS/OS recognizes.

To write a value to a file, you need to open the file for output. This is done with the fopen function. For a file variable f, the fopen call looks like this:

```
f = fopen("myfile", "w");
```

The two parameters to fopen are the file name (myfile in this case) and a flags string that tells fopen what the characteristics are for the file (in this case, "w" tells fopen to open the file for output). As we go on through the lesson, you will see more and more options for the flags. You can also find a complete description in the ORCA/C reference manual, listed under the description of fopen. The fopen function returns a pointer to a file variable it creates, which, of course, you need to store in the file variable f for the file calls you will make to write to the file and close the file.

To write to the file, you use the function fprintf. The fprintf function is almost identical to printf; the only difference is that you pass a new parameter right before the format string. The new parameter, of course, is the file variable.

```
fprintf(f, "%d\n", i);
```

Once you have written everything you need to write, you close the file using fclose. The fclose function takes a single parameter, the file variable to close. Once you close the file you cannot write to it with fprintf.

```
fclose(f);
```

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.                   */

#include <stdio.h>

void main(void)

{
FILE *f;
int i;

f = fopen("myfile", "w");
for (i = 1; i <= 10; ++i)
   fprintf(f, "%d\n", i);
fclose(f);
}
```

Type this program in and run it. The file the program creates is a standard text file; there is nothing special about it at all. You can use the Open command from the File menu to open the file, just as you would open any text file. When you do that, you will see the numbers 1 to 10, on separate lines, just like you would if you ran the same program with printf to write the values to the shell window.

Problem 9.1. The fprintf function has all of the formatting abilities you have learned for printf. Use this fact to create a set of multiplication tables showing values up to 10x10. The numbers 1 to 10 should appear across the top of the table, while 1 to 10 also label the rows. Use the _ and | characters to put dividing lines along the top and left sides of the table, as shown in the sample output. The results should be saved in a file called multable.

```
      1   2   3   4   5   6   7   8   9  10
    ------------------------------------------
 1 |  1   2   3   4   5   6   7   8   9  10
 2 |  2   4   6   8  10  12  14  16  18  20
 3 |  3   6   9  12  15  18  21  24  27  30
 4 |  4   8  12  16  20  24  28  32  36  40
 5 |  5  10  15  20  25  30  35  40  45  50
 6 |  6  12  18  24  30  36  42  48  54  60
 7 |  7  14  21  28  35  42  49  56  63  70
 8 |  8  16  24  32  40  48  56  64  72  80
 9 |  9  18  27  36  45  54  63  72  81  90
10 | 10  20  30  40  50  60  70  80  90 100
```

Reading from a File

Reading a file is just as easy. To read from a file, you first have to open it for input. You do this with the fopen function, just as you did when you wrote a file. The only difference is that you use a flags string of "r" instead of "w". The following program reads the file of ten integers you created in the last section, writing the values that are read to the shell window. Be sure you have already run the program from the last section to create myfile before you run this program, since we haven't dealt with error checking yet!

```
/* Read some numbers from a */
/* file.                    */

#include <stdio.h>

void main(void)

{
FILE *f;
int i, num;

f = fopen("myfile", "r");
do {
   num = fscanf(f, "%d", &i);
   if (num == 1)
      printf("%d\n", i);
   }
while (num != EOF);
fclose(f);
}
```

When we wrote the file, we knew exactly how many numbers we were going to write. We could write this program to read in the same number of numbers, but then we would have to change both programs to read a file of a different size. In addition, in many real-world programming situations, you don't know in advance how big a file is, so instead of relying on a fixed files size, this program adapts automatically to the size of the file. It does this by using the value returned by fscanf. When you use fscanf, it returns the number of things it reads and formats – in our example, this would normally be 1, since fscanf is trying to read one integer. If fscanf runs into some other character, like a q, it will return 0, since it could not read an integer. If fscanf tries to read a number, though, and sees that there are no more characters in the file at all, it returns a value that is equal to the constant EOF; this constant is defined in stdio.h. By checking the value returned by fscanf to see if it is EOF, the program can skip printing the value, then drop out of the do loop.

Problem 9.2. The fscanf function has all of the formatting abilities you have learned for scanf. Use this fact to write a program that reads strings from a file, rather than integers. As with the sample program, you should echo the strings to the shell window.

The file name is just a string, so you can read it in just like any other string. In this program, ask the user for the file name.

Try your program on two files, the file of integers you created in the last section, and the source for your solution to this problem. The result should tell you a lot about how fscanf breaks characters up into strings.

Problem 9.3. At this point, you have the tools to merge two files. The basic method is simple: you open one file for input and another for output. You read values from one file, writing them to the other, until you get to the end of the first file.

Start by writing 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, 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.

The portion of the program that opens a file for input, reads the file, writes to the output file, and closes the input file should be encapsulated in a function that takes the name of a file as its only input. The output file variable should be a global

variable. Open the output file once, before calling the function that will copy the contents of a file to the output file, and close the output file once just before the program stops.

Check your work by editing file3. It should contain the numbers 1 to 20.

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.

File Names

The file names we have been using for our C programs have been pretty simple. As with any name, there are some rules you must follow in choosing a file name. There is a problem, though: as you know, C is used on many different computers, and unfortunately, the way you pick a file name differs from computer to computer. If you are trying to write programs that will run on several different computers, this is something you will have to keep in mind. On the Apple IIGS, 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.

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 in a C program, you use exactly the same name, but you start it off with a slash character. For example, the disk where the C compiler is located is called ORCA.C. In a file name, you would type

```
/ORCA.C
```

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.C/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.CC 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.C/SAMPLES/BULLSEYE.CC
```

The result is called a full path name. It specifies exactly what file you want to read or write.

In our examples, 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, 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 files created by the sample programs 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.C disk, and you want to access the BULLSYS.CC file, you can use

```
SAMPLES/BULLSEYE.CC
```

The process of forming names for the fopen 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.C, for example, you would use

```
prefix /orca.c
```

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.c/samples/bullseye.cc
```

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 9.4. Move to the shell window, and use the PREFIX command to set the current prefix to /ORCA.C/SAMPLES. Verify that you did it right by using this command to open the BULLSEYE.CC program:

```
edit bullseye.cc
```

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 ORCALIB and SYSLIB, and a folder called ORCACDEFS. Close the BULLSEYE.CC window, and, without changing the default prefix, edit the file again.

Reading Text Files One Line at a Time

You already know the basics of reading and writing files, but there are a lot of variations on the basic theme. In the rest of this lesson, we will look at a variety of different programming situations that use files or formatted input and output, learning how to process information a little more effectively. The first thing we need to cover is a way to read a file a line at a time.

Of course, you have already learned how to read strings with scanf, and the same idea will work with fscanf, but when you use the %s conversion specifier, the scan functions treat a string as a blank-delimited string, not as a line of text. For example, if you use scanf or fscanf to read the line

```
How, now, brown cow.
```

they will return four strings, not one. (The four strings will be "How,", "now,", "brown" and "cow.".)

To get around this problem, we need a new and very peculiar conversion specifier, []. The bracket conversion specifier tells C to read characters as long as the characters being read match one of the characters inside of the brackets. For example, we could read the entire line above with this scanf call:

```
scanf("%[How, nbrc.]", string);
```

where string is an array of char. scanf will read characters until it gets to the \n that marks the end of the line. (The \n character marks the end of input lines in C, just as you write an \n character at the end of a line with printf.)

Of course, this is pretty useless for our purposes, since you have no idea what characters might actually appear in a given input line. We can solve this trifling difficulty by using the ^ character right after the [character, which tells scanf to read all characters *except* those that follow the ^ character. The new call looks like this:

```
scanf("%[^\n]", string);
```

Only one difficulty remains. This call to scanf reads up to the \n character, but it does not read the \n character itself. If you are reading more then one line,

you will need to skip the \n character so that the next call to scanf will start with the first character of the next line. The completed call to scanf looks like this:

```
scanf("%[^\n]%*1[\n]", string);
```

Some of that new conversion specifier you can pick up on right away from what you have learned so far. A conversion specifier of %[\n] would scan for a \n character, but by itself, this presents a problem, since the file might have several blank lines after the one you are reading, and you don't want to skip them. Putting a 1 in the conversion specifier, so that you have %1[\n], tells scanf to read at most 1 character. The * is a store suppressor. In this case, you know you will read a \n character, and you don't want to waste the time and space to create a dummy char variable and store the \n in the variable. By placing the * after the %, you are telling scanf to read the \n from the file, but *not* to store it anywhere.

Problem 9.5. Write a simple program to echo a file to the screen. Your program should start by asking for a file name, then use the name it reads as the name supplied to fopen. Read the file one line at a time using fscanf and the format string you just learned about, writing the lines using printf.

Problem 9.6. Write a program that asks for a file name. The program should open the file for input, read lines from the file, convert the characters to uppercase, and write them to the shell window. Be sure and test your program! One way to do this is to create an uppercase only version of the program itself.

Bullet-proof Input

What's wrong with this program?

```
#include <stdio.h>

void main(void)

{
char string[81];

printf("String:");
scanf("%[^\n]%*1[\n]", string);
printf("%s\n", string);
}
```

If you don't see the problem, don't feel too bad. Most of the time, the program works just fine. What happens, though, if some user types in a line that is 90 characters long? (Ever notice how user is a four-letter word for most programmers? Frustration, I guess.) The answer is that, in all likelihood, the program will crash, since the string array is only large enough to hold 80 characters in the string. We need some way to prevent crashes like this, and the way to do it is with a maximum length field. Just as we use a 1 to tell scanf to read at most one line feed, we can also use a value to tell scanf the maximum number of characters we want to read and place in the array, like this:

```
scanf("%80[^\n]%*1[\n]", string);
```

This idea is the basis for bullet-proof input in C. By limiting the number of characters the program will read, you can prevent an array overflow and the sometimes disastrous after-effects. If the user types more characters than allowed, scanf simply stops. Of course, that means you are no longer guaranteed that the next character will be a \n character, so some adjustments need to be made in the way you read lines. One good way to handle this situation is to check to see if you read the maximum number of characters. If so, you can process the characters so far, then go back and read more of the line, without reading the \n character. Another way to handle the situation is to artificially break long lines up into several smaller lines; this might be a good choice for a program that will write the results to the screen, anyway. Finally, you might want to just stop and print an error message.

Another problem with scanning input occurs when you expect one thing, like a number, and get something else, like a character. So far, the solution is to stop the program and start over. That's fine for you when you are learning to program, but in a good program that others will use, you need to handle things a bit better. A popular way to do this is to read the line, as we just did, and use yet another form of scanf to read the characters, called sscanf. Like fscanf, the difference between sscanf and the scanf function you already know about is simply where sscanf takes the characters from. In this case, the characters are read from an input string, which is the first parameter you specify. Naturally, this input string can be the string you just read as a whole line from the keyboard or a file, using one of the other forms of the scanf function.

You need one other piece of information to turn all of this into a genuinely flexible way to read and process text input, and that is a new conversion specifier for scanf, called %n. This conversion specifier is a bit of an oddball, in that no characters are read from the input

stream. Instead, scanf expects a pointer to an integer in the parameter list, and stores the number of characters read so far. This can be used to tell you whether any characters were read at all, and if so, how far you need to skip into the string to find the start of the next integer.

Nothing helps firm up arcane details like the ones we've been discussing like an example, so let's stop now and look at a program that puts all of these ideas together. This program will perform a fairly simple task, but it will do it very, very well. All our program does is read integer values from a file. The thing that distinguishes this program from all of the others we have written in the past is that it can detect and recover from absolutely any kind of error, some of which we haven't even talked about before now.

```
#include <stdio.h>
#include <string.h>

#define BOOLEAN unsigned
#define TRUE 1
#define FALSE 0

int main(void)

{
char string[81];
BOOLEAN done;
int num, i, startch, numch, val;
FILE *f;

/* open the file */
f = fopen("myfile", "r");
if (f == NULL) {
   fprintf(stderr,
      "Could not open myfile\n");
   return -1;
   }

do {
   /* read a line */
   num = fscanf(f,
      "%80[^\n]%*1[\n]", string);
   if (strlen(string) == 80) {
      fprintf(stderr,
         "Line too long.");
      return -1;
      }
   /* process the numbers */
   if (num != EOF) {
      done = FALSE;
      i = 0;
      do {
         numch = 0;
         startch = 0;
         sscanf(&string[i],
            " %n%d%n",
            &startch, &val,
            &numch);
         if (numch <= startch) {
            if (i ==
               strlen(string))
               done = TRUE;
            else {
               fprintf(stderr,
                  "Invalid: %s\n",
                  &string[i]);
               return -1;
               }
            }
         else {
            printf("%d\n", val);
            i += numch;
            }
         }
      while (!done);
      }
   }
while (num != EOF);

if (fclose(f) == EOF) {
   fprintf(stdout,
      "Could not close myfile");
   return -1;
   }
return 0;
}
```

The first difference pops up right away. When you learned to use fopen to open a file for input, I mentioned that you should be sure the file actually existed, since your program was not checking for errors. Well, here's how to check for errors:

```
f = fopen("myfile", "r");
if (f == NULL) {
   fprintf(stderr,
      "Could not open myfile\n");
   return -1;
   }
```

When the fopen function opens a file, it returns a file pointer; if it can't open the file, it returns NULL. If you want to get fancy, you can even look at the global variable errno to get some idea as to why the file could not be opened, but we won't go into exactly how this is done in the course. If you would like to explore this idea, read about errno and perror in the ORCA/C reference manual.

Assuming an error is found, an error message is printed. That's no surprise, but using fprintf with the file name stderr presents something new. In C, there are two ways to write to the shell window: you can write to stdout (as printf does, by default) or you can write to stderr. The reason for the second output file is because you can redirect the output from a text program when you run the program from the shell window. Redirected output is a very powerful idea; it lets you trap any characters that would have been written to the shell window in another file without making any change to the program. The problem is that you want to see any error messages printed by the program, so C gives you a second way to print text by writing it to stderr. Even when output is redirected, the characters written to stderr still show up in the shell window. Even if you don't use the text shell, and don't quite grasp why all of this is important, you should get in the habit of writing error messages to stderr.

If an error occurs, we also want to quit. We do that by returning right away from the program, passing back a value of -1. It turns out that main is allowed to return an integer, and in text programs, it is a good idea to do that. When your program finishes normally, you should return a zero; if it fails, you should return some non-zero value, generally -1. The reason for this is again tied up in the way the text shell works, and again, you don't really have to worry about it unless you will be writing new commands and utilities for the shell. As with stderr, the important point is to remember that you can return an error code from main, in case you start writing utilities later.

The heart of the program is the do loop that actually processes the lines once the file is open. This do loop starts off by reading a line using the techniques we discussed right before the program. In this example, we use an extra character to detect lines that are too long, and if we find one, we again quit with an error.

Once a line is read, another do loop scans the lines for numbers. There are a lot of things that can go wrong, so the logic to handle all of the various cases is pretty involved. Here's the loop that reads the numbers from the line:

```
done = FALSE;
i = 0;
do {
   numch = 0;
   startch = 0;
   sscanf(&string[i],
      " %n%d%n",
      &startch, &val, &numch);
   if (numch <= startch) {
      if (i == strlen(string))
         done = TRUE;
      else {
         fprintf(stderr,
            "Invalid: %s\n",
            &string[i]);
         return -1;
         }
      }
   else {
      printf("%d\n", val);
      i += numch;
      }
   }
while (!done);
```

The idea behind this loop is to scan a text line that is supposed to contain only numbers. The program will work if the line is blank, but it will stop with an error if anything except a number appears on a line. To handle this situation, we use sscanf to scan the string, along with the new conversion specifier that was mentioned a while back, %n. Instead, %d writes the number of characters that have been read so far to an integer. The lines

```
sscanf(&string[i],
   " %n%d%n",
   &startch, &val, &numch);
```

do the bulk of the work. The initial whitespace tells scanf to skip over any whitespace that comes before the number. We then use %d to store the starting index of the characters that we will read in startch; read the number, placing it in val; and record the number of characters read in numch. If the line contained a numeric value, numch would be larger than startch, and

the if condition right after the sscanf call would test false, so we would drop down to the else clause, print the number, remember where we need to start to scan for the next number, and loop. If numch and startch are the same, we have skipped some whitespace and arrived at the end of the line.

This explains everything except the lines

```
numch = 0;
startch = 0;
```

and the fact that the if statement checks to see if numch is *less than* or equal to startch, not simply equal to. The reason is tied up in the fact that sscanf stops scanning if it hits an error or the end of the line. If sscanf hits the end of the line right away, as would happen if we read an integer, then looped, but found nothing else in the line, then startch, val and numch will not be set at all. In this situation, the program recovers nicely, since numch and startch are preset to 0, and therefore test as equal. If startch is read, though, and we the hit a non-numeric character, like a q, numch will not be set at all. In that case, startch will be greater than numch, but we have not read a number.

Once we reach the end of the line, the if condition will test true, and we drop through for one final check: we make sure that we reached the end of the line, and didn't just hit a bad character.

The last section closes the file, again checking for errors. Assuming there are no errors, the program returns 0, which tells the shell that no errors were found.

As a footnote, I would like to point out that there is an equivalent to sscanf for the printing family of functions, called sprintf, which you can use to format text just as you do with printf, but place the text in a string instead of writing it to a file. While we won't use this function in the course, it is often useful in toolbox programming, where you might want to format some information before using one of the toolbox calls to write the string to a graphics window or use the formatted string in a dialog.

Problem 9.7. The program you just looked at is fairly short, but it is very involved. To make sure you understand exactly what it does, step through the program by hand with the following input. To step through the program, actually write down the names of the variables on a sheet of paper, and "play computer," stepping through line-by-line keeping track of what the variables are. This will take some time, but you will see very clearly how the program works.

If you have problems, you can call on the most patient helper in the world: your computer. The debugger can be used to step through the actual program.

```
1
2 3
  4
5  stop
6
```

Since this is a pen and paper exercise, there is no solution given in the solutions. You will know if you did it right!

Binary Files

So far, we have been creating text files that could be edited with an editor. This is a good way to get familiar with file handling in C, since you already know how to use printf and scanf. It also has the advantage that you can edit the file you create to see exactly what your program did. Finally, in many programs, text files are also what you want to create.

On the other hand, not all files on your computer are text files, and there are some very good reasons for this. If you stop and think about it, you already know one of the biggest reasons: space. After all, when you use fprintf to write a floating-point number, you could get something like

```
3.1416e00
```

This number is 9 characters long, with each character taking up one byte of space in the file. Variables of type float, though, only take up four bytes. The first reason to use a binary file instead of a text file, then, is that binary files often take up less room on disk.

Something you may not know is that converting a number from the internal format used by the computer to characters, or converting from characters to the internal format, takes a lot of time. There are about ten to twenty floating-point operations, as well as several integer operations, involved in converting a floating-point number from one format to the other. If you write the four bytes that are used internally to represent a floating-point number, though, no conversion is involved at all. The same argument holds for integers, although the difference in speed is not as great as it is with floating-point numbers. Speed is the second reason to use binary files instead of text files.

The last reason to use a binary file is convenience. You already know that you can't use printf to write some kinds of data, like structures or arrays (other than

string, of course). Using binary files, you can write structures or arrays directly to a file, and read them back from a file.

To investigate binary file, we'll use two short sample programs, one of which writes a file containing 10 float values, while the other reads the same file and echoes it to the shell window.

```
/* create a binary file */

#include <stdio.h>

void main(void)

{
float x;
FILE *f;

f = fopen("myfile", "wb");
for (x = 1.0; x <= 10.0; ++x)
   fwrite(&x, sizeof(x), 1, f);
fclose(f);
}


/* read a binary file */

#include <stdio.h>

void main(void)

{
float x;
FILE *f;
int num;

f = fopen("myfile", "rb");
do {
   num = fread(&x, sizeof(x), 1,
      f);
   if (num)
      printf("%f\n", x);
   }
while (num);
fclose(f);
}
```

When you create a file with fopen, tacking a b onto the w in the flags string tells fopen that you want to create a binary file, rather than a text file. The resulting file has a file type of BIN, and, of course, it cannot be loaded into the editor like a text file. When you open the file to read it in the second program, you can also put the b in the string, but fopen actually ignores the character in this case. You can open any file type you like for input.

The biggest difference between the way we handle text files and binary files, though, is the way we read and write information. Instead of fprintf, which formats information, creating strings, we use fwrite. The fwrite function writes a series of bytes directly from memory to the disk file. The first parameter is the address of the first of the bytes to write; in our case, we pass the address of the floating-point value we are writing to disk. The next parameter is the number of bytes to write; using sizeof is the best way to find out what this value is. The third parameter is the number of things to write. If you were writing an array of ten float values, you could still pass the size of a single float value, and then pass 10 as the second parameter. For writing an entire array, it might make just as much sense to pass the size of the array, and then pass 1 (which does the same thing), but this feature does come in handy when you want to write *part* of an array, instead of the whole thing. Finally, you pass the file variable for the file you want to write the bytes to.

The second program shows that fread works pretty much the same way. It uses the same parameters, in the same order, to do the same thing, but fread reads values from an open file instead of writing values to the file. Making all of the parameters the same is a great aid in remembering the functions.

Problem 9.8. Create a binary file with an array of 12 integer values, 1 to 12. In this program, fill in an array first, then write the array with a single fwrite call.

Write another program to read the same file. In the second program, use an array of integer values that can hold 5 integers at a time, and use a single fread statement to read in up to 5 integers. Keep track of the value returned by fread to decide how many elements of the array to write and when to drop out of the read loop.

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, you really want to open the file for both input and output, and you want to be able to skip around in the file. That way, you can skip to a particular entry, read the entry, and write the change back to the disk.

So far, we have always used the fopen function to open a file for either input or output, but you can also open a file for both. To open a file for input and output, you use a flag of a, rather than r or w. Of course, you can still use this flag in conjunction with b to open a binary file instead of a text file.

```
f = fopen("myfile", "ab");
```

When you open a file for input and output, fopen will open an existing file if there is one, setting up things so you will read the first value from the file, or write over the first value if you start writing right away. If the file doesn't already exist, a new one is created.

To skip around in the file, you use the fseek function. To skip to the fourth entry in a file with integer values, you would use:

```
fseek(f, sizeof(int)*3, 0);
```

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.

```
#include <stdio.h>

void main(void)

{
FILE *f;
int i;

f = fopen("numbers", "ab");
if (f == NULL) return;
fseek(f, sizeof(int)*3, 0);
i = 4;
fwrite(&i, sizeof(int), 1, f);
fclose(f);
}
```

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.

```
/* Generate 10x10 multiplication tables */

#include <stdio.h>

void main(void)

{
FILE *f;
int i,j;

f = fopen("multable", "w");
fprintf(f, "        1   2   3   4   5   6   7   8   9  10\n"
           "    ____________________________________________\n");
for (i = 1; i <= 10; ++i) {
   fprintf(f, "%3d |", i);
   for (j = 1; j <= 10; ++j)
      fprintf(f, "%4d", i*j);
   fprintf(f, "\n");
   }
fclose(f);
}
```

Solution to problem 9.2.

```
/* Read a file and break it into strings */

#include <stdio.h>

void main(void)

{
FILE *f;                                /* file variable */
char str[100];                          /* file name and input string */
int num;                                /* value returned by fscanf */

/* get a file name and open the file */
printf("File name:");
scanf("%s", str);
f = fopen(str, "r");

/* read strings from the file, echoing to the shell window */
do {
   num = fscanf(f, "%s", str);
```

```
   if (num == 1)
      printf("%s\n", str);
   }
while (num != EOF);

/* close the file */
fclose(f);
}
```

Solution to problem 9.3.

Program 1: create file1.

```
/* Write 1 to 10 to file1 */

#include <stdio.h>

void main(void)

{
FILE *f;
int i;

f = fopen("file1", "w");
for (i = 1; i <= 10; ++i)
   fprintf(f, "%d\n", i);
fclose(f);
}
```

Program 2: create file2.

```
/* Write 11 to 20 to file2 */

#include <stdio.h>

void main(void)

{
FILE *f;
int i;

f = fopen("file2", "w");
for (i = 11; i <= 20; ++i)
   fprintf(f, "%d\n", i);
fclose(f);
}
```

Program 3: merge the files.

```
/* Merge two files to create a third file */

#include <stdio.h>

FILE *outFile;                          /* output file variable */

/* Add the contents of a file to outFile                */
/*                                                      */
/* Parameters:                                          */
/*    name - input file name                            */
/*                                                      */
/* Variables:                                           */
/*    outFile - output file (opened for output)         */

void AddFile (char name[1])

{
FILE *inFile;                           /* input file variable */
int i;                                  /* value read from the file */
int num;                                /* # items read */

inFile = fopen(name, "r");              /* open the file */
do {                                    /* read & echo the file */
   num = fscanf(inFile, "%d", &i);
   if (num == 1)
      fprintf(outFile, "%d\n", i);
   }
while (num != EOF);
fclose(inFile);                         /* close the file */
}


/* main program */

void main(void)

{
outFile = fopen("file3", "w");          /* open the output file */
AddFile("file1");                      /* copy in the contents of file1 */
AddFile("file2");                      /* copy in the contents of file2 */
fclose(outFile);                        /* close the output file */
}
```

Solution to problem 9.4.

The commands needed are:

```
prefix /orca.c/samples
edit bullseye.cc
prefix /orca.c/libraries
cat
edit /orca.c/samples/bullseye.cc
```

Solution to problem 9.5.

```
/* Echo a file to the screen */

#include <stdio.h>

void main (void)

{
FILE *f;                                /* input file variable */
char line[100];                         /* input string */
int num;                                /* # of inputs */

printf("File name:");                   /* open the file */
scanf("%s", line);
f = fopen(line, "r");

do {
   num = fscanf(f, "%[^\n]%*1[\n]", line); /* read a line */
   if (num != EOF)                      /* print the line */
      printf("%s\n", line);
   }
while (num != EOF);

fclose(f);                              /* close the file */
}
```

Solution to problem 9.6.

```
/* Convert a file to uppercase */

#include <stdio.h>
#include <ctype.h>

void main (void)

{
FILE *f;                                /* input file variable */
char line[100];                         /* input string */
int num;                                /* # of inputs */
int i;                                  /* loop variable */

printf("File name:");                   /* open the file */
scanf("%s", line);
f = fopen(line, "r");

do {
   num = fscanf(f, "%[^\n]%*1[\n]", line); /* read a line */
   for (i = 0; i < strlen(line); ++i)   /* convert to uppercase */
      line[i] = toupper(line[i]);
   if (num != EOF)                      /* print the line */
      printf("%s\n", line);
   }
while (num != EOF);

fclose(f);                              /* close the file */
}
```

Solution to problem 9.8.

Program 1: create the file.

```
#include <stdio.h>

void main(void)

{
int a[12];                              /* array to write */
int i;                                  /* loop variable */
FILE *f;                                /* file variable */

f = fopen("myfile", "wb");              /* open the file */
for (i = 0; i < 12; ++i)                /* fill the array */
   a[i] = i+1;
fwrite(a, sizeof(int), 12, f);          /* write the array */
fclose(f);                              /* close the array */
}
```

Program 2: read the file.

```
#include <stdio.h>

void main(void)

{
int a[5];                                /* array to write */
int i;                                   /* loop variable */
FILE *f;                                 /* file variable */
int num;                                 /* # items read */

f = fopen("myfile", "rb");               /* open the file */
do {
   num = fread(a, sizeof(int), 5, f);    /* read up to 5 values */
   if (num) {
      for (i = 0; i < num; ++i)          /* print the values read */
         printf("%d\n", a[i]);
      }
   }
while (num);
fclose(f);                               /* close the file */
}
```

Solution to problem 9.9.

Program 1: create the test file.

```
#include <stdio.h>

void main(void)

{
FILE *f;                                /* file variable */
int i;                                  /* loop variable */
int a[5];                               /* array of values */

f = fopen("numbers", "wb");             /* open the file */
for (i = 0; i < 5; ++i)                 /* set up the array */
   a[i] = i+1;
a[3] = 3;
fwrite(a, sizeof(int), 5, f);           /* write the array */
fclose(f);                              /* close the file */
}
```

Program 2: read the file.

```
#include <stdio.h>

void main(void)

{
FILE *f;                                /* file variable */
int i;                                  /* loop variable */
int a[5];                               /* array of values */

f = fopen("numbers", "rb");             /* open the file */
fread(a, sizeof(int), 5, f);            /* read the array */
for (i = 0; i < 5; ++i)                 /* write the array */
   printf("%d\n", a[i]);
fclose(f);                              /* close the file */
}
```

Lesson Ten
Miscellaneous Useful Stuff

A Look at this Lesson

As you have probably gathered by now, this course is organized around presenting the principles of programming, not just teaching you the C language. In the first few lessons, the emphasis was on teaching you enough about the C language and the ORCA development environment so you could write simple programs. Gradually the emphasis shifted, so that the last few lessons have concentrated on specific programming goals and ideas that are common in many programs, presenting the parts of the C language that were needed to accomplish these tasks. Starting with Lesson 13, we will shift completely over to learning about programming concepts and techniques. You will have learned as much of the C language as this course will teach you, and the emphasis will instead be on learning how to use the C language you already know to solve common programming problems.

C is a big language, though, and there are many other details about the language that have not been covered in this course. The purpose of this lesson and the next one is to give you a quick tour of some of the parts of the language that have not been used in the course so far. You will see many of these features used in the lessons that follow, but for various reasons, it is a good idea to stop and cover these parts of the C language here, so we can concentrate on the programming later, and not on the mechanics of a few new features of the C language.

Some of the features you will see described in this lesson and the next one are key parts of the C language that you will find used over and over. Initializers, which let you assign an initial value to a variable when it is defined, are a good example. The only reason you haven't seen them so far is that it would have interrupted some of the thought process in the early lessons, when the emphasis was on developing an orderly flow in the programs, and because you need to know more about storage classes (the various ways variables are stored) before you could understand when it is appropriate to use initializers, and when it is a waste of time. Other features of the language are rarely used. The goto statement is a great example of this: there is only one situation in C that I can thing of when the goto statement should be used, and even then, there are ways to avoid it. There are some features that are very useful in some kinds of programs, but which some programmers rarely, if ever, use. Writing numbers in other bases is a good example: for some kinds of toolbox programming, and many bit or byte manipulation programs, the ability to write a number in a new base is very handy, yet for many other kinds of programs, you just don't need the feature. Finally, some of the sections expand on features you have already seen briefly, filling you in on the details. The descriptions of the break statement (which you used to get out of the switch statement) is one good example.

In short, this lesson and the next one are a miscellaneous collection of features of the C language. Some are neat, some you just need to know about in case you run across them in a book or magazine, and some may seem very useful to you, and worthless to someone else taking the course. In any case, this lesson will certainly give you some of the flavor of the C language as it is used by experienced programmers.

Number Bases

Most people who deal with numbers at all are very familiar with what mathematicians call the decimal number system, or base 10. The reason, of course, is right in front of you: you probably have 10 fingers. Computers, on the other hand, have one "finger": everything they do is based on internal switches. As a result, computers do math using what mathematicians call binary arithmetic, where there are only two digits, 0 and 1. When you want to write the number after 9, which is the largest one you can represent with a single digit in base 10, you write a 1 followed by a zero: the 1 is in the tens column, and the zero in the ones column. The same thing happens in base two. To write one, you write 1. One is the largest available digit, so to write two in binary, we write 10.

Binary math quickly becomes tedious, both for us and for the computer. Because of the way computers are built, we rarely deal with individual bits; instead, we group bits together to form bytes. In the very early days of computers, there were two common sizes for a byte: six bits and eight bits. Other sizes have also been tried, but these two have a direct impact on the C language. With a little work, you can convince yourself that you can represent 64 distinct values with 6 bits, and 256 distinct values with 8 bits. We could, of course, develop numbering systems for base 64 and base 256 to deal with this situation, but that gets cumbersome for another reason: you would have to learn either 64 or

256 different symbols to represent a number. In both cases, the way we really deal with the values of a byte is to split the byte up into two pieces, popularly called nibbles. The competing size for the nibbles is 3 bits for a 6 bit byte, which allows up to 8 values to be represented by one nibble; and 4 bits for an 8 bit byte, which allows up to 16 values to be represented by one nibble.

Finally, we get to the point: as you would expect, and as you already know from experience, C lets you code numbers as decimal values. For example,

```
printf("%d\n", 12);
```

prints 12. I expect you would be rather shocked if it printed anything else. Try this program, though:

```
#include <stdio.h>

void main(void)

{
printf("%d\n", 012);
}
```

The only change we made was to put a leading 0 in front of the number, something you probably wouldn't expect to make a difference. In all of the common programming languages except C, it doesn't make a difference, but C will print the value 10.

To understand why, we have to go back to the early days of C, when it was used to develop a fledgling operating system known as UNIX on a small DEC computer. DEC computers happen to print the values of bytes in three bit groups (even 8 bit bytes, strangely enough), and the authors of C undoubtedly wanted an easy way to represent numbers the way they were used to representing them. They arrived at the convention of using a leading zero to tell the computer that a number was in base 8, rather than base 10. In base 8, 012 is 1*8 + 2, just as 12 in base 10 is 1*10 + 2. Adding up the values of the digits, you can see that 012 is, in fact, 10. This peculiar way of representing base 8 numbers is the reason that you were warned back at the start of the course not to start numbers with a zero digit!

Most modern general purpose computers, and virtually all desktop computers, put eight bits in a byte. The natural way to deal with eight bits in a byte is to use base 16, or hexadecimal notation. Of course, you need six extra digits to represent the values 10 to 15, since all of these values become a single digit in hexadecimal notation. For once, everyone arrived at a single solution: the letter A is used to represent 10, B for 11, and so forth. In C, to tell the compiler that you are using hexadecimal notation, you start the number with the characters 0x. The 'x' can be either an uppercase or lowercase letter, and you can use either case for the digits that are represented by a letter, too. I tend to use a lowercase x, and uppercase letters for the digits, just to keep them straight. This convention is common, but certainly not universal.

Here's our sample program again, this time with a hexadecimal value. See if you can figure out the value before running the program. Remember, the rightmost column is the 1's column, the next column to the left will be the 16's column, the next one is the 256's column, and so on.

```
#include <stdio.h>

void main(void)

{
printf("%d\n", 0xBAD);
}
```

The answer is B*256 + A*16 + D, or 11*256 + 10*16 + 13, which is 2989.

When you read a number with scanf using the %d conversion specifier, it expects a decimal value. Scanf has another conversion specifier called %i which can handle octal and hexadecimal values, though. It works just like %d, but, like the C compiler, it treats numbers that start with a 0 as octal numbers, and numbers that start with 0x as hexadecimal numbers. The printf function also has two conversion specifiers, %o and %x, which allow you to print octal or hexadecimal values directly to the screen. Problem 10.1 explores these conversion specifiers.

Problem 10.1. Read about the %o and %x conversion specifiers for printf, and the %i conversion specifier for scanf in your ORCA/C reference manual. Use what you learn to write a simple base conversion program. You program should accept *long integer* values from the keyboard in decimal, octal, or hexadecimal notation, and print the number in all three bases. It should loop until the person using the program enters 0.

Test your program by converting the following numbers to all three bases:

a. 100000
b. 0xBAD
c. 0xBAD0
d. 012
e. 01

f. 010
g. 0100

The Bitwise And Operation

In the last section, we mentioned again that your computer's memory is made up of a series of bytes, with 8 bits grouped together to form each of the bytes. Advanced programmers and people writing programs to deal directly with hardware often need to deal with these bits individually. C has a series of four bit manipulation operators that make this possible. In the next few sections, we'll explore these bit manipulation operators.

Bit manipulation is a fairly advanced topic for a beginning programming course. If you have already learned how to manipulate bits in another language, these sections may seem a bit slow to you, but if you have never seen bit manipulation before, these sections will be challenging. While you should certainly take a crack at the information you find here, you won't use the bit manipulation operations much in the rest of the course. If some of the information doesn't sink in right away, you won't be lost in the other lessons.

Bit manipulation operators perform the same logical operations you already know, plus one new one. You already know the && and || operators, used to test two logical values to see if they are both true (the and operator, &&) or to see if either is true (the or operator, ||). The first two of the bit manipulation operators correspond directly to these two logical operators. The operators are & and |; they are called the bitwise and operator and the bitwise or operator.

When you use the && operator, it tests two integers, returning 1 if both of the integers are true, and 0 if either or both of the integers are false. The bitwise and operator does something similar, but it does it with the individual bits in the number. To see how this works, we'll try a few examples with four-bit values, specified in binary. In the last section, you saw how to convert from one number base to another, but this may be new to you, so the values are also shown in hexadecimal and decimal notation, too.

For the first example, we'll look at how the & operator works when the numbers are either 0000 or 0001.

	binary	decimal	hexadecimal
	0000	0	0
&	0000	0	0
	0000	0	0
	0000	0	0
&	0001	1	1
	0000	0	0
	0001	1	1
&	0000	0	0
	0000	0	0
	0001	1	1
&	0001	1	1
	0001	1	1

In this example, the & operator does exactly the same thing as the && operator, but in reality, it is only working on a single bit. This is a very handy table, though, because it shows exactly how the & operator works: in any given bit position, if both bits are 1, the bit in the answer is also 1. If either bit is zero, though, the answer is 0, too.

The same ideas that you saw in that example apply to the other bit positions, as these examples show.

	binary	decimal	hexadecimal
	0011	3	3
&	1010	10	A
	0010	2	2
	1100	12	C
&	1010	10	A
	1000	8	8

The four bits we have been working with so far represent a single hexadecimal digit, which is one-half of a byte. An int variable is made up of two bytes in ORCA/C, and it must be at least two bytes long in all other ANSI implementations of C. In other words, an int (or an unsigned int, which is the same size) is made up of four hexadecimal digits. Here's a short program that will extend the operations we were just doing to a full four hexadecimal digits.

```
#include <stdio.h>

void main(void)

{
int a,b;

a = 0x3C3C;
b = 0xAAAA;
printf("%04X & %04X = %04X\n",
   a, b, a&b);
}
```

When you run this program, the result is 2828. Take a close look at the last example, and match up the digits in the numbers used in the program. Do you see a pattern? You can find the result by hand by anding the hexadecimal digits, using the results from anding individual digits in the example.

The & operator can also be used with char variables, which are really just another form of int variable as far as the & operator is concerned, or with long or unsigned long variables, which work the same way but give you eight digits to play with instead of four. The & operator cannot be used with float variables. The same thing is true for all of the other bitwise operators we will explore.

Problem 10.2. In the sample program, the conversion specifier used was %04X. What do the 0 and 4 do? (Hint: each of these digits has a special meaning. You can either experiment with the program to find out what they do, or look up the printf function in the ORCA/C reference manual.)

Problem 10.3. The & operator is often used as a mask to detect the presence of individual bits. You can use this idea to write a function that writes numbers in binary format.

Your function should take an integer parameter and print the integer as a binary number, printing all 16 bits. Making use of the fact that C treats any non-zero value as true, you can write one digit like this:

```
if (num & 0x8000)
   printf("1");
else
   printf("0");
```

Use this function to create a table showing the values 0 to 15 in binary, hexadecimal and decimal notation. This table will be very handy in the sections that deal with bit manipulation!

Problem 10.4. Testing individual bits can be used for a number of programming tricks. One of these tricks is a quick test to see if a number is negative or positive. The piece of information you need to make this work is that negative numbers always have the most significant bit set, so anding the number with 0x8000 will always yield a non-zero number for negative numbers, and 0 for positive numbers.

Use this trick to implement a function that returns true if a number is negative, and false if it is positive. Test your program with inputs of 1, 0, -1, -32767 and 32767.

While mathematicians treat 0 as a number that is neither negative or positive, programmers treat 0 as a positive number.

This trick depends on a particular internal number representation called two's complement notation. Most modern digital computers use two's complement notation, but technically, it is possible that this programming trick will not work with some implementations of C. It is also possible that int values will be larger than two bytes, which would mean that the value of 0x8000 would have to be changed on some machines for this trick to work.

The Bitwise Or Operator

Just as the & operator is a bit-by-bit implementation of the && operator, the | bitwise or operator is a bitwise implementation of the || logical or operator. The | operator works across the bit, setting each bit in the answer to 1 if either of the corresponding bits in the operands were 1, and setting the bit in the answer to 0 if both of the corresponding bits in the operands were 0. To see how this works, let's take another look at the same example we used in the last section to explore the & operator. This also gives you a good way to compare the two operators.

```
    binary      decimal     hexadecimal

    0011         3          3
|   1010        10          A
    1011        11          B

    1100        12          C
|   1010        10          A
    1110        14          E
```

You saw one of the most common uses of the & operator in the problems from the last section, namely to detect if certain bits are set. One of the main uses of the bitwise or operator is to set a bit. The problems will look at one of the many practical uses of this idea.

Problem 10.5. In the last section, we used a small program to test the & operator with the operands 0x3C3C and 0xAAAA. Rewrite this program so it reads two hexadecimal values and prints the result of anding the values with the bitwise and operator, and oring the values with the bitwise or operator. Test the program with several values, including 0x3C3C and 0xAAAA.

Problem 10.6. If you look closely at the ASCII character set, you will find that the values of each of the lowercase alphabetic characters is exactly 32 greater than the value of the corresponding uppercase character. For example, the value of 'A' is 65, and the value of 'a' is 77, which is 32 greater than 65. As it turns out, 32 in hexadecimal is 0x0020, which has exactly one bit set. In other words, you can convert from uppercase to lowercase by setting the sixth bit, counting from the right, and you can do that by oring the lowercase character with the value 0x0020.

Use this idea to write your own version of the tolower function. Your function should be called lower; it should take a char parameter and return a char result. If the input is an uppercase letter, your function should return the lowercase equivalent. If the input is not an uppercase letter, your function should return the same value it was passed.

Test your function by passing the following values, and printing both the input and output:

```
A    a    U    4    {
```

The Bitwise Exclusive Or Operator

The bitwise exclusive or operator is ^. There is no corresponding logical operator for the bitwise exclusive or operator, which returns 0 if both input bits are the same, and 1 if the bits are different, as shown in this example:

```
    binary      decimal     hexadecimal

    0011         3          3
^   1010        10          A
    1001         9          9

    1100        12          C
^   1010        10          A
    0110         6          6
```

This operator is used to change individual bits within an integer. That's sort of an odd thing to do in most programming situations, but it turns out that there are some interesting applications for this operator. One use you can put it to is a simple checksum that detects when a file has been changed. Listing 10.1 shows a program that performs a checksum on a text file.

The program works by flipping bits in a seemingly random manner. The result of running the program on a single file is always the same, though. If you change even one byte in the file, the result will be different. While it is possible to change a file around and get the same checksum, it is fairly unlikely, especially if the files are close. To see the program in action, try running it on a test file that you have saved to disk, then change a single character in the file and run the program again. Be sure you save the file to disk before you run the program, since your program can't read the contents of a window directly.

This sort of checksum is far from foolproof, but it does have some useful applications. One is to check to see if files in memory have been stomped on. Another is to see if you have typed a file into memory correctly: if you have a checksum for the original file, you can generate a checksum for your file and see if they differ.

Problem 10.7. Write a program that reads two integers, computes the exclusive or, and writes the result. Try this program with the inputs 0x1234 and 0xFFFF. Try it again with the result of the first run of the program and 0xFFFF again. What is the result? Why?

Problem 10.8. One of the more interesting applications I have ever come across for the exclusive or operator is to write a file encryption program. The

idea behind a file encryption program is to encode a file so that no one can read it. The file is encrypted with the aid of a password, which is also needed to read the file again: without the password, the information in the file is useless.

An easy way to encrypt a file is to use a random number generator like the one you have used in C simulations to generate a string of random numbers, and exclusive or the random numbers with the bytes in a file. A pseudo-random number generator produces the same sequence of numbers if it is given the same seed, so the password is used as the seed for the random number generator. To encrypt the file, the bytes are read from the input file and exclusive ored with the values generated by the random number generator, then written to the output file.

Decrypting the file is surprisingly easy: you just run the same program again, using the same password. The reason this works is that exclusive oring one value with another value two times gives you back the original number, whatever it was, as you saw in problem 10.7.

Listing 10.1

```
/* compute a checksum for a file */

#include <stdio.h>

int main(void)

{
FILE *f;                                /* input file variable */
char name[100];                         /* input string */
int checkSum;                           /* checksum value */
int byte;                               /* value read from the file */

printf("File name:");                   /* open the file */
scanf("%99s", name);
f = fopen(name, "r");
if (f == NULL) {
   fprintf(stderr, "Could not open %s.\n", name);
   return -1;
   }

checkSum = 0;                             /* start with a zero checksum
*/
do {                                       /* XOR all bytes in the file
*/
   byte = fgetc(f);
   if (byte != EOF)
      checkSum = checkSum ^ byte;
   }
while (byte != EOF);
printf("The checksum is %d.\n", checkSum); /* print the checksum */

fclose(f);                              /* close the file */
return 0;
}
```

Write a program that will ask for an input file name, an output file name, and a numeric password. Use the password as the seed to C's random number generator, then encrypt the input file, writing the bytes to the output file. When you test your program, the editor will give you some sort of error when you try to load the file, but you can move to the shell window and type the file with the type command, like this:

```
type myfile
```

where myfile is the name of the encrypted file. Running the encryption program again with a different password should give you back a garbage file, but you should be able to decrypt the file by using the original password.

Hint: Even though the files involved are text files, you don't want to use fscanf and fprintf, since these text input and output functions do some conversions to the values they read and write. Instead, use fread and fwrite, which do not do any conversions of values. To make the program faster, you can also read more than one character at a time with fread – the solution reads 64 characters at a time.

This encryption routine is good enough to stump most people, but someone who is good at breaking codes would not have much trouble with it. In other words, if you want to protect some private letters from you boss/spouse/parents, go for it, but if you are trying to hide something from the National Security Agency, this won't slow them down much.

The Bitwise Negation Operator

The bitwise negation operator, ~, returns the opposite of the bits in the input value. Like the logical negation operator !, the bitwise negation operator works on a single operand.

```
     binary      decimal     hexadecimal

~    1010        10          A
     0101         5          5
```

Problem 10.9. The bitwise negation operator is sometimes used in graphics applications to reverse images on the screen. We'll write a simple program to do just that.

Create a function that initializes an int pointer to 0xE12000, which just happens to be the address of the graphics screen on the Apple IIGS. You will have to cast the integer to a pointer type, of course. The graphics screen consists of 200 rows of 640 pixels, with each pixel using two bytes. A little math shows that each row consists of 160 bytes, or 80 integers, and that the entire screen uses 16000 integers. Fortunately, on the Apple IIGS, these bytes are all in a row. Your function should use the bitwise negation operator to reverse the bits at the location your pointer points to, storing the bits back in the original spot. With this accomplished, increment the pointer to advance to the next integer. Do this 16000 times in a for loop.

In your main program, call your function, then use a for loop to pause for a while, and call it again. The results can be pretty entertaining.

With debug code on, the program takes a fair amount of time to flip the screen. Try it again with debug code turned off.

Escape Sequences

You've been using escape sequences all along in your C programs, but you heard the name so long ago that it might be a good idea to review what they are. Escape sequences are C's way of putting characters into a character or string constant that can't normally be typed from the keyboard. The most common example is the newline character which you use in printf to move to the start of a new line, and which you read from scanf when the user types a RETURN character, or when you get to the end of a line in a text file. Internally, the newline character is coded as the number 10; to get the newline character in a character constant or string, you type \n.

There are a lot of other characters that can't be typed from the keyboard that are still used fairly frequently, especially in text programs, that also have escape sequences. Table 10.1 shows a complete list of the standard C escape sequences, plus \p, which is unique to Apple-based implementations of C.

escape seq.	value	meaning
a	7	bell
b	8	back space
f	12	form feed
n	10	new line
p	--	p-string
r	13	carriage return
t	9	tab
v	11	vertical tab
\	92	\ character
'	96	' character
"	34	" character
?	63	? character

Table 10.1: Escape Sequences

Many if these escape sequences are only used from text programming environments like the one you can get to by quitting the desktop development environment. In that environment, \a beeps the warning sound, \b moves back one character, \f moves the cursor to the top left corner of the screen and clears the screen, \r moves to the start of the current line, and \v moves up one line. All of these except \r are ignored in the shell window, and even in the text environment, tabs (\t) are ignored, although some printers will do something with the tab key. Even \n is ignored by most of the tools in the Apple IIGS toolbox. In short, how these characters are used depends to some extent on where the text is being written to. In a text environment, virtually all of these characters have some use, but in graphics programming environments, many of the old techniques for dealing with text and a cursor simply don't make sense anymore.

The \p escape sequence is a little odd. There are two common string formats in use on microcomputers, the so-called p-string and the c-string. C-strings are the null-terminated character sequences you have been using with C all along; they consist of a sequence of characters followed by a zero value. P-strings are handled a little differently. A p-string starts with a one byte number that gives the size of the string, followed by the characters. The advantage of p-strings is that they can hold any character value, including a character with a value of 0. The advantage of c-strings is that they are not limited to 255 characters by the value the length byte can hold.

For the most part, you can ignore p-strings in C programs, but Apple's toolbox uses p-strings extensively. Many calls accept p-strings, with no corresponding call to accept a c-string. The \p escape sequence helps you in these situations. When you use a \p as the first character in a string constant, the first character position is filled in with the length of the string, so that the toolbox will recognize the string as a p-string. The string still has a null terminator (which the tools ignore), so you can still use standard C functions to manipulate the string. If you are doing this kind of programming, you might also want to look up the c2pstr and p2cstr functions, two non-standard functions that are also common in Apple based C compilers. They help you switch between these two string formats.

The last four escape sequences are needed to overcome the syntax of the C language. You need \\ to put a \ character into a character or string constant, since a single \ signals the start of an escape sequence. The \' escape sequence is needed in character constants so that the C compiler doesn't think the ' character marks the end of the constant, while \" is needed in string constants for the same reason. The \? escape sequence is needed because of trigraphs, a rather odd feature of the C language that lets you enter characters like [from old keyboards that don't have the character. All you really have to remember is that you can't put two ? characters in a row without causing problems. If you need two ? characters in a string, use \?. "Say what\?\?" is one example.

While these escape sequences cover the most commonly needed special characters, there are others you may need for peculiar situations. For example, many printers use the escape character, whose value is 27, to start special command codes to tell the printer to do something, like switch to 17 characters per inch. You can code any character as an escape sequence by following the \ character with a one to three digit octal value for the number. The value 27 is 033 in octal, so you could use \033 or \33 to get an escape character. Be careful, though: C will use three digits if they are there! If you want to follow the escape character with the character 0, you need to use "\0330", not "\330". The second form gets forced into a single character, since C compilers always grab all three digits when they are there.

For those of you who, like myself, find hexadecimal to be the natural base for dealing with all numbers, and find octal to be even more peculiar than decimal, you can also code numeric escape sequences with one to three hex digits by following the \ character with an x, then the digits, like this: "\x01B".

Problem 10.10. Write a program that will print the following lines to the shell window:

a. Huh???
b. C uses \n for the end of a line.
c. I said, "Quote me!"

Problem 10.11. Try to decode this string the way printf would, then check your work using a C program:

```
"\x54\145\x073\x074\x69\156\x067\5
4\0401\054\x0202\x2c\x20\x33\x2e\n"
```

The goto Statement

One of the most infrequently used statements in good C programs is the goto statement. 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.

```
#include <stdio.h>

void main(void)

{
goto there;
printf("This gets skipped.\n");
there:
  printf("This gets printed.\n");
}
```

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 label. The label tells the compiler where to go to; a corresponding label must appear somewhere in the function, followed by a colon.

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.

People who learned to program in BASIC or FORTRAN generally leaned heavily on the goto statement, since those languages don't have the rich flow of control statements that C does. In fact, for the most part, people who learn these languages first tend to overuse gotos, often creating what is known as spaghetti code. As structured programming languages and structured programming techniques came into style, there was a strong backlash against this practice. Many people got to the point where they associated the so-called goto-less programming with structured programming. In one sense, this was a good idea: overuse of goto statements can make a program very hard to read. Experience shows that programs that are hard to read are often disorganized and buggy, and bugs (which exist in all programs) are harder to track down and eliminate in programs that are hard to read. Unfortunately, this backlash went overboard. Many people now think that the goto statement is evil. We'll look at this issue again after dealing with two statements, break and continue, which are actually extensions of the concept of the goto statement.

Problem 10.12. The while loop, for loop, and do-while loop in C are all fairly sophisticated statements. They make programs easier to write and read, but they are also, technically, not needed. In very early programming languages, and even today in assembly language, you would do the same thing that these statements do with conditional checks like C's if statement and goto statements.

To get a feel for the goto statement, as well as to gain some appreciation for how much work C does for you, rewrite this loop using if statements and goto statements:

```
for (i = 1, i <= 10; ++i)
   printf("%d\n");
```

The break Statement

You saw the break statement briefly back when we looked at the switch statement. The break statement was used to leave the switch statement.

```
switch (i) {
   case 1: printf("*\n");
           break;
   case 2: printf("**\n");
           break;
   case 3: printf("***\n");
           break;
   }
```

You can do exactly the same thing with a goto statement, like this:

```
switch (i) {
   case 1: printf("*\n");
           goto out;
   case 2: printf("**\n");
           goto out;
   case 3: printf("***\n");
           goto out;
   }
out:
```

You can come up with a lot of reasons for using break instead of goto in this situation, and they are all good. I would, in fact, strongly encourage the use of

break instead of goto. The point of this example is basically to show you exactly how the break statement works, with all of the secrecy removed. The break statement, as it turns out, is just a goto statement that jumps out of a loop without the need of a label.

The switch statement is the most common place for using the break statement in C, but there are others. Occasionally, you need to get out of a do loop, while loop, or for loop early. The break statement can be used to leave all of these loops, not just a switch statement. A good example of a case where you would want to use a break statement in 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 C 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 = NULL;
while (ptr != NULL) {
   if (strcmp(ptr->name, name)
      == 0)
      found = ptr;
   ptr = ptr->next;
   }
if (found == NULL)
   printf("false\n");
else
   printf("true: \"%s\"\n",
      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 in a 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 break 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 != NULL) {
   if (strcmp(ptr->name, name)
      == 0)
      break;
   ptr = ptr->next;
   }
if (ptr == NULL)
   printf("false\n");
else
   printf("true: \"%s\"\n",
      name);
```

Occasionally, the true goto statement is actually a better choice than the break statement, and keeping in mind that the break statement is really just a goto will help you realize when this is true. While our search has been improved a lot, the if test after leaving the loop is technically redundant – after all, we knew in the loop whether the value we were looking for had been found. By making use of this information when it is available, we can avoid the if statement altogether:

```
ptr = names;
while (ptr != NULL) {
   if (strcmp(ptr->name, name)
      == 0) {
      printf("true: \"%s\"\n",
         name);
      goto out;
      }
   ptr = ptr->next;
   }
printf("false\n");
out:
```

Whether or not you use a goto statement in a situation like this is really a matter of taste. In general, I would not use a goto statement here; I would use the second form of the loop, with the break statement to exit early. The if check is not very time consuming, and it is a little easier to follow what is going on. If the loop appeared inside another loop, or in some other very time-critical place in the program, though, I would certainly take that into account, and might pick the goto statement to save every bit of time I could.

Here's a complete sample program, shown in listing 10.2, that uses the 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 enter the string "-".

The goto 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.

Listing 10.2

```
/* This program reads a list of names from the keyboard,         */
/* stopping when you enter the string -.  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.  Again,   */
/* this process repeats until you enter the string -.            */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define NAMELENGTH 20                   /* max length of a name */

typedef struct nameType {               /* element of the name list */
   struct nameType *next;
   char name[NAMELENGTH+1];
   }
   nameType,
   *namePtr;                            /* name pointer type */

namePtr names;                          /* list of names */


/* Read a list of names from the keyboard */

void ReadList (void)

{
namePtr ptr;                            /* new name */
char name[NAMELENGTH+1];                /* name read from keyboard */

names = NULL;                           /* no names so far */
```

```
   do {
      printf("Name:");                    /* get a name */
      scanf("%20s", name);
      if (strcmp(name, "-") != 0) {
         ptr = (namePtr) malloc(sizeof(nameType));/* get a new name record */
         ptr->next = names;               /* add the record to the list */
         names = ptr;
         strcpy(ptr->name, name);         /* put the name in the record */
         }
      }
   while (strcmp(name, "-") != 0);
   }


   /* Test to see if names are in the list */

   void Test (void)

   {
   namePtr ptr;                           /* used to trace the list */
   char name[NAMELENGTH+1];               /* name read from keyboard */

   do {
      printf("Name to find:");            /* get a name */
      scanf("%20s", name);
      if (strcmp(name, "-") == 0)         /* quit if no name is given */
         return;
      ptr = names;                        /* scan for the name */
      while (ptr != NULL) {
         if (strcmp(ptr->name, name) == 0) /* name found -> write it */
            {
            printf("true: \"%s\"\n", name);
            goto out;
            }
         ptr = ptr->next;
         }
      printf("false\n");                  /* name not found */
      out: ;
      }
   while (1);                             /* loop forever */
   }


   /* Main program */

   void main (void)

   {
   ReadList();
   printf("\n");
```

```
Test();
}
```

Problem 10.13. In the text, I said that the loop

```
ptr = names;
found = NULL;
while (ptr != NULL) {
   if (strcmp(ptr->name, name)
      == 0)
      found = ptr;
   ptr = ptr->next;
   }
if (found == NULL)
   printf("false\n");
else
   printf("true: \"%s\"\n",
      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 break statement,

```
ptr = names;
while (ptr != NULL) {
   if (strcmp(ptr->name, name)
      == 0)
      break;
   ptr = ptr->next;
   }
if (ptr == NULL)
   printf("false\n");
else
   printf("true: \"%s\"\n",
      name);
```

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 function that builds a list of characters, one for each letter in the alphabet. To do this, you will need to change the nameType struct so it has 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?

The break Statement in Nested Loops

As we said in the last section, the break statement can be used to exit a while loop, do loop, for loop, or case statement. What happens, though, when you break from a loop that is imbedded inside of another loop? To find out, let's try breaking from the inside of a for loop nested within another for loop:

```
#include <stdio.h>
```

```
void main(void)

{
int i,j;

for (i = 1; i <= 10; ++i) {
   for (j = 1; j <= 10; ++j) {
      printf("%3d", i*10+j);
      if (i == j)
         break;
      }
   printf(" ---\n");
   }
}
```

The first thing to notice is that the break is actually imbedded in an if statement that is inside of two for loops. The if statement, of course, has no effect on the way the break statement works. When you run the program, it prints 11 on the first line, then skips to a new line, where it prints 21 and 22. Obviously, the break statement itself will leave the inside for loop, but it only exits one of the loops. In other words, replacing the break statement with a goto statement, it works like this:

```
#include <stdio.h>

void main(void)

{
int i,j;

for (i = 1; i <= 10; ++i) {
   for (j = 1; j <= 10; ++j) {
      printf("%3d", i*10+j);
      if (i == j)
         goto out;
      }
   out: printf(" ---\n");
   }
}
```

To goto or Not to goto

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 C as it is in languages that do not have if-then-else statements, while loops, do loops, switch statements, break statements, and so on.

There is one situation that arises occasionally in C where the goto statement is undeniably necessary. Exiting early from a loop is a tremendous time-saver, as you saw in problem 10.13. In fact, there are many cases where you can save even more time with a judiciously placed early exit. On the other hand, you also just saw that a break can only exit one loop – if that loop is imbedded inside of another loop, and you need to get out of both of them, you need to revert to a goto statement.

I have tried to make the point that there is nothing inherently bad about the goto statement, but if you find yourself using the goto statement very often, you should stop and recall that the bad reputation of the goto statement comes from the fact that it is often overused and improperly used, not from any inherent fault in the statement itself. If you have a background in BASIC or FORTRAN, you may be especially prone to overusing the goto statement. To break these bad habits, I would urge you to avoid the goto statement entirely, with the exception of exiting nested loops, for at least six months. That way, you will be forced to stop and think about how to organize your programs logically around well-organized flow of control, and not jump haphazardly from one place to another. After your "training" period, you will be able to look back and realize just how rarely you really need to use the goto statement.

The continue Statement

By now, you might be thinking that the C language just goes on and on, and you'll never see the end. Take heart: you're getting there! The very last of the executable C statements is the continue statement, a close relative of break. Like break, the continue statement can be used inside of a do loop, while loop, or for loop, but unlike break, the continue statement cannot be used from inside of a switch statement. Of course, if the switch statement itself is inside of one of the loops, you can still use continue, just as you were able to use a break statement that was imbedded in an if statement, as long as the if statement was inside of a loop.

The continue statement is also a form of a goto statement. Instead of jumping out of the loop, though, the continue statement jumps to the end of the loop, so that the loop keeps going. As a quick example, let's look at a simple program that uses the continue statement to skip odd integers.

```
#include <stdio.h>

void main(void)
```

```
{
int i;

for (i = 1; i <= 10; ++i) {
   if (i & 1)
      continue;
   printf("%d\n", i);
   }
}
```

The way that we detected odd numbers is itself a bit odd, so lets stop and take a quick look at that technique in passing. Writing a few numbers in both decimal and binary, you can quickly see a pattern emerge:

decimal	binary
1	000001
2	000010
3	000011
4	000100
5	000101
6	000110
7	000111
8	001000
9	001001
10	001010

As you can see, all of the odd values have a rightmost bit of one. Thinking about how binary numbers are formed, this makes sense: after all, the rightmost column is the one's column. To detect odd numbers, then, we need to see when the rightmost bit is 1. We can do that using the & operator, which you saw earlier. Anding a number with 1 will zero any bits except the 1 bit. If the number is odd, the result will be 1, and for even numbers, we will end up with 0. Since 0 is used as false in C, and anything else is true, the statement "if (i & 1) ..." will execute the continue statement for odd numbers, skipping it for even numbers.

Getting back to the continue statement itself, you probably already see how it works, but here's the same program with a goto statement, so you can see what is happening a bit easier:

```
#include <stdio.h>

void main(void)

{
int i;

for (i = 1; i <= 10; ++i) {
   if (i & 1)
      goto loop;
   printf("%d\n", i);
   loop: ;
   }
}
```

The loop label would always appear right before the } character that closes the loop.

As with the break statement, if two loops are imbedded inside of each other, the continue statement applies to the innermost loop.

Lesson Ten
Solutions to Problems

Solution to problem 10.1.

```
/* Simple base conversion program.  Enter a number */
/* in decimal (no leading 0), octal (leading       */
/* zero), or hexadecimal (leading 0x), and the     */
/* program will echo the value back in all three   */
/* bases.  Enter 0 in any base to quit.            */

#include <stdio.h>

void main(void)

{
long val;

do {
   printf("\nValue: ");                  /* write the prompt */
   val = 0L;                           /* default (in case of error) */
   scanf("%li", &val);                 /* read the value */
   printf("Decimal    : %ld\n", val);   /* print the value */
   printf("Octal      : 0%lo\n", val);
   printf("Hexadecimal: 0x%lX\n", val);
   }
while (val);
}
```

Solution to problem 10.2.

The 0 in the conversion specifier %04X tells printf to use leading zeros to fill out the number if there aren't enough digits to fill the entire field width. The 4 is the field width. So, for example, if the value to print is 0x12, printf will print "0012". With a conversion specifier of %0X, printf would not need to fill out the field width, so it would print "12", while with a conversion specifier of %4X, printf would print " 12".

Normally, when a number follows the % character, it is used as a field width, so you might think the 0 would become part of the field width. The reason there is no confusion is because the leading zero is not used for anything anyway, so removing it does not effect the field width. Of course, a field width of 0 doesn't make sense, so there is never a need to use just a 0 for the field width.

Solution to problem 10.3.

```
/* Print a hex-binary-decimal conversion table. */

#include <stdio.h>

/* Print an integer as a 16 digit binary number         */
/*                                                      */
/* Parameters:                                          */
/*    val - value to print                              */

void PrintBin (int val)

{
if (val & 0x8000)
   printf("1");
else
   printf("0");
if (val & 0x4000)
   printf("1");
else
   printf("0");
if (val & 0x2000)
   printf("1");
else
   printf("0");
if (val & 0x1000)
   printf("1");
else
   printf("0");
if (val & 0x0800)
   printf("1");
else
   printf("0");
if (val & 0x0400)
   printf("1");
else
   printf("0");
if (val & 0x0200)
   printf("1");
else
   printf("0");
if (val & 0x0100)
   printf("1");
else
   printf("0");
if (val & 0x0080)
   printf("1");
else
   printf("0");
```

```
if (val & 0x0040)
   printf("1");
else
   printf("0");
if (val & 0x0020)
   printf("1");
else
   printf("0");
if (val & 0x0010)
   printf("1");
else
   printf("0");
if (val & 0x0008)
   printf("1");
else
   printf("0");
if (val & 0x0004)
   printf("1");
else
   printf("0");
if (val & 0x0002)
   printf("1");
else
   printf("0");
if (val & 0x0001)
   printf("1");
else
   printf("0");
}


/* Main program                                          */

void main(void)

{
int i;                                  /* loop variable */

printf("dec hex binary\n");             /* print the header */
printf("--- --- ------\n");
for (i = 0; i < 16; ++i) {              /* print the values */
   printf("%3d 0x%X ", i, i);
   PrintBin(i);
   printf("\n");
   }
}
```

Solution to problem 10.4.

```
/* Print a hex-binary-decimal conversion table. */

#include <stdio.h>

/* Test a number to see if it is negative               */
/*                                                      */
/* Parameters:                                          */
/*    val - number to test                              */

int IsNegative (int val)

{
return val & 0x8000;
}


/* Main program                                         */

void main(void)

{
if (IsNegative(1))
   printf("1 is negative.\n");
else
   printf("1 is positive.\n");
if (IsNegative(0))
   printf("0 is negative.\n");
else
   printf("0 is positive.\n");
if (IsNegative(-1))
   printf("-1 is negative.\n");
else
   printf("-1 is positive.\n");
if (IsNegative(-32767))
   printf("-32767 is negative.\n");
else
   printf("-32767 is positive.\n");
if (IsNegative(32767))
   printf("32767 is negative.\n");
else
   printf("32767 is positive.\n");
}
```

Solution to problem 10.5.

```
#include <stdio.h>

void main(void)

{
int a,b;

a = 0x3C3C;
b = 0xAAAA;
printf("%04X & %04X = %04X\n", a, b, a&b);
printf("%04X | %04X = %04X\n", a, b, a|b);
}
```

Solution to problem 10.6.

```
/* My own tolower function. */

#include <stdio.h>


/* Return the lowercase equivalent of the input.                */
/*                                                              */
/* Parameters:                                                  */
/*    ch - character to convert                                 */
/*                                                              */
/* Returns: Lowercase equivalent of ch                          */

char lower (char ch)

{
if ((ch >= 'A') && (ch <= 'Z'))
   ch = ch | 0x0020;
return ch;
}


/* Main program. */

void main(void)

{
printf("The lowercase equivalent of %c is %c.\n", 'A', lower('A'));
printf("The lowercase equivalent of %c is %c.\n", 'a', lower('a'));
printf("The lowercase equivalent of %c is %c.\n", 'U', lower('U'));
printf("The lowercase equivalent of %c is %c.\n", '4', lower('4'));
printf("The lowercase equivalent of %c is %c.\n", '{', lower('{'));
}
```

Solution to problem 10.7.

```
/* Read two integers, then write the exclusive or of    */
/* the integers.                                        */

#include <stdio.h>

void main(void)

{
int a,b;

printf("First value: ");
scanf("%i", &a);
printf("Second value: ");
scanf("%i", &b);
printf("0x%04X ^ 0x%04X = 0x%04X\n", a, b, a^b);
}
```

The first time you run the program, the inputs are 0x1234 and 0xFFFF. Expressing these values as binary numbers, you get 0001001000110100 for 0x1234, and 1111111111111111 for 0xFFFF. Exclusive oring these two numbers reverses all of the bits in the first value, since all of the bits in the second value are set. The result is 0xEDCB, which, in binary, is 1110110111001011. Doing the operation again with the result of the first run and 0xFFFF, you reverse the same bits back to their original values, so the result is 0x1234. If you run the program with 0xEDCB and 0x1234, the result will be 0xFFFF.

Solution to problem 10.8.

```
/* Encrypt or decrypt a text file.                      */

#include <stdio.h>
#include <stdlib.h>

#define BUFFSIZE 64                     /* size of the character buffer */

int main(void)

{
char ch[BUFFSIZE];                      /* character from the file */
int numCh;                              /* # of characters read */
int i;                                  /* loop variable */
char name[81];                          /* file name */
int password;                           /* password value */
FILE *inFile, *outFile;                 /* file variables */

printf("File to encrypt: ");            /* open the input file */
scanf("%80s", name);
inFile = fopen(name, "r");
if (inFile == NULL) {
   fprintf(stderr, "Could not open %s.\n", name);
   return -1;
```

```
   }
printf("Output file    : ");            /* open the output file */
scanf("%80s", name);
outFile = fopen(name, "w");
if (outFile == NULL) {
   fprintf(stderr, "Could not open %s.\n", name);
   fclose(inFile);
   return -1;
   }
printf("Password       : ");            /* get the password */
scanf("%d", &password);
srand(password);                        /* use it to set up rand() */

do {                                    /* encrypt/decrypt the file */
   numCh = fread(ch, sizeof(char), BUFFSIZE, inFile);
   if (numCh > 0) {
      for (i = 0; i < numCh; ++i)
         ch[i] = ch[i] ^ rand();
      fwrite(ch, sizeof(char), numCh, outFile);
      }
   else
      break;
   }
while (1);

fclose(inFile);                         /* close the files */
fclose(outFile);
return 0;
}
```

Solution to problem 10.9.

```
/* Invert the graphics screen */

void InvertScreen(void)

/* Invert all of the bits on the graphics screen */

{
int *ptr;                               /* graphics screen pointer */
int count;                              /* loop counter */

ptr = (int *) 0xE12000;
for (count = 0; count < 16000; ++count) {
   *ptr = ~*ptr;
   ++ptr;
   }
}

void main(void)
```

```
{
int loop;

InvertScreen();
for (loop = 1; loop <= 100; ++loop)
   ;
InvertScreen();
}
```

Solution to problem 10.10.

```
#include <stdio.h>

void main(void)

{
printf("Huh\?\?\?\n");
printf("C uses \\n for the end of a line.\n");
printf("I said, \"Quote me!\"\n");
}
```

Solution to problem 10.11.

```
#include <stdio.h>

void main(void)

{
printf("\x54\145\x073\X074\X69\156\x067\54\0321\054\x0202\x2c\x20\x33\x2e\n");
}
```

Solution to problem 10.12.

```
#include <stdio.h>

void main(void)

{
int i;

i = 1;
top:
printf ("%d\n", i);
++i;
if (i <= 10) goto top;
}
```

Solution to problem 10.13.

```
/* Find the average number of compares needed to find a character */
```

```
/* from a list of characters that make up the alphabet.          */

#include <stdio.h>
#include <stdlib.h>

#define NAMELENGTH 20                   /* max length of a name */
#define COUNT 10000                     /* number of tests */


typedef struct listType {               /* element of the list */
   struct listType *next;
   char ch;
   }
   listType,
   *listPtr;                            /* list pointer type */

long compares;                          /* # of compares done */
listPtr list;                           /* ptr to the first element */


/* Build a list of characters */

void BuildList (void)

{
listPtr ptr;                            /* new list element */
char ch;                                /* character for the list */

list = NULL;                            /* no elements so far */

for (ch = 'a'; ch <= 'z'; ++ch) {
   ptr = (listPtr) malloc(sizeof(listType)); /* get a new list record */
   ptr->next = list;                    /* add the record to the list */
   list = ptr;
   ptr->ch = ch;                        /* put the char in the struct */
   }
}


/* Check the list */

void Test (void)

{
listPtr ptr;                            /* used to trace the list */
char ch;                                /* character to test for */
int i;                                  /* loop counter */

srand(23456);                           /* initialize rand() */
for (i = 0; i < COUNT; ++i) {
```

```
   ch = (rand() %26) + 'a';            /* create a character */
   ptr = list;                         /* scan for the character */
   while (ptr != NULL) {
      ++compares;                      /* update the # of compares */
      if (ptr->ch == ch)               /* found -> exit loop */
         break;
      ptr = ptr->next;
      }
   }
}


/* Main program */

void main (void)

{
compares = 0;
BuildList();
Test();
printf("The average # of compares was %.2f.\n", ((float)compares)/COUNT);
}
```

Lesson Eleven
More Miscellaneous Useful Stuff

Unions

Unions are a rather peculiar data type that is used to overlay two different variables, causing them to use the same memory. There are some good reasons to do this, and we will get to those gradually, but we'll start off by exploring how you create unions, and exactly what they do, with a simple example.

Type in this program and run it. We'll talk about what it does in a moment.

```
#include <stdio.h>

void main(void)

{
union int_float {
   int i;
   float f;
   } u;

u.i = 4;
printf("u.i = %d\n", u.i);
u.f = 1.2;
printf("u.f = %f\n", u.f);
printf("u.i = %d\n", u.i);
}
```

As you can see, a union looks a lot like a struct. In fact, all of the rules you have learned so far that apply to structures also apply to unions. You define union types the same way that you define struct types; you get at variables inside of a union with the . operator, just like you do with a struct; and you use the -> operator to access variables imbedded in a pointer to a union, again, just as you do with structures. In fact, if you replace the union in this program with struct, it will do exactly what you expect it to: print 4 for u.i, then print 1.2 for u.f and print the value of u.i a second time.

Of course, when you run this program as is, it does something different. The first time u.i is printed, you get 4, just as you expect. When u.f is printed, you also get 1.2, just as you expect. In a union, though, variables overlay each other, using the same memory. In this program, when we store a value in u.f, we wipe out the value in u.i, so the second time this program prints u.i, the value is changed.

One reason for using unions is to save space. In a struct that has an int variable and a float variable, each struct needs six bytes (two for the int and four for the float). In this union, the union variable u only uses four bytes: the size of the union is the same as the size of the largest variable. By cleverly combining unions and structures, you can create programs that use a lot less space than programs that only use structures.

Listing 11.1 shows one use of unions. 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 union to overlay various structures. The variable kind is used to keep track of what sort of information is in the union. Each of the shapes has a color, so, rather than put the color in the union and recreating it several times, we put the union inside of a struct. This is a very common technique for mixing information that is consistent across several different kinds of things with information that varies from entry to entry.

Animation is pretty slow with debug code on, so once you get the program to work, try it with debug off.

Listing 11.1

```
/* Do a random walk with 10 random shapes */

#include <stdlib.h>

#include <quickdraw.h>

#define NUMSHAPES 10                    /* # of shapes to animate */
#define WALKLENGTH 100                  /* # of "steps" in the walk */

#define MAXX 316                        /* size of the graphics screen */
#define MAXY 83

typedef enum shapeKind {triangle, square, star} shapeKind;

typedef struct shapeType {              /* information about one shape */
   int color;                           /* color */
   shapeKind kind;                      /* kind of shape */
   union {
      struct {                          /* points for a triangle */
         int x1,x2,x3,y1,y2,y3;
         } t;
      struct {                          /* points for a square */
         int x1,x2,x3,x4,y1,y2,y3,y4;
         } s;
      struct {                          /* points for a star */
         int x1,x2,x3,x4,x5;
         int y1,y2,y3,y4,y5;
         } p;
      } coord;
   } shapeType;


shapeType shapes[NUMSHAPES];            /* current array of shapes */
shapeType oldShapes[NUMSHAPES];         /* shapes in last position */


void InitGraphics (void)

/* Standard graphics initialization.                              */

{
SetPenMode(2);                          /* pen mode = xor */
SetPenSize(3,1);                        /* use a square pen */
}
```

```
void DrawShape (shapeType s)

/* This subroutine draws one of the shapes on the screen.       */
/*                                                              */
/* Parameters:                                                  */
/*    s - shape to draw                                         */

{
SetSolidPenPat(s.color);            /* set the pen color for the shape */

switch (s.kind) {

   case triangle:                        /* draw a triangle */
      MoveTo(s.coord.t.x1, s.coord.t.y1);
      LineTo(s.coord.t.x2, s.coord.t.y2);
      LineTo(s.coord.t.x3, s.coord.t.y3);
      LineTo(s.coord.t.x1, s.coord.t.y1);
      return;

   case square:                          /* draw a square */
      MoveTo(s.coord.s.x1, s.coord.s.y1);
      LineTo(s.coord.s.x2, s.coord.s.y2);
      LineTo(s.coord.s.x4, s.coord.s.y4);
      LineTo(s.coord.s.x3, s.coord.s.y3);
      LineTo(s.coord.s.x1, s.coord.s.y1);
      return;

   case star:                            /* draw a star */
      MoveTo(s.coord.p.x1, s.coord.p.y1);
      LineTo(s.coord.p.x2, s.coord.p.y2);
      LineTo(s.coord.p.x3, s.coord.p.y3);
      LineTo(s.coord.p.x4, s.coord.p.y4);
      LineTo(s.coord.p.x5, s.coord.p.y5);
      LineTo(s.coord.p.x1, s.coord.p.y1);
      return;
   }
}


void CreateShape (shapeType *s)

/* 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                                       */

{
int cx,cy;                            /* center point for the shape */
```

```
s->color = rand() % 3 + 1;              /* get a color */
cx = rand() % (MAXX - 38) + 19;         /* get the center position,   */
cy = rand() % (MAXY - 16) + 8;          /* picking the points so the  */
                                        /* shape is in the window     */

switch (rand() % 3) {                   /* set the initial positions */
   case 0:                              /* set up a triangle */
      s->kind = triangle;
      s->coord.t.x1 = cx-19;
      s->coord.t.y1 = cy+4;
      s->coord.t.x2 = cx;
      s->coord.t.y2 = cy-8;
      s->coord.t.x3 = cx+19;
      s->coord.t.y3 = cy+4;
      return;

   case 1:                              /* set up a square */
      s->kind = square;
      s->coord.s.x1 = cx-15;
      s->coord.s.y1 = cy-6;
      s->coord.s.x2 = cx+15;
      s->coord.s.y2 = cy-6;
      s->coord.s.x3 = cx-15;
      s->coord.s.y3 = cy+6;
      s->coord.s.x4 = cx+15;
      s->coord.s.y4 = cy+6;
      return;

   case 2:                              /* set up a star */
      s->kind = star;
      s->coord.p.x1 = cx-13;
      s->coord.p.y1 = cy+7;
      s->coord.p.x2 = cx;
      s->coord.p.y2 = cy-8;
      s->coord.p.x3 = cx+13;
      s->coord.p.y3 = cy+7;
      s->coord.p.x4 = cx-21;
      s->coord.p.y4 = cy-3;
      s->coord.p.x5 = cx+21;
      s->coord.p.y5 = cy-3;
      return;
   }
}
```

```
void UpdateShape (shapeType *s)

/* This subroutine moves a shape across the screen in a random  */
/* walk.                                                        */
/*                                                              */
/* Parameters:                                                  */
/*    s - shape to update                                       */

{
int dx,dy;                              /* movement direction */

dx = rand() % 3 - 1;                    /* get the walk direction */
dy = rand() % 3 - 1;

switch (s->kind) {                     /* make sure we don't walk off of */
                                       /* the screen, then update the    */
                                       /* position                       */
   case triangle:                        /* check a triangle */
      if (dx == -1)
         if (s->coord.t.x1 < 1)
            dx = 0;
      if (dx == 1)
         if (s->coord.t.x3 >= MAXX)
            dx = 0;
      if (dy == -1)
         if (s->coord.t.y2 < 1)
            dy = 0;
      if (dy == 1)
         if (s->coord.t.y3 >= MAXY)
            dy = 0;
      s->coord.t.x1 = s->coord.t.x1+dx; /* update a triangle */
      s->coord.t.y1 = s->coord.t.y1+dy;
      s->coord.t.x2 = s->coord.t.x2+dx;
      s->coord.t.y2 = s->coord.t.y2+dy;
      s->coord.t.x3 = s->coord.t.x3+dx;
      s->coord.t.y3 = s->coord.t.y3+dy;
      return;

   case square:                          /* check a square */
      if (dx == -1)
         if (s->coord.s.x1 < 1)
            dx = 0;
      if (dx == 1)
         if (s->coord.s.x2 >= MAXX)
            dx = 0;
      if (dy == -1)
         if (s->coord.s.y1 < 1)
            dy = 0;
      if (dy == 1)
         if (s->coord.s.y3 >= MAXY)
```

```
            dy = 0;
         s->coord.s.x1 = s->coord.s.x1+dx; /* update a square */
         s->coord.s.y1 = s->coord.s.y1+dy;
         s->coord.s.x2 = s->coord.s.x2+dx;
         s->coord.s.y2 = s->coord.s.y2+dy;
         s->coord.s.x3 = s->coord.s.x3+dx;
         s->coord.s.y3 = s->coord.s.y3+dy;
         s->coord.s.x4 = s->coord.s.x4+dx;
         s->coord.s.y4 = s->coord.s.y4+dy;
         return;

      case star:                          /* check a star */
         if (dx == -1)
            if (s->coord.p.x4 < 1)
               dx = 0;
         if (dx == 1)
            if (s->coord.p.x5 >= MAXX)
               dx = 0;
         if (dy == -1)
            if (s->coord.p.y2 < 1)
               dy = 0;
         if (dy == 1)
            if (s->coord.p.y1 >= MAXY)
               dy = 0;
         s->coord.p.x1 = s->coord.p.x1+dx; /* update a star */
         s->coord.p.y1 = s->coord.p.y1+dy;
         s->coord.p.x2 = s->coord.p.x2+dx;
         s->coord.p.y2 = s->coord.p.y2+dy;
         s->coord.p.x3 = s->coord.p.x3+dx;
         s->coord.p.y3 = s->coord.p.y3+dy;
         s->coord.p.x4 = s->coord.p.x4+dx;
         s->coord.p.y4 = s->coord.p.y4+dy;
         s->coord.p.x5 = s->coord.p.x5+dx;
         s->coord.p.y5 = s->coord.p.y5+dy;
         return;
      }
   }


   void main(void)

   /* main program                                                */

   {
   int i,j;                           /* loop variables */

   InitGraphics();                    /* set up the graphics window */
   srand(6289);                       /* initialize rand() */

   for (i = 0; i < NUMSHAPES; ++i) { /* set up and draw the initial shapes */
```

```
      CreateShape(&shapes[i]);
      DrawShape(shapes[i]);
      }

   for (i = 0; i < WALKLENGTH; ++i) {      /* do the random walk */
      for (j = 0; j < NUMSHAPES; ++j) {    /* move the shapes */
         oldShapes[j] = shapes[j];
         UpdateShape(&shapes[j]);
         }
      for (j = 0; j < NUMSHAPES; ++j) {    /* redraw the shapes */
         DrawShape(shapes[j]);
         DrawShape(oldShapes[j]);
         }
      }
   }
```

Problem 11.1. One common use of unions takes advantage of the fact that the variables in the union overlap. This fact can be used to examine the values of a complicated variable type. 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 union.

Define a union that consists of a long integer and a structure containing two integers. As you know, an integer variable requires two bytes of storage, while a long variable requires four bytes of storage, so the union puts the two integers in the same memory as the long integer, giving you a way to 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 union, then write the two integers.

Experiment with this program a bit. What you should find is that for values up to 32767, 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.

Separate Compilation

The programs you are writing in this course are small compared to some of the programs that people regularly write in C. In programs that are several thousand lines long, organizing the program so you can find things and keep one part of the program from interfering with another part becomes a very serious problem. In C, the most popular way to organize these large programs is to break them up into smaller pieces. Each of the pieces is responsible for a particular function or small group of functions within the program.

To see how this might be done, take a look at the PRIZM desktop development system you are using. PRIZM is a large program, and, in fact, it is broken up into several parts. The part of the system that you use most often is the editor. This is actually separated into two different files. One of the files is responsible for keeping track of the keys you type and how you use the mouse. It figures out what you want to do, then calls the second part, which manages the internal buffers that hold text and print this text to the screen. Yet another part handles the printer, while still another part implements the debugger. PRIZM actually consists of even more individual pieces than those described here, but you get the idea.

To see how C handles separate compilation, we will take one of our old programs and split it up into two pieces. Here's one of the very first programs we wrote that used functions:

Listing 11.2

```
/* Draw three rectangles */

#include <quickdraw.h>

void Rectangle (int left, int right, int top, int bottom, int color)

/* 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                   */

{
unsigned i;                     /* loop variable */

SetSolidPenPat(color);          /* draw a rectangle */
for (i = top+1; i <= bottom-1; ++i) {
   MoveTo(left,i);
   LineTo(right,i);
   }
SetSolidPenPat(0);              /* outline it in black */
MoveTo(left,top);
LineTo(left,bottom);
LineTo(right,bottom);
LineTo(right,top);
LineTo(left,top);
}


void InitGraphics(void)

/* Standard graphics initialization.                            */

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


void main(void)

{
InitGraphics();                 /* set up for graphics */
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 */
}
```

In a huge program, one of the things we might want to do is separate all of the functions that deal with graphics into a separately compiled module. There are several advantages to this: the individual files we need to edit are smaller and more manageable; we only need to include quickdraw.h one time, so if we change the part of the program that doesn't use the graphics screen, the compiler won't have to waste time recompiling quickdraw.h; and if the program is moved to another computer that handles graphics differently, all of our graphics routines are in one place, making them easier to find and change.

Splitting the program into separate files is actually pretty easy in this case. The function main goes into one source file – we'll call it main.cc – and the rest of the program goes into another source file, which we will call graph.cc. Compiling the program is a bit harder, though. To compile this program, we will move to the shell window right away, and use shell commands. On a really large project, it would pay to learn the shell's scripting language, so you could create a shell program that would create your program in one step, but in this example, we'll do it all by hand.

There are three steps to creating our program. The first is to compile graph.cc using the shell's compile command. The command looks like this:

```
compile main.cc keep=main
```

This command tells the shell to compile the file main.cc, which you just created, and save the output from the compiler in object files that start with the name main. The C compiler will actually create two files, main.root and main.a. For the most part, you can ignore the two files, since the name main is the only one you use.

The next step is to compile the second source file, graph.cc, using the command

```
compile graph.cc keep=graph
```

From looking at the shell window as your programs compile, you may already be aware of something called the linker. The linker is a program that takes the files created by the compiler and combines them with libraries, which contain prewritten functions like printf, to produce the final program. Now that you are handling all of the steps of creating the program on your own, you must use the shell's link command to get the linker to do its job. You need to tell the linker the names of the files created by the compiler, using the same name you used when you compiled the file, and give it a name for the executable program, like this:

```
link main graph keep=main
```

You can use the same name for the executable program, as we just did, or use a different one. The order of the files is important: you need to give the linker the name of the file that has the function main in it first. The order of any other files is unimportant.

Now that you have created a program, you can run it by typing the name of the file, main.

Problem 11.2. One of the many advantages of separate compilation is the amount of compile time it can save you. You know from experience that it takes a long time to compile a program that includes quickdraw.h compared to the time it takes to compile a program that does not use this header file; now you can avoid that time in many cases.

Change the file main.cc so it draws a fourth rectangle with this call:

```
Rectangle(55,305,45,85,1);
```

What commands are needed to create the program?

Header Files

One of the things the compiler has been doing for you all along is checking to make sure that you call functions correctly. If you define a function as needing five integer parameters, and then pass two integers, the compiler tells you about your mistake.

Take a look at the file main.cc from the last section:

```
void main(void)

{
InitGraphics();
Rectangle(10,250,10,60,0);
Rectangle(220,270,30,50,1);
Rectangle(50,300,40,80,2);
}
```

When the compiler sees this program, it has no way of knowing whether InitGrapics really needs parameters or not, or if Rectangle really needs five integers. If the program is wrong, the compiler doesn't know it – and your program could crash, corrupt memory, or just not work quite right.

To regain some of the safety we have apparently lost, we need a way to tell the compiler how the

functions are declared. C handles this with something called a function declaration: the first part looks just like the functions you have created all along, but there are no statements. Instead, the function ends with a semicolon, telling the compiler that nothing else follows.

Here are the declarations for our graphics.cc module:

```
void Rectangle (int left,
                int right,
                int top,
                int bottom,
                int color);

void InitGraphics(void);
```

You can put these declarations in the file main.cc, and everything would work fine: the compiler would find the declarations and check the function calls, and if you were to make a mistake, the compiler would catch it. The problem is that in a very large program, several different source files might use the functions defined in graph.cc. If you change one of the functions, you want to be able to change the function definitions these other files use in one place, not hunt through the files, hoping you find all of the places you need to change. You might also want a convenient place to find a list of all of the functions in a separately compiled file.

The traditional way to handle this situation in C is to put the function declarations in a header file. Also by convention, this file has the same name as the C source file with the .cc removed, and .h added. For our example, then, you should save the function declarations for Rectangle and InitGraphics in a file called graph.h. To get access to the header file, you use an include in main.cc, just as you would use an include to get access to the functions in stdio.h. The only difference is that file names for standard header files like stdio.h are enclosed in brackets, while file names for your own header files are inclosed in parenthesis.

Putting in the include statement, main.cc looks like this:

```
#include "graph.h"

void main(void)

{
InitGraphics();
Rectangle(10,250,10,60,0);
Rectangle(220,270,30,50,1);
Rectangle(50,300,40,80,2);
}
```

Just to be sure you understand what was just said, stop and try it: make the changes, and run the program.

As you can see, there is a lot more work involved in creating a program using separate compilation, and you shouldn't do it lightly. When your programs regularly start to approach 500 lines or so, or when you find yourself constantly reusing the same subroutines in several different programs, it is time to look at separate compilation. At that point, it would also be a great idea to find a good book on structured programming techniques to learn some of the tricks and ideas that will help you write large programs quickly and accurately.

Storage Classes

One of the things that isn't very obvious from a high-level language like C is that there are a lot of ways to set aside space for a variable. In a sense, you have already seen two of these: you can allocate a variable, either globally or locally, or you can allocate space for a variable using malloc, and keep track of the variable with a pointer. It turns out that in most implementations of C, and ORCA/C is no exception, local variables are also allocated a different way than global variables. The reason for this won't become obvious until a little later, when we study recursion, but the practical implication is something you already know about. To see how this effects us, let's try a short sample program.

```
#include <stdio.h>

void test1 (int i)

{
int j;

if (i)
   j = i;
printf("In test1, i = %d, "
   "j = %d\n", i, j);
}


void test2 (int i)

{
int j;

if (i)
```

```
   j = i;
printf("in test2, i = %d, "
   "j = %d\n", i, j);
}


void main(void)

{
test1(2);
test2(3);
test1(0);
}
```

Let's start by tracing through this program by hand to see what it *looks* like it will do. On the first call to test1, i will be non-zero, so j will be set to 2 (which is the value of i), and the program will print

```
In test1, i = 2, j = 2
```

On the call to test2, pretty much the same thing happens. This time, the program prints

```
In test2, i = 3, j = 3
```

On the second call to test1, the parameter i is set to 0, so the if condition is false, and j dose not get assigned a value. You might expect, then, that the program would print

```
In test1, i = 0, j = 2
```

In fact, the value printed for j turns out to be something entirely different! To say the least, this is a bit weird. The reason all of this happens is that space for local variables is allocated when the function is called, not when the program is compiled. There are a number of advantages to this, most of which make a lot more sense to compiler writers than to beginning programmers, but the effect is that the value of j in test1 gets stomped on when test2 runs. The second time test1 was called, the local variable j wasn't set at all, so the value printed was whatever happened to have been left in memory by the previous operations.

In C, variables that are allocated off of the stack like this are called auto variables; auto is called the storage class of the variable. In fact, you could define the function test1 like this, telling the compiler the storage class explicitly:

```
void test1 (int i)
{
auto int j;

if (i)
   j = i;
printf("in test2, i = %d, "
   "j = %d\n", i, j);
}
```

The program will do the same thing, but it takes more typing, and most C programmers don't put the auto in, anyway. The idea of specifying the storage class is important, though, since there is another storage class that you can use, called static. When you use the static storage class on a local variable, you tell the compiler that you don't want the variable to come from the stack. In this case, the compiler actually reserves space in your program for the variable, just like it does for global variables. The net effect is that the program remembers the values of the variables between function calls, and keeps the various variable values straight. Going back and changing the variables to static, our test program looks like this:

```
#include <stdio.h>

void test1 (int i)

{
static int j;

if (i)
   j = i;
printf("In test1, i = %d, "
   "j = %d\n", i, j);
}


void test2 (int i)

{
static int j;

if (i)
   j = i;
printf("in test2, i = %d, "
   "j = %d\n", i, j);
}


void main(void)
```

```
{
test1(2);
test2(3);
test1(0);
}
```

When you run this program, you get the results you probably expected originally.

There is another advantage to static variables that isn't very obvious. Auto variables are allocated off of the run-time stack, which has a default size of 8192 bytes. Due to the way the 65816 (the 65816 is the CPU used in the Apple IIGS) and the Apple IIGS are designed, the largest that you can make the stack is about 32768 bytes long. In other words, if you create a large array as a local variable, you could fill up all of this memory. When you do this, your program will crash. With static variables, the memory doesn't come from the stack, and you can fill up all of your memory before you would run out of space. On the off chance that you did run out of space with a static variable, you would get an error message from the loader, too, instead of just seeing your program crash. While it probably isn't necessary to worry about stack space for your local scalar variables, if you are declaring large local arrays, you should keep this in mind.

There is another storage class that you can use with local variables, called register. Register variables are a special case of the storage class auto. The register storage class is a hint to the compiler that the variable will be used very frequently, so it should handle the variable as efficiently as possible. Telling the compiler that all of the variables are register variables is self-defeating. In addition, the ORCA/C compiler (and many other compilers, too) doesn't do anything with the register storage class, anyway. Finally, on computers where the register storage class would make a difference, a good compiler will do a better job of figuring out how to handle variables than you can. In short, unless you are sure you are working on a machine where register variables make a difference, and you are also sure that the compiler you are using is stupid enough not to do the sensible thing on its own, there is no point in using the register storage class.

Global variables can have a storage class specifier, too, but you can't use auto or register with global variables. With global variables, you can specify a storage class of extern or static. Just to keep things interesting, the C language says that the default storage class for a global variable is extern, but that there is a slight difference between actually saying that a global variable is extern, and just leaving the storage class specifier off altogether.

Before going much further, let's stop and look at why there are different storage class specifiers for global variables at all. The reason is tied up in the fact that C supports separate compilation, which you saw in the last section. It is possible for one of the separately compiled files to use a variable in another file, but that presents two problems. The first is that C insists that you define a variable before using it. If you define the global variable

```
int i;
```

in two separate files, you will actually get a link error: the linker, finding two occurrences of the same global variable, will give you a duplicate label error. In other words, this declaration really did two things: it told the compiler that the integer variable i exists in the program, and it also told the compiler to set aside space for the variable. Adding the extern storage class, like this:

```
extern int i;
```

tells the compiler something a little different. In this case, you are still telling the compiler that the integer variable i exists in the program, but you are also telling the compiler that some other source file will reserve space for the variable, so the compiler doesn't need to do it.

Of course, there is another reasonable way to want to handle this situation: it could very well be that you want to have a global variable in both source files called i, but they have nothing to do with one another. In this case, you want to *hide* your global variable from other source files. That's what the static storage class does. When you define a global variable with a static storage class, like this:

```
static int i;
```

the variable works just like you are used to in the source file where it appears, but it is invisible to all other source files. The other source files can define a global variable of the same name with no conflict at all.

Incidentally, functions also have a storage class. In the case of a function, the default storage class is extern, which tells the linker that other source files can use the function. As with variables, a storage class of static tells the linker that the function is private to the source file it appears in, and that other source files should not be aware that the function exists. With functions, though, there is no difference between the default storage class of extern and specifying extern explicitly, since the compiler has other ways of telling

if you are defining the function: it looks to see if there are any statements to compile!

Problem 11.3. In this problem, you will experiment with the static and extern storage classes a bit in a separately compiled program. The point of this problem is to play with storage classes in a very short program, though, so we'll just be writing a short program that doesn't do anything really useful.

This program will consist of three source files, which we will refer to as main.cc, a.cc, and b.cc. In main.cc, create the function main, and have it call a function called loop with two integer variables, 1 and 10, then call doit, like this:

```
void main(void)

{
loop(1,10);
doit();
}
```

In the source file a.cc, define two global variables, start and stop. These should have the default storage class, and be defined as int variables. Also, define a function called loop which also has the default storage class. This is the function that you called from main.cc; it should set the variable start to the value of its first parameter, and the variable end to the value of the second parameter.

Define the function doit in the source file b.cc. This function must also have the default storage class of extern, since it is called from main.cc. The function doit has a single call to another function called loop. This is a different function from the one that appears in the file a.cc. This loop function is only called from doit, and should not conflict with the function in the source file a.cc, so this function must be static. The function should loop from start to stop, printing the integers, like this:

```
for (i = start; i <= stop; ++i)
   printf("%d\n", i);
```

Of course, since the function is using the start and stop variables defined in the source file a.cc, you also need to define the same variables in the source file b.cc with a storage class of extern.

Run the program to make sure you got all of the storage classes correct. If not, you will get either a compiler or linker error.

Initializers

By now you have written several programs where you declare a variable, then initialize it to something almost right away. C can actually initialize the variable for you so that you don't have to use two different statements, as in this program:

```
#include <stdio.h>

void main (void)

{
int i = 4;

printf("%d\n", i);
}
```

There are some advantages to initializers besides just saving some typing, although the exact advantages may vary from one C compiler to another. One advantage that is common in all C programs, though, is that an initializer collects the initial value with the variable, which is a sort of comment all by itself: it tells the person reading the program that the variable should start at some specific value, and lists the value. This may seem like a pretty minor point, but when you are digging through a 10,000 line program, every little bit helps!

To understand the other advantages and disadvantages of initializers, you have to keep the various storage classes in mind. In the sample program you just saw, the variable i has auto extent, which means that it is created when the function is called, and goes away when the function returns to the caller. Naturally, this means that the variable will have to be initialized each time the function is called. With a simple variable like this integer, the compiler generates exactly the same program if you initialized the variable with a separate assignment statement, like this:

```
#include <stdio.h>

void main (void)

{
int i;

i = 4;
```

```
printf("%d\n", i);
}
```

Local static variables and all global variables, on the other hand, are allocated space by the compiler, and this space is reserved when your program is loaded into memory. These variables exist until your program finishes. In these cases, an initialized variable is only initialized one time. There are both advantages and disadvantages to this. The advantage is that your program is actually smaller and runs faster if you use an initializer as compared to initializing the variable with an assignment statement. The other interesting effect could be an advantage or a disadvantage, depending on your outlook. Since the variable is only initialized one time, if you change the value of the variable in your program, it stays changed. Try to apply these ideas to the following program, and predict what it will write. After you decide what you think it will write, give it a try.

```
#include <stdio.h>

int c = 3;

void fn(void)

{
static int a = 1;
int b = 2;

printf ("%d %d %d\n", a, b, c);
a *= 2;
b *= 2;
c *= 2;
}

void main (void)

{
fn();
fn();
}
```

You can initialize any C variable except the parameter to a function, regardless of the type of the variable. Some of the initialization rules for complicated types are pretty messy, and we won't get into them in this introductory course, but a couple of points are worth mentioning.

First, when you initialize an array or struct, you specify the various values enclosed in braces, like this:

```
int a[3] = {1,2,3};
```

You don't have to initialize all of the values, either. If you only initialize part of an array or struct, the rest of the values are set to 0 for you. For example, if we only initialize two values, like this:

```
int a[3] = {1,2};
```

then a[2] (which is the third element of the array – remember?) is set to 0. Finally, you can actually specify the *size* of an array by initializing it! This is most useful with initialized strings, where you don't have to keep track of the number of characters in the string. When you use an initializer to set the length of an array, you just leave out the subscript, like this:

```
char str[] = "Sample";
```

You can use expressions in initializers, too. The rules get a little complicated, but you can almost always use any expression that evaluates to a constant, like this one:

```
#define SIZE 3
int i = 150 - 20*SIZE;
```

Problem 11.4. Use an initializer to initialize a char array to "Hello, world.\n". Set the length of the array using the initializer. Print the string with printf.

Problem 11.5. An interesting problem from probability theory makes an interesting graphics program, too. The problem is called the drunkard's walk: you start the drunkard off in the middle of a roof, and assume that he has an equal probability of moving in any direction. The problem is to figure out how long, on average, it takes to reach the edge. (In recent and less graphic times, the problem has also been called the random walk.)

Write a program to implement the drunkard's walk in the graphics window. Draw a box that extends 20 steps in each direction away from a staring point that is initialized to x = 150, y = 40, and use the random number generator to move the point one step in a random direction. Do this by animating the point, so the old position is erased after the point moves. Stop the program and print the number of steps after the point reaches the edge of the square. Use initializers and static variables wherever appropriate.

Be sure and check your program against the solution to see if you made the best use of static variables and initialized variables.

Lesson Eleven
Solutions to Problems

Solution to problem 11.1.

```
/* Break a long value up into words. */

#include <stdio.h>

void main(void)

{
union {                                 /* union to split a long */
   long l;
   struct {
      int i1, i2;
      } i;
   } cnv;

do {
   printf("long value: ");              /* get a long value */
   scanf("%ld", &cnv.l);
                                        /* print the components */
   printf("least significant word: %d\n", cnv.i.i1);
   printf("most significant word : %d\n\n", cnv.i.i2);
   }
while (cnv.l);                          /* loop if input is not 0 */
}
```

Solution to problem 11.2.

The commands needed are:

```
compile main.cc keep=main
link main graph keep=main
```

You do not need to recompile graph, since it has not changed since the last compile.

Solution to problem 11.3.

main.cc

```
void main(void)

{
loop(1,10);
doit();
}
```

a.cc

```
int start, stop;

void loop (int p1, int p2)

{
start = p1;
stop = p2;
}
```

b.cc

```
#include <stdio.h>

extern int start, stop;

static void loop (void)

{
int i;

for (i = start; i <= stop; ++i)
   printf("%d\n", i);
}


void doit(void)

{
loop();
}
```

Commands to compile and run the program:

```
compile main.cc keep=main
compile a.cc keep=a
compile b.cc keep=b
link main a b keep=main
main
```

Solution to problem 11.4.

```
#include <stdio.h>

void main(void)

{
char str[] = "Hello, world.\n";
```

```
printf(str);
}
```

Solution to problem 11.5.

```
/* Implement the drunkard's walk */

#include <stdio.h>
#include <stdlib.h>

#include <quickdraw.h>

#define WIDTH 3                          /* width of a pixel */

void main(void)

{
static unsigned left = 150 - 20*WIDTH;  /* dimensions of the walk area */
static unsigned right = 150 + 20*WIDTH;
static unsigned top = 40 - 20;
static unsigned bottom = 40 + 20;

static unsigned x = 150, y = 40;        /* starting position */
static tx, ty;                          /* temp x, y (for animation) */

static int steps = 0;                   /* # steps taken */
static rval;                            /* number generated by rand() */

srand(12345);                           /* initialize rand() */
SetPenSize(3,1);                        /* set up the graphics screen */
SetSolidPenPat(0);                      /* plot the initial point */
MoveTo(x, y);
LineTo(x, y);
MoveTo(left, top);                      /* draw the confining box */
LineTo(right, top);
LineTo(right, bottom);
LineTo(left, bottom);
LineTo(left, top);

do {
   tx = x;                              /* save the current point */
   ty = y;
   rval = rand() % 3;                   /* update x */
   if (rval == 0)
      x -= WIDTH;
   else if (rval == 2)
      x += WIDTH;
   rval = rand() % 3;                   /* update y */
   if (rval == 0)
      --y;
```

```
   else if (rval == 2)
      ++y;
   SetSolidPenPat(3);                  /* erase the old point */
   MoveTo(tx, ty);
   LineTo(tx, ty);
   SetSolidPenPat(0);                  /* draw the new point */
   MoveTo(x, y);
   LineTo(x, y);
   ++steps;                            /* update the step count */
   }                                   /* loop if not at the edge */
while ((x != left) && (x != right) && (y != top) && (y != bottom));

printf("The walk took %d steps.\n", steps); /* write the results */
}
```

Lesson Twelve
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/C, 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/C runs an S16 program, it runs it the same way the Finder does: ORCA/C shuts itself down, then runs the program. Once your program finishes, the Apple IIGS's operating system returns to ORCA/C. When ORCA/C runs an EXE program, it does not shut down. Because ORCA/C is still there, you can see your source file, use the debugger, and take advantage of the fact that ORCA/C 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/C 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/C 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 C is not particularly easy, so ORCA/C has a function that will initialize QuickDraw for you. The function is called startgraph. When you start a tool, you need to shut it down, too. ORCA/C has a function called endgraph to do that. Both of these are declared in orca.h, so you need to put a

```
#include <orca.h>
```

at the start of your program when you are going to use these functions.

The startgraph function 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.

The startgraph function 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.

The startgraph function does a couple of other things that are a bit different than what you are used to: it 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 12.1, 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.

When you run this program, one of the peculiar things that happens is that the debugger's menu bar vanishes, to be replaced by a blank menu bar with a foot and a double-arrow. Any time you initialize any tools for yourself (the startgraph function initializes QuickDraw) the debugger assumes you are trying to

Listing 12.1

```
/* Draw an X across the screen */

#include <quickdraw.h>
#include <orca.h>
#include <stdio.h>


/* Standard graphics initialization. */

void InitGraphics (void)

{
Rect r;

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 */
}


/* main program */

void main (void)

{
Rect r;                       /* size of the graphics area */

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);
getchar();                    /* wait for the user to press RETURN */
endgraph();                   /* shut down QuickDraw */
}
```

write a desktop program with its own menu bar, and it sets up a new one for you. You can flip back to the debugger's menu bar by clicking on the arrow, or you can single step without switching to the debugger's menu bar by clicking on the foot (to step, of course!). For the most part, you can ignore this situation, since the debugger's menu bar will come back automatically as soon as your program finishes executing. If you would like to know more about what is happening, you can refer to the ORCA/C reference manual's description of the debugger.

There are a couple of new tool calls in this program which could be useful to you in your own programs, so we'll stop now and take a look at them. In the InitGraphics function, you see the declaration

```
Rect r;
```

A Rect is a structure defined in the toolbox, and it is one of the basic structures used by QuickDraw. Inside this structure, which you can find in the types.h header file in the ORCACDEFS folder, there are four fields: h1, h2, v1 and v2. These integer fields mark the left, right, top and bottom edges of a rectangle on the graphics screen.

The InitGraphics function makes use of the rectangle to find the size of the graphics window with the call

```
GetPortRect(&r);
```

This call fills in the rectangle with the size of the current graphics port. When you run the program from ORCA/C, the rectangle is filled in with the size of the graphics window, even if you have moved it or changed its size. The way we use the graphics window, the values of h1 and v1 will always be zero, too, so r.h2 is the width of the graphics window, and r.v2 is the height of the graphics window. Later, when you run this program as a stand-alone program, this rectangle will get set to the size of the entire graphics screen.

This information is put to use in the PaintRect call, which fills in the rectangle with the current pen color. This is a fast way to paint the entire graphics screen.

There are two steps involved in making this a stand-alone program, and 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/C 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. 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 C 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 12.1. 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.

Desktop Programs

Writing full-blown desktop programs with pull-down menus, windows, and so forth is something that needs a course all to itself, but it is worth pointing out that once you know how to create desktop programs,

you make them stand-alone the same way you make a graphics program a stand-alone program.

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 printf or scanf 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 printf and scanf?

The answer, fortunately, is that ORCA/C 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: the 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.

The program in listing 12.2 gives an example of some of the effects you can get with these colors. Give it a try as an EXE program.

There is one important difference between the way this program handles the ends of lines and the way most of your other programs handled the end of a line. When you are writing to the text screen, the character \n causes the text to shift to the start of the next line. On the graphics screen, you should use \r, instead, as this program does.

Problem 12.2. 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 getchar() 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.

Listing 12.2

```
/* Exploring text mixed with graphics. */

#include <quickdraw.h>
#include <orca.h>
#include <stdio.h>


/* Standard graphics initialization. */

void InitGraphics (void)

{
Rect r;

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 */
}


/* main program */

void main (void)

{
InitGraphics();              /* set up the graphics screen */
SetForeColor(0);             /* write some text */
SetBackColor(3);
MoveTo(10,10);
printf("You can get\rsome ");
SetForeColor(3);
SetBackColor(0);
printf("interesting effects\r");
SetForeColor(2);
SetBackColor(3);
printf("by mixing text\r");
SetForeColor(1);
printf("with graphics!");
endgraph();                  /* shut down QuickDraw */
}
```

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

```
#include <texttool.h>
```

at the start of your program.

```
/* Start the text tools for    */
/* a stand-alone text program. */

void StartText (void)

{
TextStartUp();
SetErrGlobals(127,0);
SetErrorDevice(1,3L);
SetInGlobals(127,0);
SetInputDevice(1,3L);
SetOutGlobals(127,0);
SetOutputDevice(1,3L);
}

/* Shut down the text tools    */
/* for a stand-alone text      */
/* program.                    */

void EndText (void)

{
TextShutDown();
}
```

Problem 12.3. 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.

Lesson Twelve
Solutions to Problems

Solution to problem 12.1.

```
/* Do a random walk with 10 random shapes */

#include <stdlib.h>
#include <orca.h>

#include <quickdraw.h>

#define NUMSHAPES 10                    /* # of shapes to animate */
#define WALKLENGTH 100                  /* # of "steps" in the walk */

#define MAXX 320                        /* size of the graphics screen */
#define MAXY 200

typedef enum shapeKind {triangle, square, star} shapeKind;

typedef struct shapeType {              /* information about one shape */
   int color;                           /* color */
   shapeKind kind;                      /* kind of shape */
   union {
      struct {                          /* points for a triangle */
         int x1,x2,x3,y1,y2,y3;
         } t;
      struct {                          /* points for a square */
         int x1,x2,x3,x4,y1,y2,y3,y4;
         } s;
      struct {                          /* points for a star */
         int x1,x2,x3,x4,x5;
         int y1,y2,y3,y4,y5;
         } p;
      } coord;
   } shapeType;


shapeType shapes[NUMSHAPES];            /* current array of shapes */
shapeType oldShapes[NUMSHAPES];         /* shapes in last position */
```

```
void InitGraphics (void)

/* Standard graphics initialization.                          */

{
Rect r;                                 /* graphics screen */

startgraph(320);                        /* set up the graphics screen */
SetPenMode(0);                          /* pen mode = copy */
SetPenSize(1,1);                        /* use a square pen */
SetSolidPenPat(15);                     /* paint the screen white */
GetPortRect(&r);
PaintRect(&r);
SetPenMode(2);                          /* pen mode = xor */
}


void DrawShape (shapeType s)

/* This subroutine draws one of the shapes on the screen.     */
/*                                                            */
/* Parameters:                                                */
/*    s - shape to draw                                       */

{
SetSolidPenPat(s.color);            /* set the pen color for the shape */

switch (s.kind) {

   case triangle:                       /* draw a triangle */
      MoveTo(s.coord.t.x1, s.coord.t.y1);
      LineTo(s.coord.t.x2, s.coord.t.y2);
      LineTo(s.coord.t.x3, s.coord.t.y3);
      LineTo(s.coord.t.x1, s.coord.t.y1);
      return;

   case square:                         /* draw a square */
      MoveTo(s.coord.s.x1, s.coord.s.y1);
      LineTo(s.coord.s.x2, s.coord.s.y2);
      LineTo(s.coord.s.x4, s.coord.s.y4);
      LineTo(s.coord.s.x3, s.coord.s.y3);
      LineTo(s.coord.s.x1, s.coord.s.y1);
      return;
```

```
   case star:                          /* draw a star */
      MoveTo(s.coord.p.x1, s.coord.p.y1);
      LineTo(s.coord.p.x2, s.coord.p.y2);
      LineTo(s.coord.p.x3, s.coord.p.y3);
      LineTo(s.coord.p.x4, s.coord.p.y4);
      LineTo(s.coord.p.x5, s.coord.p.y5);
      LineTo(s.coord.p.x1, s.coord.p.y1);
      return;
   }
}


void CreateShape (shapeType *s)

/* 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                                       */

{
int cx,cy;                             /* center point for the shape */

s->color = rand() % 15 + 1;            /* get a color */
cx = rand() % (MAXX - 19) + 9;         /* get the center position,   */
cy = rand() % (MAXY - 8) + 4;          /* picking the points so the  */
                                       /* shape is in the window     */

switch (rand() % 3) {                  /* set the initial positions */
   case 0:                             /* set up a triangle */
      s->kind = triangle;
      s->coord.t.x1 = cx-10;
      s->coord.t.y1 = cy+4;
      s->coord.t.x2 = cx;
      s->coord.t.y2 = cy-8;
      s->coord.t.x3 = cx+10;
      s->coord.t.y3 = cy+4;
      return;

   case 1:                             /* set up a square */
      s->kind = square;
      s->coord.s.x1 = cx-8;
      s->coord.s.y1 = cy-6;
      s->coord.s.x2 = cx+8;
      s->coord.s.y2 = cy-6;
      s->coord.s.x3 = cx-8;
      s->coord.s.y3 = cy+6;
      s->coord.s.x4 = cx+8;
      s->coord.s.y4 = cy+6;
```

```
      return;

   case 2:                              /* set up a star */
      s->kind = star;
      s->coord.p.x1 = cx-7;
      s->coord.p.y1 = cy+7;
      s->coord.p.x2 = cx;
      s->coord.p.y2 = cy-8;
      s->coord.p.x3 = cx+7;
      s->coord.p.y3 = cy+7;
      s->coord.p.x4 = cx-11;
      s->coord.p.y4 = cy-3;
      s->coord.p.x5 = cx+11;
      s->coord.p.y5 = cy-3;
      return;
   }
}


void UpdateShape (shapeType *s)

/* This subroutine moves a shape across the screen in a random  */
/* walk.                                                        */
/*                                                              */
/* Parameters:                                                  */
/*    s - shape to update                                       */

{
int dx,dy;                              /* movement direction */

dx = rand() % 3 - 1;                    /* get the walk direction */
dy = rand() % 3 - 1;

switch (s->kind) {                      /* make sure we don't walk off */
                                        /* of the screen, then update  */
                                        /* the position                */
```

```
case triangle:                        /* check a triangle */
   if (dx == -1)
      if (s->coord.t.x1 < 1)
         dx = 0;
   if (dx == 1)
      if (s->coord.t.x3 >= MAXX)
         dx = 0;
   if (dy == -1)
      if (s->coord.t.y2 < 1)
         dy = 0;
   if (dy == 1)
      if (s->coord.t.y3 >= MAXY)
         dy = 0;
   s->coord.t.x1 = s->coord.t.x1+dx; /* update a triangle */
   s->coord.t.y1 = s->coord.t.y1+dy;
   s->coord.t.x2 = s->coord.t.x2+dx;
   s->coord.t.y2 = s->coord.t.y2+dy;
   s->coord.t.x3 = s->coord.t.x3+dx;
   s->coord.t.y3 = s->coord.t.y3+dy;
   return;

case square:                          /* check a square */
   if (dx == -1)
      if (s->coord.s.x1 < 1)
         dx = 0;
   if (dx == 1)
      if (s->coord.s.x2 >= MAXX)
         dx = 0;
   if (dy == -1)
      if (s->coord.s.y1 < 1)
         dy = 0;
   if (dy == 1)
      if (s->coord.s.y3 >= MAXY)
         dy = 0;
   s->coord.s.x1 = s->coord.s.x1+dx; /* update a square */
   s->coord.s.y1 = s->coord.s.y1+dy;
   s->coord.s.x2 = s->coord.s.x2+dx;
   s->coord.s.y2 = s->coord.s.y2+dy;
   s->coord.s.x3 = s->coord.s.x3+dx;
   s->coord.s.y3 = s->coord.s.y3+dy;
   s->coord.s.x4 = s->coord.s.x4+dx;
   s->coord.s.y4 = s->coord.s.y4+dy;
   return;
```

```
   case star:                          /* check a star */
      if (dx == -1)
         if (s->coord.p.x4 < 1)
            dx = 0;
      if (dx == 1)
         if (s->coord.p.x5 >= MAXX)
            dx = 0;
      if (dy == -1)
         if (s->coord.p.y2 < 1)
            dy = 0;
      if (dy == 1)
         if (s->coord.p.y1 >= MAXY)
            dy = 0;
      s->coord.p.x1 = s->coord.p.x1+dx; /* update a star */
      s->coord.p.y1 = s->coord.p.y1+dy;
      s->coord.p.x2 = s->coord.p.x2+dx;
      s->coord.p.y2 = s->coord.p.y2+dy;
      s->coord.p.x3 = s->coord.p.x3+dx;
      s->coord.p.y3 = s->coord.p.y3+dy;
      s->coord.p.x4 = s->coord.p.x4+dx;
      s->coord.p.y4 = s->coord.p.y4+dy;
      s->coord.p.x5 = s->coord.p.x5+dx;
      s->coord.p.y5 = s->coord.p.y5+dy;
      return;
   }
}


void main(void)

/* main program                                                  */

{
int i,j;                               /* loop variables */

InitGraphics();                        /* set up the graphics window */
srand(6289);                           /* initialize rand() */

for (i = 0; i < NUMSHAPES; ++i) { /* set up and draw the initial shapes */
   CreateShape(&shapes[i]);
   DrawShape(shapes[i]);
   }
```

```
for (i = 0; i < WALKLENGTH; ++i) {      /* do the random walk */
   for (j = 0; j < NUMSHAPES; ++j) {    /* move the shapes */
      oldShapes[j] = shapes[j];
      UpdateShape(&shapes[j]);
      }
   for (j = 0; j < NUMSHAPES; ++j) {    /* redraw the shapes */
      DrawShape(shapes[j]);
      DrawShape(oldShapes[j]);
      }
   }

endgraph();                            /* shut down the graphics screen */
}
```

Solution to problem 12.2.

```
/* Exploring text mixed with graphics. */

#include <quickdraw.h>
#include <orca.h>
#include <stdio.h>


/* Standard graphics initialization. */

void InitGraphics (void)

{
Rect r;

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 */
}
```

```
/* main program */

void main (void)

{
InitGraphics();             /* set up the graphics screen */
SetForeColor(0);            /* write some text */
SetBackColor(3);
MoveTo(10,10);
printf("You can get\rsome ");
SetForeColor(3);
SetBackColor(0);
printf("interesting effects\r");
SetForeColor(2);
SetBackColor(3);
printf("by mixing text\r");
SetForeColor(1);
printf("with graphics!");
                            /* wait for the user to press RETURN */
printf("\r\r\rPress RETURN to quit.");
getchar();
endgraph();                 /* shut down QuickDraw */
}
```

Solution to problem 12.3.

```
/* 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.                                      */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <texttool.h>

#define MAXWORDS 10             /* possible words */
#define MAXCHARS 8              /* number of characters in each word */

char words[MAXWORDS][MAXCHARS+1]; /* word array */
```

```
void FillArray (void)

/* Fill the word array.                                    */
/*                                                         */
/* Variables:                                              */
/*    words - word array                                   */

{
strcpy(words[0], "computer");
strcpy(words[1], "whale");
strcpy(words[2], "megabyte");
strcpy(words[3], "modem");
strcpy(words[4], "chip");
strcpy(words[5], "online");
strcpy(words[6], "disk");
strcpy(words[7], "monitor");
strcpy(words[8], "window");
strcpy(words[9], "keyboard");
}


void GetSeed (void)

/* Initialize the random number generator             */

{
int val;                        /* seed value */

printf("Please enter a random number seed:");
scanf(" %d", &val);
srand(val);
}


unsigned RandomValue (unsigned max)

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

{
return rand() % max + 1;
}
```

```
void Play (void)

/* Play a game of hangman.                              */
/*                                                      */
/* Variables:                                           */
/*    words - word array                                */

{
int allFound;                   /* used to test for unknown chars */
char ch;                        /* character from player */
int done;                       /* is the game over? */
char found[MAXCHARS];           /* characters found by player */
unsigned len;                   /* length of word; for efficiency */
unsigned i;                     /* loop variable */
int inString;                   /* is ch in the string? */
char word[MAXCHARS+1];          /* word to guess */
int wrong;                      /* number of wrong guesses */

for (i = 0; i < MAXCHARS; ++i)  /* no letters guessed, so far */
   found[i] = 0;
wrong = 0;                      /* no wrong guesses, yet */
done = 0;                       /* the game is not over, yet */
                                /* pick a word */
strcpy(word, words[RandomValue(MAXWORDS)]);
len = strlen(word);             /* record the length of the word */
do {
   printf("\nThe word is: \""); /* write the word */
   for (i = 0; i < len; ++i)
      if (found[i])
         printf("%c", word[i]);
      else
         printf("-");
   printf("\"\nGuess a character:"); /* get the player's choice */
   scanf(" %c", &ch);
   inString = 0;                /* see if ch is in the string */
   for (i = 0; i < len; ++i)
      if (word[i] == ch) {
         found[i] = 1;
         inString = 1;
         }
   if (inString)                /* handle a correct guess */
      printf("%c is in the string.\n", ch);
```

```
   else {                        /* handle an incorrect guess */
      printf("%c is not in the string.\n", ch);
      ++wrong;                   /* one more wrong answer... */
      printf("Your ");           /* tell the player how they are doing */
      if (wrong == 1)
         printf("head");
      else if (wrong == 2)
         printf("body");
      else if (wrong == 3)
         printf("left arm");
      else if (wrong == 4)
         printf("right arm");
      else if (wrong == 5)
         printf("left leg");
      else /* if (wrong == 6) */
         printf("right leg");
      printf(" is now in the noose!\n");
      }

   if (wrong == 6) {             /* see if the player is hung */
      printf("\n\nSorry, Jack Ketch got you!\nThe word was \"%s\".\n",
         word);
      done = 1;
      }
   allFound = 1;                 /* check for unknown characters */
   for (i = 0; i < len; ++i)
      if (!found[i])
         allFound = 0;
   if (allFound) {               /* see if the player got the word */
      printf("You got it!  The word is \"%s\".\n", word);
      done = 1;
      }
   }
while (!done);
}


int PlayAgain (void)

/* See if the player wants to play another game.      */
/*                                                    */
/* Returns:                                           */
/*    True to play again, false to quit.              */
```

```
{
char ch;                        /* player's response */

printf("\n\n");
do {
   printf("Would you like to play again (y or n)?");
   scanf(" %c", &ch);
   }
while ((ch != 'y') && (ch != 'Y') && (ch != 'n') && (ch != 'N'));
return (ch == 'y') || (ch == 'Y');
}


void StartText (void)

/* Start the text tools for a stand-alone text        */
/* program.                                           */

{
TextStartUp();
SetErrGlobals(127, 0);
SetErrorDevice(1, 3L);
SetInGlobals(127, 0);
SetInputDevice(1, 3L);
SetOutGlobals(127, 0);
SetOutputDevice(1, 3L);
}


void EndText (void)

/* Shut down the text tools for a stand-alone text    */
/* program.                                           */

{
TextShutDown();
}
```

```
void main (void)

/* Main program                                        */

{
StartText();                    /* start the text toolbox */
FillArray();                    /* fill the word array */
GetSeed();                    /* initialize the random number generator */
do
   Play();                      /* play a game */
while (PlayAgain());            /* loop if he wants to play again */
EndText();                      /* shut down the text tool set */
}
```

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 twelve lessons were concerned primarily with teaching you the mechanics of programming. In those lessons, you learned most of the features of C. While we used a number of real programs to illustrate the features of the C language, and frequently discussed principals of good programming practice, crafting a program was not the primary topic.

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 C 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 C 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 C program, and see how a scanner would break it up into words.

```
#include <stdio.h>

void main(void)

{
printf("Hello, world.\n");
}
```

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 void and); constants, like the string written by printf; and identifiers, like main. 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 C compiler knows that void starts a function, 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 C compiler can't afford to skip commas or numbers, but it can skip comments, spaces, and end of line marks. In 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.

```
void GetToken(void)

/* Read a word from the source file    */
/*                                     */
/* Variables:                          */
/*    f - source file                  */
/*    token - string read              */

{
/* length of the string */
int len = 0;

/* skip to the first character */
while ((! isalpha(ch)) && (ch != EOF))
   ch = fgetc(f);

/* read the word */
while ((ch != EOF) && (isalpha(ch))) {
   if (len < MAXLENGTH-1) {
      token[len] = ch;
      ++len;
      }
   ch = fgetc(f);
   }

/* set the null terminator */
token[len] = (char) 0;
}
```

Take a close look at how the subroutine works. Think it through by writing the values of len, ch 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; it is defined in the main part of the program. For this program, MAXLENGTH is set to 30 instead of 29, leaving space for the terminating null character.)

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.

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 structure called symbolStruct, which defines a single entry in the symbol table. This struct is defined globally, so that we can also use a global variable to point to the first element of the linked list. The structure looks like this:

```
typedef struct symbolStruct {
   struct symbolStruct *next;
   int count;
   char symbol[MAXLENGTH];
   }
   symbolStruct;

typedef symbolStruct *symbolPtr;
```

The function that actually inserts a value into the symbol table is shown in listing 13.1.

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 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.

Listing 13.1

```
void Insert (void)

/* Insert a word in the symbol table                                  */
/*                                                                    */
/* Variables:                                                         */
/*    token - symbol                                                  */
/*    table - pointer to the first element in the symbol              */
/*       table                                                        */

{
symbolPtr ptr;                        /* work pointer */
int len;                              /* length of the string */

ptr = table;                         /* see if the symbol already exists */
while (ptr != NULL) {
   if (strcmp(ptr->symbol, token) == 0) {
      ++ptr->count;                  /* yes -> update the count and exit */
      return;
      }
   ptr = ptr->next;
   }
                                      /* no -> create a new entry */
ptr = (symbolPtr) malloc(sizeof(symbolStruct));
ptr->next = table;
table = ptr;
ptr->count = 1;
strcpy(ptr->symbol, token);
}
```

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. The parser is shown in listing 13.2. 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.

Listing 13.2

```
                                        /* command subjects */
typedef enum {none, up, down, left, right} subjectType;
typedef enum {nada, go, stop} verbType; /* commands */

void GetAction (void)

/* Find out what the player wants to do                          */
/*                                                               */
/* Variables:                                                    */
/*    verb - action to take                                      */
/*    subject - what we do the action to or with                 */

{
int done = 0;                       /* loop exit variable */

do {
   scanf("%255[^\n]%*1[\n]", cline); /* get a line */
   lineIndex = 0;
   verb = nada;                     /* start with no subject,verb */
   subject = none;
   do {
      GetToken();                   /* get a token */
                                    /* handle a subject */
      if ((strcmp(token, "NORTH") == 0)
         || (strcmp(token, "UP") == 0))
         subject = up;
      else if ((strcmp(token, "SOUTH") == 0)
         || (strcmp(token, "DOWN") == 0))
         subject = down;
      else if ((strcmp(token, "EAST") == 0)
         || (strcmp(token, "RIGHT") == 0))
         subject = right;
      else if ((strcmp(token, "WEST") == 0)
         || (strcmp(token, "LEFT") == 0))
         subject = left;
                                    /* handle a verb */
      else if ((strcmp(token, "QUIT") == 0)
         || (strcmp(token, "STOP") == 0))
         verb = stop;
      else if ((strcmp(token, "GO") == 0)
         || (strcmp(token, "MOVE") == 0))
         verb = go;
      }
   while (strlen(token));
```

```
      switch (verb) {                /* make sure the input is consistent */
         case nada:
            printf("Please tell me what to do (go or stop).\n");
            break;

         case stop:
            done = 1;
            break;

         case go:
            if (subject == none)
               printf("Please tell me which way to go.\n");
            else
               done = 1;
            break;
         }
      }
   while (!done);
   }
```

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:

```
do
   GetAction();
   if (verb == go) {
      DrawPoint(x,y,3);
      switch (subject) {
         case up:
            y -= MOVEY;
            break;
         case down:
            y += MOVEY;
            break;
         case left:
            x -= MOVEX;
            break;
         case right:
            x += MOVEX;
            break;
         }
      DrawPoint(x,y,0);
      }
   }
while (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.

Lesson Thirteen
Solutions to Problems

Solution to problem 13.1.

```
/* Write the words in a file */

#include <stdio.h>
#include <ctype.h>

#define MAXLENGTH 30                    /* length of a word */
#define FNAME "prob.13.1.cc"            /* file name to scan */

char token[MAXLENGTH];                  /* last word read */
char ch = ' ';                          /* last character read */
FILE *f;                                /* source file */


void GetToken(void)

/* Read a word from the source file                              */
/*                                                               */
/* Variables:                                                    */
/*    f - source file                                            */
/*    token - string read                                        */

{
int len = 0;                            /* length of the string */

while ((! isalpha(ch)) && (ch != EOF))  /* skip to the first character */
   ch = fgetc(f);
while ((ch != EOF) && (isalpha(ch))) {  /* read the word */
   if (len < MAXLENGTH-1) {
      token[len] = ch;
      ++len;
      }
   ch = fgetc(f);
   }
token[len] = (char) 0;                  /* set the null terminator */
}
```

```
void main(void)

/* main program                                                    */

{
f = fopen(FNAME, "r");                  /* open the file */
if (f == NULL) {
   printf("Could not open %s.\n", FNAME);
   return;
   }
do {                                    /* scan the file */
   GetToken();
   if (strlen(token))
      printf("%s\n", token);
   }
while (strlen(token));
fclose(f);                              /* close the file */
}
```

Solution to problem 13.2.

```
/* Write the number of times a word occurs in a file */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXLENGTH 30                    /* length of a word */
#define FNAME "prob.13.2.cc"            /* file name to scan */

typedef struct symbolStruct {           /* symbol table entry */
   struct symbolStruct *next;           /* ptr to next entry */
   int count;                           /* # of occurrences */
   char symbol[MAXLENGTH];              /* symbol */
   }
   symbolStruct;

typedef symbolStruct *symbolPtr;        /* ptr to a symbol table entry */

char token[MAXLENGTH];                  /* last word read */
char ch = ' ';                          /* last character read */
symbolPtr table = NULL;                 /* the symbol table */
FILE *f;                                /* source file */
```

```
void GetToken(void)

/* Read a word from the source file                                  */
/*                                                                   */
/* Variables:                                                        */
/*    f - source file                                                */
/*    token - string read                                            */

{
int len = 0;                            /* length of the string */

while ((! isalpha(ch)) && (ch != EOF))  /* skip to the first character */
   ch = fgetc(f);
while ((ch != EOF) && (isalpha(ch))) {  /* read the word */
   if (len < MAXLENGTH-1) {
      token[len] = ch;
      ++len;
      }
   ch = fgetc(f);
   }
token[len] = (char) 0;                  /* set the null terminator */
}


void Insert (void)

/* Insert a word in the symbol table                                 */
/*                                                                   */
/* Variables:                                                        */
/*    token - symbol                                                 */
/*    table - pointer to the first element in the symbol             */
/*       table                                                       */

{
symbolPtr ptr;                          /* work pointer */
int len;                                /* length of the string */

ptr = table;                         /* see if the symbol already exists */
while (ptr != NULL) {
   if (strcmp(ptr->symbol, token) == 0) {
      ++ptr->count;                  /* yes -> update the count and exit */
      return;
      }
   ptr = ptr->next;
   }
```

```
                                        /* no -> create a new entry */
ptr = (symbolPtr) malloc(sizeof(symbolStruct));
ptr->next = table;
table = ptr;
ptr->count = 1;
strcpy(ptr->symbol, token);
}


void PrintSymbols (void)

/* Print the symbol table                                          */
/*                                                                 */
/* Variables:                                                      */
/*    table - pointer to the first element in the symbol           */
/*       table                                                     */

{
symbolPtr ptr;                          /* work pointer */

ptr = table;
while (ptr != NULL) {
   printf("%10d  %s\n", ptr->count, ptr->symbol);
   ptr = ptr->next;
   }
}


void main(void)

/* main program                                                    */

{
f = fopen(FNAME, "r");                  /* open the file */
if (f == NULL) {
   printf("Could not open %s.\n", FNAME);
   return;
   }
do {                                    /* scan the file */
   GetToken();
   if (strlen(token))
      Insert();
   }
while (strlen(token));
PrintSymbols();                         /* print the symbol table */
fclose(f);                              /* close the file */
}
```

Solution to problem 13.3.

```
/* Move a "robot" around on the graphics screen */

#include <stdio.h>
#include <ctype.h>
#include <string.h>

#include <quickdraw.h>

#define MAXLENGTH 10                    /* length of a word */
#define MAXLINE 256                     /* max length of a line */
#define MOVEX 30                        /* distance of one move */
#define MOVEY 10

                                        /* command subjects */
typedef enum {none, up, down, left, right} subjectType;
typedef enum {nada, go, stop} verbType; /* commands */

char cline[MAXLINE];                    /* command line */
int lineIndex;                          /* index of next char in line */
char token[MAXLENGTH];                  /* last word read */

subjectType subject;                    /* subject of the last command */
verbType verb;                          /* verb of the last command */


void GetToken(void)

/* Read a word from the command line                          */
/*                                                            */
/* Variables:                                                 */
/*    f - source file                                         */
/*    cline - source line                                     */
/*    lineIndex - index to next char in source line           */

{
int len = 0;                            /* length of the string */

                                        /* skip to the first character */
while ((! isalpha(cline[lineIndex])) && (lineIndex <= strlen(cline)))
   ++lineIndex;
                                        /* read the word */
while ((lineIndex <= strlen(cline)) && (isalpha(cline[lineIndex]))) {
   if (len < MAXLENGTH-1) {
      token[len] = toupper(cline[lineIndex]);
      ++len;
      }
   ++lineIndex;
   }
```

```
token[len] = (char) 0;                  /* set the null terminator */
}


void GetAction (void)

/* Find out what the player wants to do                          */
/*                                                               */
/* Variables:                                                    */
/*    verb - action to take                                      */
/*    subject - what we do the action to or with                 */

{
int done = 0;                           /* loop exit variable */

do {
   scanf("%255[^\n]%*1[\n]", cline);    /* get a line */
   lineIndex = 0;
   verb = nada;                         /* start with no subject,verb */
   subject = none;
   do {
      GetToken();                       /* get a token */
                                        /* handle a subject */
      if ((strcmp(token, "NORTH") == 0)
         || (strcmp(token, "UP") == 0))
         subject = up;
      else if ((strcmp(token, "SOUTH") == 0)
         || (strcmp(token, "DOWN") == 0))
         subject = down;
      else if ((strcmp(token, "EAST") == 0)
         || (strcmp(token, "RIGHT") == 0))
         subject = right;
      else if ((strcmp(token, "WEST") == 0)
         || (strcmp(token, "LEFT") == 0))
         subject = left;
                                        /* handle a verb */
      else if ((strcmp(token, "QUIT") == 0)
         || (strcmp(token, "STOP") == 0))
         verb = stop;
      else if ((strcmp(token, "GO") == 0)
         || (strcmp(token, "MOVE") == 0))
         verb = go;
      }
   while (strlen(token));
```

```
   switch (verb) {                /* make sure the input is consistent */
      case nada:
         printf("Please tell me what to do (go or stop).\n");
         break;

      case stop:
         done = 1;
         break;

      case go:
         if (subject == none)
            printf("Please tell me which way to go.\n");
         else
            done = 1;
         break;
      }
   }
while (!done);
}


void DrawPoint (int x, int y, int color)

/* Draw the robot                                                  */
/*                                                                 */
/* Parameters:                                                     */
/*    x,y - position of the robot                                  */
/*    color - robot color                                          */

{
SetSolidPenPat(color);
SetPenMode(0);
SetPenSize(9,3);
MoveTo(x,y);
LineTo(x,y);
}
```

```
void main(void)

/* main program                                                    */

{
int x = 60, y = 20;                    /* position of the robot */

DrawPoint(x, y, 0);
do {
   GetAction();                 /* find out what we are supposed to do */
   if (verb == go) {                   /* if it is a movement then move */
      DrawPoint(x, y, 3);               /* erase the old robot */
      switch (subject) {                /* move the robot */
         case up:       y -= MOVEY;
                        break;
         case down:     y += MOVEY;
                        break;
         case left:     x -= MOVEX;
                        break;
         case right:    x += MOVEX;
                        break;
         }
      DrawPoint(x, y ,0);             /* draw the robot in the new spot */
      }
   }
while (verb != stop);
}
```

Lesson Fourteen
Recursion

A Quick Look at Recursion

By now, you are well acquainted with defining and calling functions. An interesting point about functions that we haven't talked about, and that you may not have noticed, is that a function can call itself. After all, the definition of the function comes before the statements, so the function has been defined. The ability of a function 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 Functions Call Themselves

Let's start by looking at a short program. This program multiplies two positive integers.

```
#include <stdio.h>

int Mult (int x, int y)

{
if (y)
   return Mult(x, y-1) + x;
return 0;
}

void main(void)

{
printf("%d\n", Mult(4,5));
}
```

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:

```
return Mult(x, y-1) + x;
```

This statement returns x*y, but it does it in a rather strange way. If you think about it, you know that x*y is the same thing as x*(y-1)+x; for example, 4*5 is the same thing as 4*4+4. The program is using the fact to gradually make the problem simpler. It calls itself to find out what 4*4 is, and then adds 4 and returns the answer.

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 function is called; the stack frame contains all of the auto variables in the function. 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, do 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 C function, we get something like this:

```
void Tower (int fromPeg,
            int toPeg,
            int sparePeg,
            int count);

{
if (count) {
   Tower(fromPeg, sparePeg,
      toPeg, count-1);
   Move(fromPeg, toPeg);
   Tower(sparePeg, toPeg,
      fromPeg, count-1);
   }
}
```

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; i <= 10; ++i) {
   ptr = (listPtr)
      malloc(sizeof(listStruct));
   ptr->next = list;
   ptr->value = i;
   list = ptr;
   }
```

Next, write a recursive function that prints the values in the list. On each call, the recursive function should return if the pointer that is passed to it is NULL. If the pointer is not NULL, the function should call itself, then print the current value, like this:

```
Print(ptr->next);
printf("%d\n", 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 C, 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.

```
int Expression (void)

{
int value, newValue;
tokenType operation;

value = Factor();
while ((token == add)
   || (token == subtract)) {
   operation = token;
   GetToken();
   newValue = Factor();
   if (operation == add)
      value += newValue;
   else
      value -= newValue;
   }
return value;
}
```

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.

```
int Factor (void)

{
int value, newValue;
tokenType operation;
```

```
value = Term();
while ((token == multiply)
   || (token == divide)) {
   operation = token;
   GetToken();
   newValue = Term();
   if (operation == multiply)
      value *= newValue;
   else
      value /= newValue;
   }
return value;
}
```

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.

```
int Term (void)

{
int val;

if (token == integer) {
   val = tokenValue;
   GetToken();
   }
else if (token == lparen) {
   GetToken();
   val = Expression();
   if (token == rparen)
      GetToken();
   else
      printf(") expected\n");
   }
return val;
}
```

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/C manual. Now you know where those error messages come from!

There is one minor detail that you need to deal with at this point. Looking at our functions, you can see that Expression calls Factor, so it is natural to put Factor before Expression so the compiler can check our parameter lists. Factor calls Term, so term should go before Factor. Term calls Expression, so... oops.

The problem, of course, is that you can't put these functions in any order where each function is declared before it is used. To solve this problem, we use exactly the same technique that we used with separate compilation: we declare the function Expression before the function Factor, like this:

```
int Expression (void);
```

but we still put the body of the Expression function after Factor and Term. That way, the compiler can check to be sure Expression is called correctly in Term, and it can even check to be sure the parameters for the function definition (the place we put the statements) match the parameters for the function declaration (the line you just saw).

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.

```
#include <stdio.h>

int Exp (int x, int y)

{
if (y)
   return Exp(x, y-1) * x;
return 1;
}

void main(void)

{
printf("%d\n", Exp(5,4));
}
```

Solution to problem 14.2.

```
/* Graphic solution to the Towers of Hanoi puzzle. */

#include <quickdraw.h>

#define DISKS 6                         /* # of disks to move */

typedef int peg[DISKS];                 /* one peg, with contents */

peg pegs[3];                            /* all three pegs */


void DrawDisk (int disk, int peg, int height, int color)

/* 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                                */
```

```
{
int x,y;                          /* position of the center of the disk */

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);
}


void Move (int fromPeg, int toPeg)

/* Move a disk                                                    */
/*                                                                */
/* Parameters:                                                    */
/*    fromPeg - peg to move the disk from                         */
/*    toPeg - peg to move the disk to                             */

{
int i;                                  /* peg array index */
int disk;                               /* disk being moved */

i = DISKS-1;                            /* find the disk to move */
while (pegs[fromPeg][i] == 0)
   --i;
disk = pegs[fromPeg][i];                /* remove the disk */
pegs[fromPeg][i] = 0;
DrawDisk(disk, fromPeg, i, 3);          /* erase the disk */
i = 0;                                 /* find the new spot for the disk */
while (pegs[toPeg][i] != 0)
   ++i;
pegs[toPeg][i] = disk;                  /* place the disk on the peg */
DrawDisk(disk, toPeg, i, 0);            /* draw the disk */
}


void Tower(int fromPeg, int toPeg, int sparePeg, int count)

/* 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                      */
```

```
{
if (count) {
   Tower(fromPeg, sparePeg, toPeg, count-1);
   Move(fromPeg, toPeg);
   Tower(sparePeg, toPeg, fromPeg, count-1);
   }
}


void main (void)

/* main program                                                    */

{
int i;                                  /* loop variable */

SetPenMode(0);                          /* initialize the graphics pen */
SetPenSize(8,3);
for (i = 0; i < DISKS; ++i) {           /* set up the game */
   pegs[0][i] = DISKS-i+1;
   pegs[1][i] = 0;
   pegs[2][i] = 0;
   DrawDisk(pegs[0][i], 0, i, 0);
   }
Tower(0, 2, 1, DISKS);                  /* move the disks */
}
```

Solution to problem 14.3.

```
/* Use recursion to reverse the elements in a linked list */

#include <stdio.h>
#include <stdlib.h>

typedef struct listStruct {             /* element in the list */
   struct listStruct *next;
   int value;
   }
   listStruct,
   *listPtr;
```

```
void Print (listPtr ptr)

/* Print the values in the list in reverse order                */

{
if (ptr != NULL) {
   Print(ptr->next);
   printf("%d\n", ptr->value);
   }
}


void main (void)

/* main program                                                 */

{
int i;                                  /* loop variable */
listPtr list;                           /* the list */
listPtr ptr;                            /* work pointer */

for (i = 1; i <= 10; ++i) {             /* create the list */
   ptr = (listPtr) malloc(sizeof(listStruct));
   ptr->next = list;
   ptr->value = i;
   list = ptr;
   }
Print(list);                            /* print the list */
}
```

Solution to problem 14.4.

```
/* A simple, recursive descent expression evaluator.  This       */
/* program handles +, -, * and /, as well as parenthesis.  All   */
/* operations are integer operations.                            */

#include <stdio.h>
#include <string.h>
#include <ctype.h>

typedef enum tokenType                  /* tokens in an expression */
   {add,subtract,multiply,divide,integer,lparen,rparen,eol}
   tokenType;

char ch;                                /* last char read by GetCh */
int index;                              /* index into str */
char str[81];                          /* string read from the keyboard */
tokenType token;                        /* last token read */
int tokenValue;                         /* value of last integer token */
```

```
void GetCh (void)

/* Read the next character from str                                  */
/*                                                                   */
/* Variables:                                                        */
/*    ch - char read; 0 if at the end of the string                  */
/*    index - index into str                                         */
/*    str - string to read the character from                        */

{
if (index >= strlen(str))
   ch = 0;
else {
   ch = str[index];
   ++index;
   }
}


void GetToken (void)

/* Read a token from the input string                                */

{
while (ch == ' ')                  /* skip to the first real character */
   GetCh();
if (ch == 0) {                         /* handle an end of line */
   token = eol;
   return;
   }
if (ch == '+') {                       /* handle add */
   token = add;
   GetCh();
   return;
   }
if (ch == '-') {                       /* handle subtract */
   token = subtract;
   GetCh();
   return;
   }
if (ch == '*') {                       /* handle multiply */
   token = multiply;
   GetCh();
   return;
   }
```

```
if (ch == '/') {                        /* handle divide */
   token = divide;
   GetCh();
   return;
   }
if (ch == '(') {                        /* handle ( */
   token = lparen;
   GetCh();
   return;
   }
if (ch == ')') {                        /* handle ) */
   token = rparen;
   GetCh();
   return;
   }
if (isdigit(ch)) {                      /* handle a number */
   token = integer;
   tokenValue = 0;
   while (isdigit(ch)) {
      tokenValue = tokenValue*10 + ch-'0';
      GetCh();
      }
   return;
   }
                                        /* handle bad input */
printf("\"%c\" is an illegal character.\n", ch);
token = eol;
}


int Expression (void);                  /* forward dec. of expression */

int Term (void)

/* Handle a number or parenthesis                                  */

{
int val;                                /* integer value */

if (token == integer) {                 /* handle an integer */
   val = tokenValue;
   GetToken();
   }
```

```
else if (token == lparen) {
   GetToken();                          /* skip the ( */
   val = Expression();                  /* evaluate the expression */
   if (token == rparen)                 /* skip the ) */
      GetToken();
   else
      printf(") expected\n");
   }
return val;                             /* return the value */
}


int Factor (void)

/* Do multiplies and divides                                    */

{
int value, newValue;                    /* values from Term */
tokenType operation;                    /* type of the operation */

value = Term();                         /* get the first value */
while ((token == multiply) || (token == divide)) {
   operation = token;                   /* skip the operation */
   GetToken();
   newValue = Term();                   /* get the second value */
   if (operation == multiply)           /* do the operation */
      value *= newValue;
   else
      value /= newValue;
   }
return value;                           /* return the result */
}
```

```
int Expression (void)

/* Evaluate an expression                                          */

{
int value, newValue;                    /* values from Factor */
tokenType operation;                    /* type of the operation */

value = Factor();                       /* get the first value */
while ((token == add) || (token == subtract)) {
   operation = token;                   /* skip the operation */
   GetToken();
   newValue = Factor();                 /* get the second value */
   if (operation == add)                /* do the operation */
      value += newValue;
   else
      value -= newValue;
   }
return value;                           /* return the result */
}


void main (void)

/* main program                                                    */

{
do {
   printf("Expression: ");
   scanf("%80[^\n]%*1[\n]", str);
   if (strlen(str)) {
      ch = ' ';
      index = 0;
      GetToken();
      printf("The value is %d\n\n", Expression());
      }
   }
while (strlen(str));
}
```

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.

```
do {
   swap = 0;
   for (i = 0; i < SIZE-1; ++i)
      if (nums[i] > nums[i+1]) {
         temp = nums[i];
         nums[i] = nums[i+1];
         nums[i+1] = temp;
         swap = 1;
         }
   }
while (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 often the most time-consuming operation. 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)/log(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 following program 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.                                          */

#include <stdio.h>

#define SIZE 10                          /* size of the array to sort */

int a[SIZE];                             /* array to sort */


void Fill (void)

/* Fill an array                                                    */
/*                                                                  */
/* Variables:                                                       */
/*    a - array to fill                                             */

{
int i;                                   /* loop variable */

for (i = 0; i < SIZE; ++i)
   a[i] = SIZE-i;
}


void Sort (int left, int right)

/* 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                                             */

{
int i,j;                                 /* array indices */
int pivot;                               /* pivot value */
int temp;                                /* used to swap values */

if (right > left) {                   /* quit if there is only 1 element */
   i = (left-1) + ((right-left+1) / 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) {                      /* partition the array */
         while ((a[i] <= pivot) && (i != j))
            ++i;
         while ((a[j] >= pivot) && (i != j))
            --j;
         temp = a[i];
         a[i] = a[j];
         a[j] = temp;
         }
      if (a[i] < pivot)                    /* find the pivot insert point */
         ++i;
      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 */
      }
   }


   void Print (void)

   /* Print the array                                                  */
   /*                                                                  */
   /* Variables:                                                       */
   /*    a - array to print                                            */

   {
   int i;

   for (i = 0; i < SIZE; ++i)
      printf("%d\n", a[i]);
   }


   void main (void)

   /* main program                                                     */

   {
   Fill();
   Sort(0, SIZE-1);
   Print();
   }
```

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 function. Just before the call to sort, bring up the variables window and type in a[0], a[1], 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 function, we will not worry too much about how each statement works. Instead, let's look closely at what happens on the whole. The Sort function 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 function 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) / 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 function. 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 function get called in the example shown in this section? (Hint: put a counter in the Sort function 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 functions 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 function 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 structures, the swap may take longer than the compare!).

As a programmer, you need a practical way to compare algorithms in a real setting. ORCA/C 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/C 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 function and the quick sort function into the same program, and write a main program that will call each of these functions 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 10 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 function 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 function 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 function call, your program uses a small amount of memory from the stack. The stack is limited in size. By default, programs written in ORCA/C have an 8192 byte stack. You can use the stacksize pragma 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/C 1.1, every function call uses 19 byes from the stack frame. If you call a function several times from a loop, the function uses the same 19 bytes each time you call it, but if a function 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 function, there are 2 parameters and 4 local variables. They use an additional 12 bytes of stack space, so that each call uses 31 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 function we have used in this lesson, it is easy to see that the Sort function cannot safely recur more than 264 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 function. 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 function. 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.                                                */

#include <stdio.h>

#define SIZE 2                          /* size of the array to sort */

int count = 0;                          /* number of compares */
int a[SIZE];                            /* array to sort */


void Sort (void)

/* Sort an array                                                */
/*                                                              */
/* Variables:                                                   */
/*    a - array to sort                                         */

{
int i;                                  /* loop variable/array index */
int swap;                               /* was a value swapped? */
int temp;                               /* temp; used for swapping */
```

```
do {
   swap = 0;
   for (i = 0; i < SIZE-1; ++i) {
      ++count;
      if (a[i] > a[i+1]) {
         temp = a[i];
         a[i] = a[i+1];
         a[i+1] = temp;
         swap = 1;
         }
      }
   }
while (swap);
}


void Fill (void)

/* Fill an array                                                    */
/*                                                                  */
/* Variables:                                                       */
/*    a - array to fill                                             */

{
int i;                                  /* loop variable */

for (i = 0; i < SIZE; ++i)
   a[i] = SIZE-i;
}


void main (void)

/* main program                                                     */

{
Fill();
Sort();
printf("There were %d compares.\n", count);
printf("n*(n-1) is %d\n", SIZE*(SIZE-1));
}
```

Solution to problem 15.2.

By changing the value of the constant size in the following program, you can find 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	3.46
4	7.83
5	12.08
10	65.16

```
/* Count the compares needed to sort a pseudo-random array with  */
/* a shell sort.                                                 */

#include <stdio.h>
#include <stdlib.h>

#define SIZE 2                          /* size of the array to sort */
#define TRIALS 100                      /* number of trial runs */

int count = 0;                          /* number of compares */
int a[SIZE];                            /* array to sort */


void Sort (void)

/* Sort an array                                                 */
/*                                                               */
/* Variables:                                                    */
/*    a - array to sort                                          */

{
int i;                                  /* loop variable/array index */
int swap;                               /* was a value swapped? */
int temp;                               /* temp; used for swapping */

do {
   swap = 0;
   for (i = 0; i < SIZE-1; ++i) {
      ++count;
      if (a[i] > a[i+1]) {
         temp = a[i];
         a[i] = a[i+1];
         a[i+1] = temp;
         swap = 1;
         }
      }
   }
while (swap);
}
```

```
void Fill (void)

/* Fill an array                                                    */
/*                                                                  */
/* Variables:                                                       */
/*    a - array to fill                                             */

{
int i;                                  /* loop variable */

for (i = 0; i < SIZE; ++i)
   a[i] = rand() % SIZE;
}


void main (void)

/* main program                                                     */

{
int i;                                  /* loop variable */

srand(2345);                   /* initialize the random number generator */
for (i = 0; i < TRIALS; ++i) {          /* do the trial runs */
   Fill();
   Sort();
   }
                                        /* print the results */
printf("The average number of compares is %.2f.\n",
   ((float)count)/TRIALS);
}
```

Solution to problem 15.3.

The quick sort subroutine is called 13 times.

```
/* Determine how many times Sort is called                          */

#include <stdio.h>

#define SIZE 10                         /* size of the array to sort */

int a[SIZE];                            /* array to sort */
int count = 0;                          /* number of calls */
```

```
void Fill (void)

/* Fill an array                                                    */
/*                                                                  */
/* Variables:                                                       */
/*    a - array to fill                                             */

{
int i;                                  /* loop variable */

for (i = 0; i < SIZE; ++i)
   a[i] = SIZE-i;
}


void Sort (int left, int right)

/* 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                                             */

{
int i,j;                                /* array indices */
int pivot;                              /* pivot value */
int temp;                               /* used to swap values */

++count;                                /* update the counter */
if (right > left) {                   /* quit if there is only 1 element */
   i = (left-1) + ((right-left+1) / 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) {                     /* partition the array */
      while ((a[i] <= pivot) && (i != j))
         ++i;
      while ((a[j] >= pivot) && (i != j))
         --j;
      temp = a[i];
      a[i] = a[j];
      a[j] = temp;
      }
   if (a[i] < pivot)                    /* find the pivot insert point */
      ++i;
```

```
   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 */
   }
}


void Print (void)

/* Print the array                                                  */
/*                                                                  */
/* Variables:                                                       */
/*    a - array to print                                            */

{
int i;

for (i = 0; i < SIZE; ++i)
   printf("%d\n", a[i]);
}


void main (void)

/* main program                                                     */

{
Fill();
Sort(0, SIZE-1);
Print();
printf("Sort is called %d times.\n", count);
}
```

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	3.46	4.72
4	7.83	8.56
5	12.08	12.77
10	65.16	39.61

```
/* Check to see how many compares are needed by a typical quicksort */

#include <stdio.h>
#include <stdlib.h>

#define SIZE 10                         /* size of the array to sort */
#define TRIALS 100                      /* number of trial runs */

int a[SIZE];                            /* array to sort */
int count = 0;                      /* number of array element compares */


void Fill (void)

/* Fill an array                                                    */
/*                                                                  */
/* Variables:                                                       */
/*    a - array to fill                                             */

{
int i;                                  /* loop variable */

for (i = 0; i < SIZE; ++i)
   a[i] = rand() % SIZE;
}


void Sort (int left, int right)

/* 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                                             */

{
int i,j;                                /* array indices */
int pivot;                              /* pivot value */
int temp;                               /* used to swap values */

if (right > left) {                  /* quit if there is only 1 element */
   i = (left-1) + ((right-left+1) / 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) {                    /* partition the array */
      ++count;
      while ((a[i] <= pivot) && (i != j)) {
         ++count;
         ++i;
         }
      ++count;
      while ((a[j] >= pivot) && (i != j)) {
         ++count;
         --j;
         }
      temp = a[i];
      a[i] = a[j];
      a[j] = temp;
      }
   ++count;
   if (a[i] < pivot)                   /* find the pivot insert point */
      ++i;
   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 */
   }
}


void main (void)

/* main program                                                  */

{
int i;                                 /* loop variable */

srand(2345);                  /* initialize the random number generator */
for (i = 0; i < TRIALS; ++i) {         /* do the trial runs */
   Fill();
   Sort(0, SIZE-1);
   }
                                       /* print the results */
printf("The average number of compares is %.2f.\n",
   ((float)count)/TRIALS);
}
```

Solution to problem 15.5.

The shell sort and quick sort require almost exactly the same amount of time when the array has 19 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 float, instead of an array of integer, the break-even point changes from 19 elements to 13 elements.

```
/* Compare the time for a quick sort to the time for a shell    */
/* sort.                                                        */

#include <stdio.h>
#include <stdlib.h>

#define SIZE 10                         /* size of the array to sort */
#define TRIALS 10                       /* number of trial runs */

int a[SIZE], b[SIZE];                   /* array(s) to sort */


void Fill (void)

/* Fill two arrays                                              */
/*                                                              */
/* Variables:                                                   */
/*    a,b - arrays to fill                                      */

{
int i;                                  /* loop variable */

for (i = 0; i < SIZE; ++i) {
   a[i] = rand() % SIZE;
   b[i] = a[i];
   }
}


void ShellSort (void)

/* Sort an array                                                */
/*                                                              */
/* Variables:                                                   */
/*    b - array to sort                                         */

{
int i;                                  /* loop variable/array index */
int swap;                               /* was a value swapped? */
int temp;                               /* temp; used for swapping */
```

```
do {
   swap = 0;
   for (i = 0; i < SIZE-1; ++i) {
      if (b[i] > b[i+1]) {
         temp = b[i];
         b[i] = b[i+1];
         b[i+1] = temp;
         swap = 1;
         }
      }
   }
while (swap);
}


void QuickSort (int left, int right)

/* 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                                             */

{
int i,j;                                /* array indices */
int pivot;                              /* pivot value */
int temp;                               /* used to swap values */

if (right > left) {                   /* quit if there is only 1 element */
   i = (left-1) + ((right-left+1) / 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) {                     /* partition the array */
      while ((a[i] <= pivot) && (i != j))
         ++i;
      while ((a[j] >= pivot) && (i != j))
         --j;
      temp = a[i];
      a[i] = a[j];
      a[j] = temp;
      }
   if (a[i] < pivot)                    /* find the pivot insert point */
      ++i;
```

```
   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 */
   }
}


void main (void)

/* main program                                                  */

{
int i;                                   /* loop variable */

srand(2345);                   /* initialize the random number generator */
for (i = 0; i < TRIALS; ++i) {           /* do the trial runs */
   Fill();
   QuickSort(0, SIZE-1);
   ShellSort();
   }
}
```

Solution to problem 15.6.

```
/* A better sort than QuickSort                                  */

#include <stdio.h>
#include <stdlib.h>

#define SIZE 100                         /* size of the array to sort */

int a[SIZE];                             /* array to sort */


void Fill (void)

/* Fill an array                                                 */
/*                                                               */
/* Variables:                                                    */
/*    a - array to fill                                          */

{
int i;                                   /* loop variable */

for (i = 0; i < SIZE; ++i)
   a[i] = rand() % SIZE;
}
```

```
void Sort (int left, int right, int depth)

/* 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                                             */

#define SHELLSIZE 19               /* when a quick sort becomes faster */
#define MAXDEPTH 100                   /* max recursion depth */
{
int i,j;                               /* array indices */
int pivot;                             /* pivot value */
int swap;                              /* was a value swapped? */
int temp;                              /* used to swap values */

if (((right - left) < SHELLSIZE) || (depth > MAXDEPTH)) {
   do {                                /* do a shell sort */
      swap = 0;
      for (i = left; i < right; ++i) {
         if (a[i] > a[i+1]) {
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
            swap = 1;
            }
         }
      }
   while (swap);
   }
else {                                 /* do a quick sort */
   if (right > left) {              /* quit if there is only 1 element */
      i = (left-1) + ((right-left+1) / 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) {                  /* partition the array */
         while ((a[i] <= pivot) && (i != j))
            ++i;
         while ((a[j] >= pivot) && (i != j))
            --j;
         temp = a[i];
         a[i] = a[j];
         a[j] = temp;
         }
      if (a[i] < pivot)                 /* find the pivot insert point */
         ++i;
      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 */
      }
   }
}


void Print (void)

/* Print the array                                                  */
/*                                                                  */
/* Variables:                                                       */
/*    a - array to print                                            */

{
int i;

for (i = 0; i < SIZE; ++i) {
   printf("%4d", a[i]);
   if ((i % 8) == 7)
      printf("\n");
   }
}


void main (void)

/* main program                                                     */

{
srand(2345);                        /* initialize rand() */
Fill();                             /* fill the array */
Sort(0, SIZE-1, 1);                 /* sort the array */
Print();                            /* print the array */
}
```

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 should the last topic of an introductory programming course be this one, 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 structures, a sequential search would look like this:

```
i = 1;
found = 0;
do {
   if (a[i].age == 40)
      found = 1;
   else
      ++i;
   }
while ((!found)
   && (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 -1 if it is not. Use an array of 100 integers, with each array element containing an even number. For example, a[0] would be 2, a[1] 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 C

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.

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 C cross reference program to look at binary trees. The purpose of this lesson isn't really to make you write a C cross reference program, so this section gives you one to start with. This C 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. The C language has a way to do this – when you run a C program from the shell window, the function main actually gets passed two parameters in a very special way, so that if you ignore the parameters, nothing bad happens. The two parameters are argc, an integer that tells you how many parameters were passed; and argv, an array of pointers to strings. The things you type in the shell window, including the name of the program itself, are broken up into tokens, splitting the command line wherever a space appears. The number of tokens are passed in argc, while the array argv contains pointers to the strings, with one extra pointer set to NULL to mark the end of the list.

Here's a simple program showing how it's done. The program even asks for a file name if the user forgets to give one.

```
/* read a file name */

#include <stdio.h>

char fName[65];
/* file name */
char *fNamePtr;
/* pointer to the file name */

int main (int argc, char *argv[])

{
if (argc < 2) {
   printf("File to cross reference: ");
   fNamePtr = fName;
   scanf("%64s", fName);
   if (strlen(fName) == 0)
      return -1;
   }
else {
   if (argc > 2)
      printf("Extra input ignored.\n");
   fNamePtr = argv[1];
   }

printf("File name = '%s'\n", fNamePtr);
}
```

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.CC; 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.

It doesn't take much to turn this into a program that reads a file and echoes it to the shell window. That's something you have done before, but never in a way that your program looked like a command in the shell window, where you just type the name of the program and the file to echo. Listing 16.1 shows the program after its next step of development, reading and echoing the file.

Listing 16.1

```
/* Read a file and echo it to the console */

#include <stdio.h>

char fName[65];                         /* file name */
FILE *f;                                /* file variable */
char *fNamePtr;                         /* pointer to the file name */

int main (int argc, char *argv[])

{
char ch;                                /* character from the file */

if (argc < 2) {                         /* get a file name */
   printf("File to cross reference: ");
   fNamePtr = fName;
   scanf("%64s", fName);
   if (strlen(fName) == 0)
```

```
      return -1;
   }
else {
   if (argc > 2)
      printf("Extra input ignored.\n");
   fNamePtr = argv[1];
   }

f = fopen(fNamePtr, "r");              /* open the file */
if (f == NULL) {
   printf("Could not open %s.\n", fNamePtr);
   return -1;
   }
do {                                   /* echo the file */
   ch = fgetc(f);
   if (ch != EOF)
      putchar(ch);
   }
while (ch != EOF);
fclose(f);                             /* close the file */
return 0;                              /* return with no error */
}
```

The last step is to tie all of this together into a C 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 carefully.

If you have time, you might want to try writing this program on your own before typing in the version you see in listing 16.2.

Listing 16.2

```
/* XREF                                                          */
/*                                                               */
/* This program generates a cross reference of a C 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.                                                    */

#include <stdio.h>
#include <ctype.h>
```

```
#include <string.h>
#include <stdlib.h>

#define symbolLength 80                 /* max length of a symbol */

char fName[65];                         /* file name */
FILE *f;                                /* file variable */
char *fNamePtr;                         /* pointer to the file name */

typedef struct lineStruct {             /* line number list */
   struct lineStruct *next;
   int number;
   }
   lineStruct, *linePtr;

typedef struct symbolStruct {           /* symbol table entry */
   struct symbolStruct *next;
   char symbol[symbolLength+1];
   linePtr lines;
   }
   symbolStruct, *symbolPtr;


symbolPtr symbols = NULL;               /* symbol table */
int lineNumber = 1;                     /* current line number */
char ch = ' ';                          /* current character */
char token[symbolLength+1];             /* current token */
int tokenLine;                         /* line number at start of token */


void GetCh (void)

/* Read a character from the file                                  */
/*                                                                 */
/* Variables:                                                      */
/*    ch - character read                                          */
/*    lineNumber - current line number                             */

{
ch = fgetc(f);
if (ch == '\n')
   ++lineNumber;
}


void SkipComment (void)

/* Skip comments in the program                                    */

{
```

```
   do {
      GetCh();
      if (ch == '*') {
         GetCh();
         if (ch == '/')
            return;
         }
      }
   while (ch != EOF);
   }


   void NextCh (void)

   /* Get the next character from the file, skipping comments      */

   {
   GetCh();                                  /* get the next character */
   if (ch == '/') {                          /* skip comments */
      GetCh();
      if (ch == '*') {
         GetCh();
         SkipComment();
         }
      }
   }


   void GetToken (void)

   /* Read a word from the source file                              */
   /*                                                               */
   /* Variables:                                                    */
   /*    lineNumber - current line number                           */
   /*    token - string read                                        */
   /*    tokenLine - line number at the start of the token          */

   {
   int len = 0;                              /* length of the token */

   if (ch != EOF) {
                                             /* skip to the next token */
      while ((!iscsymf(ch)) && (ch != EOF))
         GetCh();
      tokenLine = lineNumber;                /* record the line number */
                                             /* record the token */
      while (iscsym(ch) && (ch != EOF)) {
         if (len < symbolLength) {
            token[len] = ch;
            ++len;
```

```
         }
      GetCh();
      }
   }
token[len] = (char) 0;                  /* mark the end of the string */
}


void Insert (void)

/* 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  */

{
linePtr lPtr;                           /* current line number pointer */
symbolPtr sPtr;                         /* current symbol pointer */

sPtr = symbols;                         /* try to find the symbol */
while (sPtr != NULL) {
   if (strcmp(token, sPtr->symbol) == 0)
      goto make;
   sPtr = sPtr->next;
   }
                                     /* none exists: create a new entry */
sPtr = (symbolPtr) malloc(sizeof(symbolStruct));
sPtr->next = symbols;
symbols = sPtr;
strcpy(sPtr->symbol, token);
sPtr->lines = NULL;
make:
                                        /* enter the line number */
lPtr = (linePtr) malloc(sizeof(lineStruct));
lPtr->next = sPtr->lines;
sPtr->lines = lPtr;
lPtr->number = tokenLine;
}


void PrintNumber (linePtr nPtr)

/* Recursively print the line numbers in reverse order          */
/*                                                              */
/* Parameters:                                                  */
/*    nPtr - pointer to the remainder of the line number list   */
```

```
{
if (nPtr != NULL) {
   PrintNumber(nPtr->next);
   printf("%d ", nPtr->number);
   }
}


void PrintSymbols (void)

/* Print the symbols found and line numbers                    */
/*                                                             */
/* Variables:                                                  */
/*    symbols - pointer to the first entry in the symbol table  */

{
symbolPtr sPtr;                         /* current symbol pointer */

sPtr = symbols;
while (sPtr != NULL) {
   printf("%16s  ", sPtr->symbol);
   PrintNumber(sPtr->lines);
   printf("\n");
   sPtr = sPtr->next;
   }
}


int main (int argc, char *argv[])

/* Main program                                                */

{
if (argc < 2) {                         /* get a file name */
   printf("File to cross reference: ");
   fNamePtr = fName;
   scanf("%64s", fName);
   if (strlen(fName) == 0)
      return -1;
   }
else {
   if (argc > 2)
      printf("Extra input ignored.\n");
   fNamePtr = argv[1];
   }

f = fopen(fNamePtr, "r");               /* open the file */
if (f == NULL) {
   printf("Could not open %s.\n", fNamePtr);
   return -1;
```

```
   }

do {                                /* collect the symbols in the file */
   GetToken();
   if (strlen(token))
      Insert();
   }
while (strlen(token));
PrintSymbols();                        /* print the symbol table */

fclose(f);                             /* close the file */
return 0;                              /* return with no error */
}
```

There are a couple of problems with the C 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 void, for example. That one you will solve yourself a bit later, as one of the problems.

The Binary Tree

The major problem with a sequential search of a linked list is the same as the major problem with a sequential search of an array: 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 structure in each structure, rather than one. In a linked list, each record has a pointer we have called next that points to the next structure in the list. In a binary tree, each structure has two pointers, which we will call left and right. If we look at a particular structure, 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 in listing 16.3 reads strings from the keyboard and adds them to a binary tree.

Listing 16.3

```
/* Create a binary tree from keyboard strings */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct treeStruct {            /* tree entry */
   struct treeStruct *left, *right;
   char str[20];
```

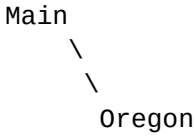
```
   }
   treeStruct, *treePtr;

treePtr tree = NULL;                    /* top of the tree */


void Add (treePtr *ptr, treePtr rec)

/* Add a record to the tree                                    */
/*                                                             */
/* Parameters:                                                 */
/*    ptr - next node in the tree                              */
/*    rec - record to add to the tree                          */

{
int cmp;                                /* result of strcmp */

if (*ptr == NULL)
   *ptr = rec;
else {
   cmp = strcmp(rec->str, (*ptr)->str);
   if (cmp < 0)
      Add(&((*ptr)->left), rec);
   else if (cmp > 0)
      Add(&((*ptr)->right), rec);
   }
}


void main (void)

/* Main program                                                */

{
char str[20];                           /* work string */
treePtr tPtr;                           /* work pointer */

do {
   printf("string: ");                  /* get a string */
   scanf("%19[^\n]%*1[\n]", str);
   if (strlen(str)) {
                                        /* create a new record */
      tPtr = (treePtr) malloc(sizeof(treeStruct));
      tPtr->left = NULL;
      tPtr->right = NULL;
      strcpy(tPtr->str, str);
      Add(&tree, tPtr);                 /* add it to the tree */
      }
   }
while (strlen(str));                   /* loop until no string is given */
```

```
}
```

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 NULL, record the string, and call Add. The procedure Add sees that *ptr is NULL, 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:

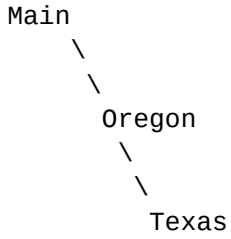
Well, there isn't much there, yet, so our meager tree doesn't look very impressive. Adding Oregon shapes things up a bit, though. This time, when we call Add, the function sees that ptr is not NULL, and checks to see if Oregon is less than Main. It isn't, so it moves on to the next check to be sure that 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 NULL, 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:

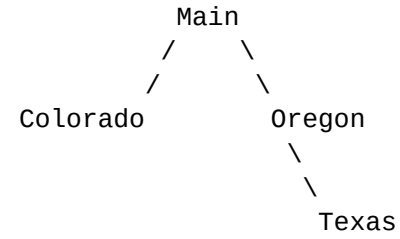


Notice how recursion handled the problem of tracing the tree fairly neatly. Once we decided that the top node existed, and which way to go, we called Add again, treating ptr->right as a brand-new tree, which in a sense it is. If you recall, when recursion was first introduced, I said that the way to 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.

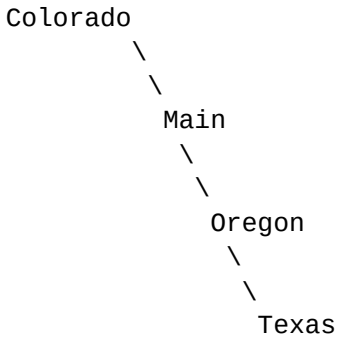
The next state to add is Texas, which makes two recursive calls, getting tacked onto the tree as the right child of Oregon. Follow through the code, writing the steps down on paper if necessary, to see how this is done.



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:



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 structure 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 structures 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 structures 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 structure to the tree searches the tree as a side effect! Here's a function, based on the Add function, that will search the tree, returning a pointer to the correct structure, or NULL if the structure does not exist:

```
treePtr  Search  (treePtr  ptr,  char
str[20])

/* Search the tree for a structure     */
/*                                     */
/* Parameters:                         */
/*    ptr - next node in the tree      */
/*    str - value to search for        */
/*                                     */
/* Returns:                            */
/*    Pointer to the matching          */
/*       structure; NULL if the        */
/*       structure does not exist.     */

{
int cmp;   /* result of string compare */

if (ptr == NULL)
   return NULL;
cmp = strcmp(rec->str, str);
if (cmp < 0)
   return Search(ptr->left, rec);
else if (cmp > 0)
   return Search(ptr->right, rec);
else
   return ptr;
}
```

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.

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 subroutine that will trace through the tree, doing something in order. In our case, we want to print the symbols found in the C program. Here's a simple print subroutine that prints the states in our example program; the subroutine in the C cross reference program will have exactly the same structure.

```
void Print (treePtr ptr)

/* Print a tree                        */
/*                                     */
/* Parameters:                         */
/*    ptr - next node in the tree      */

{
if (ptr != NULL) {
   Print(ptr->left);
   printf("%s\n", ptr->str);
   Print(ptr->right);
   }
}
```

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 structure 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 NULL keeps us from stepping off of the "end" of the tree.

Problem 16.2. Add the print subroutine to the binary tree 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 C. You can find a list of the reserved words in your C 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.

Lesson Sixteen
Solutions to Problems

Solution to problem 16.1.

```
/* Implement a binary search                                        */

#include <stdio.h>

#define SIZE 100                        /* size of the array */

int a[SIZE];                            /* array to search */


int Find (int val)

/* 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, -1 is returned.        */
/*                                                                  */
/* Variables:                                                       */
/*    a - array to search                                           */

{
int left,right,middle;                  /* array indices */

left = 0;
right = SIZE-1;
do {
   middle = (left+right) / 2;
   if (val < a[middle])
      right = middle-1;
   else
      left = middle+1;
   }
while ((val != a[middle]) && (left <= right));
if (val == a[middle])
   return middle;
return -1;
}
```

```
void Fill (void)

/* Fill the array                                                    */
/*                                                                   */
/* Variables:                                                        */
/*    a - array to search                                            */

{
int i;                                  /* loop index */

for (i = 0; i < SIZE; ++i)
   a[i] = (i+1)*2;
}


void Test (void)

/* Test the Find procedure                                           */

{
int i;                                  /* loop index */

for (i = 0; i < SIZE; ++i)  /* check to be sure each value can be found */
   if (a[Find((i+1)*2)] != (i+1)*2)
      printf("Failed to find %d\n", i*2);
if (Find(0) != -1)     /* check to be sure missing values are not found */
   printf("Reported \"found\" for Find(0)\n");
if (Find(101) != -1)
   printf("Reported \"found\" for Find(101)\n");
if (Find(202) != -1)
   printf("Reported \"found\" for Find(202)\n");
}


void main (void)

/* main program                                                      */

{
Fill();                                 /* fill the array */
Test();                                 /* test the Find procedure */
}
```

Solution to problem 16.2.

```
/* Create a binary tree from keyboard strings */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
typedef struct treeStruct {             /* tree entry */
   struct treeStruct *left, *right;
   char str[20];
   }
   treeStruct, *treePtr;

treePtr tree = NULL;                    /* top of the tree */


void Add (treePtr *ptr, treePtr rec)

/* Add a record to the tree                                          */
/*                                                                   */
/* Parameters:                                                       */
/*    ptr - next node in the tree                                    */
/*    rec - record to add to the tree                                */

{
int cmp;                                /* result of strcmp */

if (*ptr == NULL)
   *ptr = rec;
else {
   cmp = strcmp(rec->str, (*ptr)->str);
   if (cmp < 0)
      Add(&((*ptr)->left), rec);
   else if (cmp > 0)
      Add(&((*ptr)->right), rec);
   }
}


void Print (treePtr ptr)

/* Print a tree                                                      */
/*                                                                   */
/* Parameters:                                                       */
/*    ptr - next node in the tree                                    */

{
if (ptr != NULL) {
   Print(ptr->left);
   printf("%s\n", ptr->str);
   Print(ptr->right);
   }
}
```

```
void main (void)

/* Main program                                                    */

{
char str[20];                          /* work string */
treePtr tPtr;                          /* work pointer */

do {
   printf("string: ");                 /* get a string */
   scanf("%19[^\n]%*1[\n]", str);
   if (strlen(str)) {
                                       /* create a new record */
      tPtr = (treePtr) malloc(sizeof(treeStruct));
      tPtr->left = NULL;
      tPtr->right = NULL;
      strcpy(tPtr->str, str);
      Add(&tree, tPtr);                /* add it to the tree */
      }
   }
while (strlen(str));                  /* loop until no string is given */
Print(tree);                           /* print the tree in order */
}
```

Solution to problem 16.3.

```
/* XREF                                                            */
/*                                                                 */
/* This program generates a cross reference of a C 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.                                                      */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#define symbolLength 80                /* max length of a symbol */

char fName[65];                        /* file name */
FILE *f;                               /* file variable */
char *fNamePtr;                        /* pointer to the file name */
```

```
typedef struct lineStruct {             /* line number list */
   struct lineStruct *next;
   int number;
   }
   lineStruct, *linePtr;

typedef struct symbolStruct {           /* symbol table entry */
   struct symbolStruct *left, *right;
   char symbol[symbolLength+1];
   linePtr lines;
   }
   symbolStruct, *symbolPtr;


symbolPtr symbols = NULL;               /* symbol table */
int lineNumber = 1;                     /* current line number */
char ch = ' ';                          /* current character */
char token[symbolLength+1];             /* current token */
int tokenLine;                         /* line number at start of token */


void GetCh (void)

/* Read a character from the file                                  */
/*                                                                 */
/* Variables:                                                      */
/*    ch - character read                                          */
/*    lineNumber - current line number                             */

{
ch = fgetc(f);
if (ch == '\n')
   ++lineNumber;
}
```

```
void SkipComment (void)

/* Skip comments in the program                                    */

{
do {
   GetCh();
   if (ch == '*') {
      GetCh();
      if (ch == '/')
         return;
      }
   }
while (ch != EOF);
}


void NextCh (void)

/* Get the next character from the file, skipping comments      */

{
GetCh();                                  /* get the next character */
if (ch == '/') {                          /* skip comments */
   GetCh();
   if (ch == '*') {
      GetCh();
      SkipComment();
      }
   }
}


void GetToken (void)

/* Read a word from the source file                                */
/*                                                                 */
/* Variables:                                                      */
/*    lineNumber - current line number                             */
/*    token - string read                                          */
/*    tokenLine - line number at the start of the token            */
```

```
{
int len = 0;                            /* length of the token */

if (ch != EOF) {
                                        /* skip to the next token */
   while ((!iscsymf(ch)) && (ch != EOF))
      GetCh();
   tokenLine = lineNumber;              /* record the line number */
                                        /* record the token */
   while (iscsym(ch) && (ch != EOF)) {
      if (len < symbolLength) {
         token[len] = ch;
         ++len;
         }
      GetCh();
      }
   }
token[len] = (char) 0;                  /* mark the end of the string */
}


void AddUse (symbolPtr ptr)

/* Add a line number to the symbol table entry                  */
/*                                                              */
/* Parameters:                                                  */
/*    ptr - symbol table entry to update                        */
/*                                                              */
/* Variables:                                                   */
/*    tokenLine - line number at the start of the token         */

{
linePtr lPtr;                           /* current line number pointer */

lPtr = (linePtr) malloc(sizeof(lineStruct));
lPtr->next = ptr->lines;
ptr->lines = lPtr;
lPtr->number = tokenLine;
}
```

```
void Insert (symbolPtr *ptr)

/* Insert a symbol use in the symbol table.  If the symbol does */
/* not exist, create a new entry.                               */
/*                                                              */
/* Parameters:                                                  */
/*    ptr - pointer to the top node of the tree                 */
/*                                                              */
/* Variables:                                                   */
/*    token - symbol to insert                                  */

{
symbolPtr sPtr;                         /* work pointer */
int cmp;                                /* result of strcmp */

if (*ptr == NULL) {                     /* no entry: create one */
   sPtr = (symbolPtr) malloc(sizeof(symbolStruct));
   sPtr->left = NULL;
   sPtr->right = NULL;
   strcpy(sPtr->symbol, token);
   sPtr->lines = NULL;
   *ptr = sPtr;                         /* add it to the tree */
   AddUse(sPtr);                        /* mark the line number */
   }
else {
   cmp = strcmp(token, (*ptr)->symbol);
   if (cmp < 0)
      Insert(&((*ptr)->left));          /* follow the left link */
   else if (cmp > 0)
      Insert(&((*ptr)->right));         /* follow the right link */
   else
      AddUse(*ptr);                     /* found an existing entry */
   }
}


void PrintNumber (linePtr nPtr)

/* Recursively print the line numbers in reverse order          */
/*                                                              */
/* Parameters:                                                  */
/*    nPtr - pointer to the remainder of the line number list   */

{
if (nPtr != NULL) {
   PrintNumber(nPtr->next);
   printf("%d ", nPtr->number);
   }
}
```

```
void PrintSymbols (symbolPtr ptr)

/* Print the symbols found and line numbers                    */
/*                                                             */
/* Parameters:                                                 */
/*    ptr - pointer to the next node in the symbol table       */

{
if (ptr != NULL) {
   PrintSymbols(ptr->left);             /* print symbols to the left */
   printf("%16s  ", ptr->symbol);      /* print this symbol */
   PrintNumber(ptr->lines);
   printf("\n");
   PrintSymbols(ptr->right);            /* print symbols to the right */
   }
}


int main (int argc, char *argv[])

/* Main program                                                */

{
if (argc < 2) {                         /* get a file name */
   printf("File to cross reference: ");
   fNamePtr = fName;
   scanf("%64s", fName);
   if (strlen(fName) == 0)
      return -1;
   }
else {
   if (argc > 2)
      printf("Extra input ignored.\n");
   fNamePtr = argv[1];
   }

f = fopen(fNamePtr, "r");               /* open the file */
if (f == NULL) {
   printf("Could not open %s.\n", fNamePtr);
   return -1;
   }
```

```
do {                                /* collect the symbols in the file */
   GetToken();
   if (strlen(token))
      Insert(&symbols);
   }
while (strlen(token));
PrintSymbols(symbols);                 /* print the symbol table */

fclose(f);                             /* close the file */
return 0;                              /* return with no error */
}
```

Solution to problem 16.4.

```
/* XREF                                                          */
/*                                                               */
/* This program generates a cross reference of a C 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.                                                    */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#define symbolLength 80                /* max length of a symbol */

char fName[65];                        /* file name */
FILE *f;                               /* file variable */
char *fNamePtr;                        /* pointer to the file name */

typedef struct lineStruct {            /* line number list */
   struct lineStruct *next;
   int number;
   }
   lineStruct, *linePtr;

typedef struct symbolStruct {          /* symbol table entry */
   struct symbolStruct *left, *right;
   char symbol[symbolLength+1];
   int reserved;
   linePtr lines;
   }
   symbolStruct, *symbolPtr;
```

```
symbolPtr symbols = NULL;               /* symbol table */
int lineNumber = 1;                     /* current line number */
char ch = ' ';                          /* current character */
char token[symbolLength+1];             /* current token */
int tokenLine;                         /* line number at start of token */


void GetCh (void)

/* Read a character from the file                                    */
/*                                                                   */
/* Variables:                                                        */
/*    ch - character read                                            */
/*    lineNumber - current line number                               */

{
ch = fgetc(f);
if (ch == '\n')
   ++lineNumber;
}


void SkipComment (void)

/* Skip comments in the program                                      */

{
do {
   GetCh();
   if (ch == '*') {
      GetCh();
      if (ch == '/')
         return;
      }
   }
while (ch != EOF);
}
```

```
void NextCh (void)

/* Get the next character from the file, skipping comments      */

{
GetCh();                                /* get the next character */
if (ch == '/') {                        /* skip comments */
   GetCh();
   if (ch == '*') {
      GetCh();
      SkipComment();
      }
   }
}


void GetToken (void)

/* Read a word from the source file                             */
/*                                                              */
/* Variables:                                                   */
/*    lineNumber - current line number                          */
/*    token - string read                                       */
/*    tokenLine - line number at the start of the token         */

{
int len = 0;                            /* length of the token */

if (ch != EOF) {
                                        /* skip to the next token */
   while ((!iscsymf(ch)) && (ch != EOF))
      GetCh();
   tokenLine = lineNumber;              /* record the line number */
                                        /* record the token */
   while (iscsym(ch) && (ch != EOF)) {
      if (len < symbolLength) {
         token[len] = ch;
         ++len;
         }
      GetCh();
      }
   }
token[len] = (char) 0;                  /* mark the end of the string */
}
```

```
void AddUse (symbolPtr ptr)

/* Add a line number to the symbol table entry                  */
/*                                                              */
/* Parameters:                                                  */
/*    ptr - symbol table entry to update                        */
/*                                                              */
/* Variables:                                                   */
/*    tokenLine - line number at the start of the token         */

{
linePtr lPtr;                          /* current line number pointer */

lPtr = (linePtr) malloc(sizeof(lineStruct));
lPtr->next = ptr->lines;
ptr->lines = lPtr;
lPtr->number = tokenLine;
}


void Insert (symbolPtr *ptr, int reserved)

/* Insert a symbol use in the symbol table.  If the symbol does */
/* not exist, create a new entry.                               */
/*                                                              */
/* Parameters:                                                  */
/*    ptr - pointer to the top node of the tree                 */
/*    reserved - is this a reserved word?                       */
/*                                                              */
/* Variables:                                                   */
/*    token - symbol to insert                                  */

{
symbolPtr sPtr;                        /* work pointer */
int cmp;                               /* result of strcmp */

if (*ptr == NULL) {                    /* no entry: create one */
   sPtr = (symbolPtr) malloc(sizeof(symbolStruct));
   sPtr->left = NULL;
   sPtr->right = NULL;
   strcpy(sPtr->symbol, token);
   sPtr->reserved = reserved;
   sPtr->lines = NULL;
   *ptr = sPtr;                        /* add it to the tree */
   if (!reserved)                      /* mark the line number */
      AddUse(sPtr);
   }
```

```
else {
   cmp = strcmp(token, (*ptr)->symbol);
   if (cmp < 0)
      Insert(&((*ptr)->left), reserved); /* follow the left link */
   else if (cmp > 0)
      Insert(&((*ptr)->right), reserved); /* follow the right link */
   else if (!(*ptr)->reserved)
      AddUse(*ptr);                    /* found an existing entry */
   }
}


void PrintNumber (linePtr nPtr)

/* Recursively print the line numbers in reverse order          */
/*                                                              */
/* Parameters:                                                  */
/*    nPtr - pointer to the remainder of the line number list   */

{
if (nPtr != NULL) {
   PrintNumber(nPtr->next);
   printf("%d ", nPtr->number);
   }
}


void PrintSymbols (symbolPtr ptr)

/* Print the symbols found and line numbers                     */
/*                                                              */
/* Parameters:                                                  */
/*    ptr - pointer to the next node in the symbol table        */

{
if (ptr != NULL) {
   PrintSymbols(ptr->left);            /* print symbols to the left */
   if (!ptr->reserved) {
      printf("%16s  ", ptr->symbol);    /* print this symbol */
      PrintNumber(ptr->lines);
      printf("\n");
      }
   PrintSymbols(ptr->right);           /* print symbols to the right */
   }
}
```

```
void ReservedWords (void)

/* Add the C reserved words to the symbol table */

{
strcpy(token, "if");            Insert(&symbols, 1);
strcpy(token, "default");       Insert(&symbols, 1);
strcpy(token, "case");          Insert(&symbols, 1);
strcpy(token, "asm");           Insert(&symbols, 1);
strcpy(token, "auto");          Insert(&symbols, 1);
strcpy(token, "break");         Insert(&symbols, 1);
strcpy(token, "comp");          Insert(&symbols, 1);
strcpy(token, "char");          Insert(&symbols, 1);
strcpy(token, "continue");      Insert(&symbols, 1);
strcpy(token, "const");         Insert(&symbols, 1);
strcpy(token, "extended");      Insert(&symbols, 1);
strcpy(token, "double");        Insert(&symbols, 1);
strcpy(token, "do");            Insert(&symbols, 1);
strcpy(token, "enum");          Insert(&symbols, 1);
strcpy(token, "else");          Insert(&symbols, 1);
strcpy(token, "float");         Insert(&symbols, 1);
strcpy(token, "extern");        Insert(&symbols, 1);
strcpy(token, "goto");          Insert(&symbols, 1);
strcpy(token, "for");           Insert(&symbols, 1);
strcpy(token, "sizeof");        Insert(&symbols, 1);
strcpy(token, "register");      Insert(&symbols, 1);
strcpy(token, "int");           Insert(&symbols, 1);
strcpy(token, "inline");        Insert(&symbols, 1);
strcpy(token, "pascal");        Insert(&symbols, 1);
strcpy(token, "long");          Insert(&symbols, 1);
strcpy(token, "segment");       Insert(&symbols, 1);
strcpy(token, "return");        Insert(&symbols, 1);
strcpy(token, "signed");        Insert(&symbols, 1);
strcpy(token, "short");         Insert(&symbols, 1);
strcpy(token, "union");         Insert(&symbols, 1);
strcpy(token, "struct");        Insert(&symbols, 1);
strcpy(token, "static");        Insert(&symbols, 1);
strcpy(token, "typedef");       Insert(&symbols, 1);
strcpy(token, "switch");        Insert(&symbols, 1);
strcpy(token, "void");          Insert(&symbols, 1);
strcpy(token, "unsigned");      Insert(&symbols, 1);
strcpy(token, "while");         Insert(&symbols, 1);
strcpy(token, "volatile");      Insert(&symbols, 1);
}
```

```
int main (int argc, char *argv[])

/* Main program                                                    */

{
if (argc < 2) {                          /* get a file name */
   printf("File to cross reference: ");
   fNamePtr = fName;
   scanf("%64s", fName);
   if (strlen(fName) == 0)
      return -1;
   }
else {
   if (argc > 2)
      printf("Extra input ignored.\n");
   fNamePtr = argv[1];
   }

f = fopen(fNamePtr, "r");               /* open the file */
if (f == NULL) {
   printf("Could not open %s.\n", fNamePtr);
   return -1;
   }

ReservedWords();                        /* add the reserved words */
do {                                  /* collect the symbols in the file */
   GetToken();
   if (strlen(token))
      Insert(&symbols, 0);
   }
while (strlen(token));
PrintSymbols(symbols);                  /* print the symbol table */

fclose(f);                              /* close the file */
return 0;                               /* return with no error */
}
```

Lesson Seventeen
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 compared to most commercial programs, 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 of the paddle, we can check to see if the ball has hit the paddle by comparing the X coordinate (the horizontal position) of 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:

```
do {
   MoveBall();
   MovePaddle();
   }
while (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.

```
do {
   event = GetNextEvent(
      EVENTMASK,
      &myevent);
   MoveBall();
   MovePaddle();
   }
while (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 structure with a type of EventRecord. You need to include event.h, the event manager's header file, in your program to use GetNextEvent; event.h also pulls in the correct header file to define EventRecord.

When you are learning about a new structure, it is a good idea to actually look at the source code for the toolbox interface files to see how the structure is defined. Here is the definition for an event record:

```
struct EventRecord {
   int what;
   long message;
   long when;
   Point where;
   int modifiers;
   }
```

Most of these fields are of no interest to us at the moment, although we will use a couple more later. The what 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 message field varies, depending on what kind of event occurs. For a key press, for example, this field tells what key was pressed.

The when 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 where field. This field is actually a structure itself. The structure is a point, which is a structure used by the toolbox to hold both an x and y position at the

same time. 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 modifiers field contains still more information about the particular event that the event manager is reporting.

If you check the EventRecord structure in event.h, you will find a couple of differences between that structure and the one shown here. The structure in event.h uses types of Word and LongWord for int and long; these are just macros that replace int and long. You will also find some other fields after the modifiers field; these are used by other tool calls besides GetNextEvent which share the EventRecord structure with GetNextEvent. Since they are not filled in by GetNextEvent, we won't worry about them in our program.

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, defined in quickdraw.h, 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 sample program in listing 17.1 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 lesson 12 that also starts the event manager. The only other new call is GetPortRect, defined in quickdraw.h, while fills in a rectangle that tells you how big the graphics window is. This lets the program adapt to whatever size you set the graphics window to, instead of assuming some fixed size. The rectangle itself is yet another toolbox structure that contains the top, bottom, left and right coordinates of the window you are drawing in – or the entire graphics screen if your program is not using the window manager.

Listing 17.1

```
/* Move a paddle in the graphics window */

#include <quickdraw.h>
#include <event.h>
#include <memory.h>

#include <orca.h>

#define EVENTMASK 0x0F6E                /* GetNextEvent event mask */
#define SIZE 640                        /* graphics mode */

EventRecord myevent;                    /* current event record */
int maxX;                       /* max X distance the paddle can travel */
int paddlePosition = 0;             /* current X position of the paddle */
Rect screen;                            /* port rectangle */
```

```
void StartTools (void)

/* Start the tools                                                  */

{
handle memory;                          /* memory returned by NewHandle */
Rect r;                                 /* screen size */

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(256L,userid(),0xC015,0L); /* start up the event mgr */
EMStartUp((int) *memory, 0, 0, SIZE, 0, 200, userid());
FlushEvents(0xFFFF, 0);
}


void ShutDownTools (void)

/* Shut down the tools                                              */

{
EMShutDown();
endgraph();
}


void DrawPaddle (int position, int color)

/* Draw the paddle                                                  */
/*                                                                  */
/* Parameters:                                                      */
/*    position - position to draw the paddle                        */
/*    color - color of the paddle                                   */

#define WIDTH 70                        /* width of the paddle */
#define HEIGHT 3                        /* height of the paddle */
{
int y;                           /* position of the paddle on the screen */

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)   /* make sure we don't go off of the screen */
   position = maxX-WIDTH;
```

```
   if (position < 0)
      position = 0;
   y = screen.v2-12;                        /* find the paddle's y position */
   MoveTo(position,y);                      /* draw the paddle */
   LineTo(position,y);
   }


   void MovePaddle (void)

   /* Track and move the paddle                                          */
   /*                                                                    */
   /* Variables:                                                         */
   /*    paddlePosition - position of the paddle                         */
   /*    myevent - last event returned by GetNextEvent                   */

   {
   /* convert the point to our window */
   GlobalToLocal(&myevent.where);

   /* if the mouse moved, move the paddle */
   if (myevent.where.h != paddlePosition) {
      DrawPaddle(paddlePosition,3);
      paddlePosition = myevent.where.h;
      DrawPaddle(paddlePosition,0);
      }
   }


   void main (void)

   /* Main program                                                       */

   {
   int event;                     /* event flag; returned by GetNextEvent */

   StartTools();                            /* start the tools */
   GetPortRect(&screen);                    /* set the limit on the paddle */
   maxX = screen.h2;
   DrawPaddle(paddlePosition,0);            /* draw the initial paddle */

   do {                                     /* event loop */
      event = GetNextEvent(EVENTMASK, &myevent);
      MovePaddle();
      }
   while (!event);

   ShutDownTools();                         /* shut down the tools */
   }
```

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/C 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 do-while 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 return an integer result. 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 {
      event = GetNextEvent(
         EVENTMASK,
         &myevent);
      MovePaddle();
      }
   while (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 static variable, and initialize it to true. PlayAGame should return the value of the static variable, but set it to false. That way, PlayAGame returns true the first time it is called, but false the second time.

```
void PlayAGame (void)

{
static int play = 1;
int ret;

ret = play;
play = 0;
return ret;
}
```

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.what == mouseUpEvt)
   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 12, 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 17.2, expands the graphics window to the full size of the screen, and brings 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 header file, so you will need to add window.h to the list of tool header files you include. Stop now, and get your entire program to work with the stubs.

Listing 17.2

```
void ExpandGraph (void)

/* Expand the graphics window to full screen                    */

{
/* <<<>>> */
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 */
}


void ShrinkGraph (void)

/* Put the graphics window back in the corner                   */

{
/* <<<>>> */
MoveWindow(320, 115, GetPort());       /* move graph back to the corner */
SizeWindow(320, 85, GetPort());    /* make the window a reasonable size */
}
```

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. 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 subroutine to draw the original set of six rows of bricks on the screen. For consistency with the solution, call the subroutine 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:

```
#define ROWS 6
#define LEVELS 16

int points[ROWS][[LEVELS];
```

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:

```
#define ROWS 6
#define COLUMNS 16

int stillThere[ROWS][COLUMNS];
```

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);
printf("Score:");
MoveTo(560,190);
printf("Balls:");
```

The score and number of balls must be updated throughout the game, so I put the code to write these numbers in two subroutines, 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 when 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 when field itself is suitably random! Here is how I chose the position and direction of the ball:

```
if (event.when & 0x0001)
   dx = -4;
else
   dx = 4;
x = event.when % 640;
```

In effect, I have used a tiny part of the when field to decide if the ball should head to the left or right when it starts. The last line uses the when field 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 when 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 when 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 when field as a timer, we start be recording the value of the when field 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:

```
do {
   GetNextEvent(EVENTMASK,
      &myevent);
   timer =
      myevent.when-lastWhen;
   if (timer > PAUSE) {
      MoveBall();
      lastWhen += PAUSE;
      }
   MovePaddle();
   }
while (balls);
```

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.when. 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; in this program, I recorded the bottom of the lowest row of bricks. 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 17.3. 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 value 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.

```
if (y > brickY)
   return;
row = (brickY-y) / SPACING;
dispY = (brickY-y) % SPACING;
if ((row < ROWS)
   && (dispY <= THICKNESS) {
   column = x / WIDTH;
   dispX = x % WIDTH ;
   if (stillThere[row][column]) {
      /* <<<remove the brick>>> */
      /* <<<add in the score>>> */
      if (dispY == THICKNESS)
         /* <<<hit from above>>> */
      else if (dispY == 0)
         /* <<<hit from below>>> */
      else if (dispX < (WIDTH / 2))
         /* <<<hit from left>>> */
      else
         /* <<<hit from right>>> */
      }
   }
```

There is a lot packed into these few lines, so let's take a moment to look them over carefully. The first if check makes sure that the ball has at least reached the first row of bricks; this gets us out of the subroutine quickly if the ball is below the bricks and simplifies the rest of the calculations, all at the same time. The next line calculates the row that the ball is in, returning 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. The first if check returns right away until y drops below the value of brickY, so the first value we have to worry about is when y is 56. The value of brickY-y is 0, so the value of row gets set to 0. We have reached the first row of bricks, which we have labeled row 0. 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 of the 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: dispY is equal to 0

Listing 17.3

```
#define ROWS 6          /* # of rows of bricks */
#define COLUMNS 16      /* # of columns of bricks */
#define THICKNESS 6     /* thickness of a brick */
#define WIDTH 40        /* width of a brick */
#define SPACING 8       /* # of pixels between rows */

int x,y;                /* ball position */
int dx,dy;              /* ball velocity */
int brickY;             /* position of the bottom of the lowest brick */
```

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:

```
if (y > brickY)
   return;
row = (brickY-y) / SPACING;
dispY = (brickY-y) % SPACING;
if ((row < ROWS)
   && (dispY <= THICKNESS) {
   column = x / WIDTH;
   dispX = x % WIDTH ;
   if (stillThere[row][column]) {
      /* <<<remove the brick>>> */
      /* <<<add in the score>>> */
      if ((dispY == THICKNESS)
         || (dispY == 0))
         dy = -dy;
      else
         dx = -dx;
      }
   }
```

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.

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:

```
if (y > brickY)
   return;
row = (brickY-y) / SPACING;
dispY = (brickY-y) % SPACING;
if ((row < ROWS)
   && (dispY <= THICKNESS) {
   column = x / WIDTH;
```

```
dispX = x % WIDTH ;
if (stillThere[row][column]) {
   HitBrick(row,column);
   if ((dispY == THICKNESS)
      || (dispY == 0))
      dy = -dy;
   else
      dx = -dx;
   }
else if ((dispY == 0)
   || (dispY == THICKNESS))
   if (dispX == 0) {
      if (stillThere[row][column-1])
         {
         HitBrick(row,column-1);
         dy = -dy;
         }
      }
   else if (dispX == WIDTH)
      if (stillThere[row][column+1])
         {
         HitBrick(row,column+1);
         dy = -dy;
         }
}
```

Making the check this way presents a common problem, though. We can end up checking column number -1 or column number COLUMNS, 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 0 to COLUMNS-1. The leftmost and rightmost columns represent two fake columns which are not actually on the screen; these are initialized to false, so that the algorithm never reports that a brick has been hit. This is the way the solution handles the situation. Of course, if you use this method, you need to adjust all of the places that you have used the array stillThere, since the column numbers have changed; the leftmost column on the screen used to be represented by stillThere[row][0], and is now stillThere[row][1], and so on.

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);
printf(
   "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.

```
do
   GetNextEvent(EVENTMASK,
      &myevent);
while (myevent.what
   != 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, "Play a Game". For the other choice, use "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:

```
do {
   GetNextEvent(EVENTMASK,
      &myevent);
   MovePaddle();
   }
while (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. If you like, you can also add the line

```
#pragma optimize 9
```

before the first function in your program to kick in the most useful of ORCA/C's compiler optimizations, speeding up the program a little more.

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.

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/C conforms to the ANSI standard. 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 ANSI standard C, and on most other Cs, too. The reverse is also important: any book that uses ANSI C will work with ORCA/C. If you would like to learn more about programming, there are many fine books that meet this requirement. The best way to find books that fit your needs and interests is to visit a well-stocked bookstore and browse for a few hours.

If you would like to learn to program the toolbox, writing desktop programs with pull down menus and so forth, you need to study books that are specific to the Apple IIGS. 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. There are no good introductory books for learning to program the toolbox that use C. 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 Seventeen
The Complete Game

```
/* This program plays the game of break-out, a classic arcade    */
/* game.                                                         */

#pragma optimize 9

#include <quickdraw.h>
#include <event.h>
#include <memory.h>
#include <window.h>

#include <orca.h>

#include <stdio.h>

#define EVENTMASK 0x0F6E               /* GetNextEvent event mask */
#define SIZE 320                       /* graphics mode */
#define LETTERY 2           /* height of letters for score, ball count */
#define PAUSE 1                        /* 60ths of a sec. to pause */

EventRecord myevent;                   /* current event record */
Rect screen;                           /* port rectangle */
int score;                             /* current score */
long lastWhen;                         /* event timer */

                                       /* the paddle */
                                       /*------------*/
#define PADDLECOLOR 15                 /* paddle color */
#define PADDLEHEIGHT 3                 /* paddle height */
#define PADDLEWIDTH 35                 /* paddle width */
#define PADDLEY 184                    /* y position of the paddle */
#define EASY 2                         /* x velocity for easy spin */
#define HARD 3                         /* x velocity for hard spin */

#define AREA1 7                       /* sensitive areas of the paddle */
#define AREA2 14
#define AREA3 21
#define AREA4 28

int maxX;                        /* max X distance the paddle can travel */
int paddlePosition;                  /* current X position of the paddle */
```

```
                                        /* the bricks */
                                        /*------------*/
#define COLUMNS 16                      /* # of bricks in a row */
#define ROWS 6                          /* # of rows of bricks */
#define SPACING 8                       /* spacing of the rows */
#define STARTHEIGHT 56         /* starting distance to bottom of bricks */
#define THICKNESS 6                     /* thickness of a brick */
#define WIDTH 20                        /* width of a brick */

int brickY;                          /* disp to the bottom row of bricks */
int level;                              /* playing level */
int numBricks;                          /* # of bricks visible */
int stillThere[ROWS][COLUMNS+2];        /* brick array */

                                        /* the ball */
                                        /*----------*/
#define BALLCOLOR 15                    /* ball color */
#define BALLHEIGHT 3                    /* ball height; should be odd */
#define BALLWIDTH 3                     /* ball width; should be odd */
#define BALLDX (BALLWIDTH/2)            /* half width of ball */
#define BALLDY (BALLHEIGHT/2)           /* half height of ball */
#define SPEED 2                         /* vertical ball speed */

int balls;                              /* # of balls left */
int dx,dy;                              /* speed of the ball */
int x,y;                                /* position of the ball */

                                        /* button sizes */
                                        /*--------------*/
#define LEFT 110
#define RIGHT 210
#define TOP1 90
#define BOTTOM1 101
#define TOP2 105
#define BOTTOM2 116

/*-------------------------------------------------------------*/

void StartTools (void)

/* Start the tools                                             */

{
handle memory;                          /* memory returned by NewHandle */

startgraph(SIZE);                       /* initialize QuickDraw */
memory = NewHandle(256L,userid(),0xC015,0L); /* start up the event mgr */
EMStartUp((int) *memory, 0, 0, SIZE, 0, 200, userid());
FlushEvents(0xFFFF, 0);
}
```

```
void ShutDownTools (void)

/* Shut down the tools                                              */

{
EMShutDown();
endgraph();
}


void DrawPaddle (int position, int color)

/* Draw the paddle                                                  */
/*                                                                  */
/* Parameters:                                                      */
/*    position - position to draw the paddle                        */
/*    color - color of the paddle                                   */

{
int y;                          /* position of the paddle on the screen */

                               /* 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);
}


void MovePaddle (void)

/* Track and move the paddle                                        */
/*                                                                  */
/* Variables:                                                       */
/*    paddlePosition - position of the paddle                       */
/*    myevent - last event returned by GetNextEvent                 */

{
/* convert the point to our window */
GlobalToLocal(&myevent.where);

/* make sure we don't go off of the screen */
if (myevent.where.h+PADDLEWIDTH > maxX)
   myevent.where.h = maxX-PADDLEWIDTH;
```

```
/* if the mouse moved, move the paddle */
if (myevent.where.h != paddlePosition) {
   DrawPaddle(paddlePosition,0);
   paddlePosition = myevent.where.h;
   DrawPaddle(paddlePosition, PADDLECOLOR);
   }
}


void DrawBrick (int row, int column, int color)

/* 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             */

{
Rect r;                                 /* brick's rectangle */

SetPenMode(0);                          /* get ready to draw */
SetSolidPenPat(color);
SetPenSize(1,1);
r.h1 = column*WIDTH;                    /* set up the brick's rectangle */
r.h2 = r.h1+WIDTH;
r.v2 = brickY - row*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);
}
```

```
void DrawBricks (void)

/* Draw a set of bricks                                          */

{
int colors[ROWS];                       /* brick colors */
int column;                             /* loop variable */
int row;                                /* loop variable */
Rect r;                                 /* brick rectangle */

numBricks = ROWS*COLUMNS;               /* set the brick count */
colors[0] = 7;                         /* fill in the brick color array */
colors[1] = 6;
colors[2] = 9;
colors[3] = 10;
colors[4] = 13;
colors[5] = 12;
for (row = 0; row < ROWS; ++row) {      /* draw the bricks */
   for (column = 0; column < COLUMNS; ++column) {
      DrawBrick(row, column, colors[row]);
      stillThere[row][column+1] = 1;
      }
   stillThere[row][0] = 0;
   stillThere[row][COLUMNS+1] = 0;
   }
}


void WriteBalls (void)

/* Draw the number of balls left                                 */
/*                                                               */
/* Variables:                                                    */
/*    balls - number of balls left                               */

#define BALLX 290
{
MoveTo(BALLX, LETTERY);
printf("%d          ", balls);
}
```

```
void WriteScore (void)

/* Draw the current score                                          */
/*                                                                 */
/* Variables:                                                      */
/*    score - score to draw                                        */

#define SCOREX 50

{
MoveTo(SCOREX, LETTERY);
printf("%d          ", score);
}


void DrawBall (void)

/* 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.                                           */

{
SetSolidPenPat(BALLCOLOR);
SetPenMode(2);
SetPenSize(BALLWIDTH, BALLHEIGHT);
MoveTo(x-BALLDX, y-BALLDY);
LineTo(x-BALLDX, y-BALLDY);
}
```

```
void StartBall (void)

/* Start a ball                                                    */
/*                                                                 */
/* Variables:                                                      */
/*    x,y - position of the ball                                   */
/*    dx,dy - speed of the ball                                    */

{
if (myevent.when & 0x0001)              /* set the speed, position */
   dx = -EASY;
else
   dx = EASY;
x = myevent.when % screen.h2;
dy = SPEED;
y = brickY + 4;

DrawBall();                             /* draw the ball */

GetNextEvent(EVENTMASK, &myevent);      /* set the timer */
lastWhen = myevent.when;
}


void WaitForClick (void)

/* Pause until the player is ready for a ball                      */

{
static Rect r = {90, 0, 100, 320};      /* used to erase the message */

MoveTo(65,100);                         /* write the message */
printf("Click for the next ball");
do {                                    /* wait for the click */
   GetNextEvent(EVENTMASK, &myevent);
   MovePaddle();
   }
while (myevent.what != mouseUpEvt);
SetSolidPenPat(0);
SetPenMode(0);
PaintRect(&r);
}
```

```
void HitBrick (int row, int column)

/* Handle a hit brick                                          */
/*                                                             */
/* Parameters:                                                 */
/*    row,column - brick that was hit                          */

{
Rect ball;                              /* ball rectangle */

stillThere[row][column+1] = 0;          /* 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 += (row+1+level)*5;               /* add in the score */
WriteScore();
--numBricks;                            /* see if they are all gone */
if (numBricks == 0) {
   DrawBall();
   ++balls;
   WriteBalls();
   brickY += SPACING;
   ++level;
   DrawBricks();
   WaitForClick();
   StartBall();
   }
GetNextEvent(EVENTMASK, &myevent);      /* reset the timer */
lastWhen = myevent.when;
}
```

```
void CheckBricks (void)

/* Move the ball.                                                    */
/*                                                                   */
/* Variables:                                                        */
/*    x,y - position of the ball                                     */
/*    numBricks - # of bricks left                                   */

{
int row,column;                          /* brick row,column */
int dispX,dispY;                         /* position along the brick */

if (y > brickY)                  /* quit now if we are below the bricks */
   return;
row = (brickY-y) / SPACING;           /* find the vertical brick values */
dispY = (brickY-y) % SPACING;
if ((row < ROWS) && (dispY <= THICKNESS)) {
   column = x / WIDTH;              /* find the horizontal brick values */
   dispX = x % WIDTH;
                                         /* check for a hit */
   if (stillThere[row][column+1]) {
      HitBrick(row,column);
      if ((dispY == 0) || (dispY == THICKNESS))
         dy = -dy;
      else
         dx = -dx;
      }
   else if ((dispY == 0) || (dispY == THICKNESS))
      if (dispX == 0) {
         if (stillThere[row][column]) {
            HitBrick(row,column-1);
            dy = -dy;
            }
         }
      else if (dispX == WIDTH)
         if (stillThere[row][column+2]) {
            HitBrick(row,column+1);
            dy = -dy;
            }
   }
}
```

```
void GetANewBall (Rect *oldBall)

/* Missed; get a new ball                                          */
/*                                                                 */
/* Parameters:                                                     */
/*    oldBall - ptr to MoveBall's old ball position                */

{
--balls;                                /* reduce the number of balls */
WriteBalls();
if (balls) {
   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;
   }
}


void MoveBall (void)

/* Move the ball                                                   */
/*                                                                 */
/* Variables:                                                      */
/*    x,y - position of the ball                                   */
/*    dx,dy - speed of the ball                                    */

{
Rect oldBall, newBall;                  /* ball rectangles */
int px;                               /* disp of ball on paddle surface */

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 += dx;                                /* move the ball */
if (x < 0) {
   x = 0;
   dx = -dx;
   }
else if (x > screen.h2) {
   x = screen.h2;
   dx = -dx;
   }
y += dy;
```

```
if (y < 0) {
   y = 0;
   dy = -dy;
   }
else if (y >= PADDLEY) {
   if ((x < paddlePosition) || (x > paddlePosition+PADDLEWIDTH))
      GetANewBall(&oldBall);
   else {
      px = x-paddlePosition;
      if (px < AREA1)
         dx = -HARD;
      else if (px < AREA2)
         dx = -EASY;
      else if (px < AREA3)
         dx = 0;
      else if (px < AREA4)
         dx = EASY;
      else
         dx = HARD;
      dy = -dy;
      y = PADDLEY;
      }
   }
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 */
}
```

```
int WhichButton (void)

/* See which button the mouse is in                                  */
/*                                                                   */
/* Variables:                                                        */
/*    myevent.where - location of mouse at mouseup                   */
/*    myevent.what - kind of event                                   */

{
int result = 0;                         /* value to return */

if (myevent.what == mouseUpEvt)
   if (myevent.where.h >= LEFT)
      if (myevent.where.h <= RIGHT)
         if (myevent.where.v >= TOP1)
            if (myevent.where.v <= BOTTOM2)
               if (myevent.where.v <= BOTTOM1)
                  result = 1;
               else if (myevent.where.v >= TOP2)
                  result = 2;
return result;
}


int PlayAGame (void)

/* See if the player wants to play a game or quit.                   */
/*                                                                   */
/* Returns: True to play a game, else false.                         */

{
Rect r;                                 /* rect inclosing the buttons */

MoveTo(LEFT+10, BOTTOM1-2);             /* draw the messages */
printf("Play a Game");
MoveTo(LEFT+30,BOTTOM2-2);
printf("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 */
do {
   GetNextEvent(EVENTMASK, &myevent);
   GlobalToLocal(&myevent.where);
   MovePaddle();
   }
while (WhichButton() == 0);
HideCursor();
r.h1 = LEFT;                           /* erase the messages */
r.h2 = RIGHT;
r.v1 = TOP1;
r.v2 = BOTTOM2;
SetSolidPenPat(0);
return WhichButton() == 1;             /* set the return value */
}


void InitScreen (void)

/* Draw the initial screen                                    */

{
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);
balls = 3;                             /* give the player 3 balls */
level = 1;                             /* play level = 1 */
SetForeColor(11);                /* draw the initial score, ball count */
SetBackColor(0);
MoveTo(0, LETTERY);
printf("Score:");
MoveTo(240, LETTERY);
printf("Balls:");
score = 0;
WriteScore();
WriteBalls();
}
```

```
void main (void)

/* Main program                                                    */

{
int time;                              /* timer */

StartTools();                          /* start the tools */
InitCursor();                          /* set up the cursor */
HideCursor();
GetPortRect(&screen);                  /* set the limit on the paddle */
maxX = screen.h2;
InitScreen();                         /* give them something to look at */

while (PlayAGame()) {
   InitScreen();                       /* set up the screen */
   StartBall();                        /* start a ball */
   do {                                /* event loop */
      GetNextEvent(EVENTMASK, &myevent);
      time = myevent.when - lastWhen;
      if (time > PAUSE) {
         MoveBall();
         CheckBricks();
         lastWhen += PAUSE;
         }
      MovePaddle();
      }
   while (balls);
   DrawBall();                         /* erase the last ball */
   }

ShutDownTools();                       /* shut down the tools */
}
```