Programming Concepts using ‘ C’

Table of Content

Module 1: An Introduction of C

Module Objective:

The participants will have an understanding of

- Structure of a 'C' program
- how to execute a 'C' program

Chapter 1: History of C language
History of B language
Founders of C

Chapter 2: Structure of C program
Header file
Main function

Chapter 3: Compilation Process
Preprocessor
Compiler
Linker

Chapter 4: Exercise

An Introduction of C

History of the C Language

You might be wondering about the origin of the C language and where it got its name. C was created by Dennis Ritchie at the Bell Telephone Laboratories in 1972. The language wasn't created for the fun of it, but for a specific purpose: to design the UNIX operating system (which is used on many computers). From the beginning, C was intended to be useful--to allow busy programmers to get things done.

Because C is such a powerful and flexible language, its use quickly spread beyond Bell Labs. Programmers everywhere began using it to write all sorts of programs. Soon, however, different organizations began utilizing their own versions of C, and subtle differences between implementations started to cause programmers headaches. In response to this problem, the American National Standards Institute (ANSI) formed a committee in 1983 to establish a standard definition of C, which became known as ANSI Standard C. With few exceptions, every modern C compiler has the ability to adhere to this standard.

Now, what about the name? The C language is so named because its predecessor was called B. The B language was developed by Ken Thompson of Bell Labs. You should be able to guess why it was called B.

Why Use C?

In today's world of computer programming, there are many high-level languages to choose from, such as C, Pascal, BASIC, and Java. These are all excellent languages suited for most programming tasks. Even so, there are several reasons why many computer professionals feel that C is at the top of the list:

C is a powerful and flexible language. What you can accomplish with C is limited only by your imagination. The language itself places no constraints on you. C is used for projects as diverse as operating systems, word processors, graphics, spreadsheets, and even compilers for other languages.

- C is a popular language preferred by professional programmers. As a result, a wide variety of C compilers and helpful accessories are available.

- C is a portable language. *Portable* means that a C program written for one computer system (an IBM PC, for example) can be compiled and run on another system (a DEC VAX system, perhaps) with little or no modification. Portability is enhanced by the ANSI standard for C, the set of rules for C compilers.

- C is a language of few words, containing only a handful of terms, called *keywords,* which serve as the base on which the language's functionality is built. You might think that a language with more keywords (sometimes called *reserved words*) would be more powerful. This isn't true. As you program with C, you will find that it can be programmed to do any task.

- C is modular. C code can (and should) be written in routines called *functions*. These functions can be reused in other applications or programs. By passing pieces of information to the functions, you can create useful, reusable code.

As these features show, C is an excellent choice for your first programming language. What about C++? You might have heard about C++ and the programming technique called *object- oriented programming*. Perhaps you're wondering what the differences are between C and C++ and whether you should be teaching yourself C++ instead of C.

Not to worry! C++ is a superset of C, which means that C++ contains everything C does, plus new additions for object-oriented programming. If you do go on to learn C++, almost everything you learn about C will still apply to the C++ superset. In learning C, you are not only learning one of today's most powerful and popular programming languages, but you are also preparing yourself for object-oriented programming.

Another language that has gotten lots of attention is Java. Java, like C++, is based on C. If later you decide to learn Java, you will find that almost everything you learned about C can be applied.

Preparing to Program

You should take certain steps when you're solving a problem. First, you must define the problem. If you don't know what the problem is, you can't find a solution! Once you know what the problem is, you can devise a plan to fix it. Once you have a plan, you can usually implement it. Once the plan is implemented, you must test the results to see whether the problem is solved. This same logic can be applied to many other areas, including programming.

When creating a program in C (or for that matter, a computer program in any language), you should follow a similar sequence of steps:

1. Determine the objective(s) of the program.
2. Determine the methods you want to use in writing the program.

3. Create the program to solve the problem.
4. Run the program to see the results.

An example of an objective (see step 1) might be to write a word processor or database program. A much simpler objective is to display your name on the screen. If you didn't have an objective, you wouldn't be writing a program, so you already have the first step done.

The second step is to determine the method you want to use to write the program. Do you need a computer program to solve the problem? What information needs to be tracked? What formulas will be used? During this step, you should try to determine what you need to know and in what order the solution should be implemented.

As an example, assume that someone asks you to write a program to determine the area inside a circle. Step 1 is complete, because you know your objective: determine the area inside a circle. Step 2 is to determine what you need to know to ascertain the area. In this example, assume that the user of the program will provide the radius of the circle. Knowing this, you can apply the formula pr2 to obtain the answer. Now you have the pieces you need, so you can continue to steps 3 and 4, which are called the Program Development Cycle.

Compilation Process

The Program Development Cycle has its own steps. In the first step, you use an editor to create a disk file containing your source code. In the second step, you compile the source code to create an object file. In the third step, you link the compiled code to create an executable file. The fourth step is to run the program to see whether it works as originally planned.

Creating the Source Code

Source code is a series of statements or commands that are used to instruct the computer to perform your desired tasks. As mentioned, the first step in the Program Development Cycle is to enter source code into an editor. For example, here is a line of C source code:
printf("Hello, Mom!");
This statement instructs the computer to display the message Hello, Mom! on-screen. (For now, don't worry about how this statement works.)

Using an Editor

Most compilers come with a built-in editor that can be used to enter source code; however, some don't. Consult your compiler manuals to see whether your compiler came with an editor. If it didn't, many alternative editors are available.
Most computer systems include a program that can be used as an editor. If you're using a UNIX system, you can use such editors as ed, ex, edit, emacs, or vi. If you're using Microsoft Windows, Notepad is available. If you're using MS/DOS 5.0 or later, you can use Edit. If you're using a version of DOS before 5.0, you can use Edlin. If you're using PC/DOS 6.0 or later, you can use E. If you're using OS/2, you can use the E and EPM editors.
Most word processors use special codes to format their documents. These codes can't be read correctly by other programs. The American Standard Code for Information Interchange (ASCII) has specified a standard text format that nearly any program, including C, can use. Many word processors, such as WordPerfect, AmiPro, Word, WordPad, and WordStar, are capable of saving source files in ASCII form (as a text file rather than a document file). When you want to save a word processor's file as an ASCII file, select the ASCII or text option when saving.

If none of these editors is what you want to use, you can always buy a different editor. There are packages, both commercial and shareware, that have been designed specifically for entering source code.

❑: To find alternative editors, you can check your local computer store or computer mail-order catalogs. Another place to look is in the ads in computer programming magazines.

When you save a source file, you must give it a name. The name should describe what the program does. In addition, when you save C program source files, give the file a .C extension. Although you could give your source file any name and extension, .C is recognized as the appropriate extension to use.

Compiler

Although you might be able to understand C source code (at least, after reading this book you will be able to), your computer can't. A computer requires digital, or binary, instructions in what is called machine language. Before your C program can run on a computer, it must be translated from source code to machine language. This translation, the second step in program development, is performed by a program called a compiler. The compiler takes your source code file as input and produces a disk file containing the machine language instructions that correspond to your source code statements. The machine language instructions created by the compiler are called object code, and the disk file containing them is called an object file.

❑: This book covers ANSI Standard C. This means that it doesn't matter which C compiler you use, as long as it follows the ANSI Standard.

Each compiler needs its own command to be used to create the object code. To compile, you typically use the command to run the compiler followed by the source filename. The following are examples of the commands issued to compile a source file called RADIUS.C using various DOS/Windows compilers:

Compiler	**Command**
Microsoft C	cl radius.c
Borland's Turbo C	tcc radius.c
Borland C	bcc radius.c
Zortec C	ztc radius.c

To compile RADIUS.C on a UNIX machine, use the following command:

```
cc radius.c
```

Consult the compiler manual to determine the exact command for your compiler.
If you're using a graphical development environment, compiling is even simpler. In most graphical environments, you can compile a program listing by selecting the compile icon or selecting something from a menu. Once the code is compiled, selecting the run icon or selecting something from a menu will execute the program. You should check your compiler's manuals for specifics on compiling and running a program.

After you compile, you have an object file. If you look at a list of the files in the directory or folder in which you compiled, you should find a file that has the same name as your source file, but with an .OBJ (rather than a .C) extension. The .OBJ extension is recognized as an object file and is used by the linker. On UNIX systems, the compiler creates object files with an extension of .O instead of .OBJ.

Linker

One more step is required before you can run your program. Part of the C language is a function library that contains object code (code that has already been compiled) for predefined functions. A predefined function contains C code that has already been written and is supplied in a ready-to-use form with your compiler package.
The printf() function used in the previous example is a library function. These library functions perform frequently needed tasks, such as displaying information on-screen and reading data from disk files. If your program uses any of these functions (and hardly a program exists that doesn't use at least one), the object file produced when your source code was compiled must be combined with object code from the function library to create the final executable program. (Executable means that the program can be run, or executed, on your computer.) This process is called linking, and it's performed by a program called (you guessed it) a linker.

Completing the Development Cycle

Once your program is compiled and linked to create an executable file, you can run it by entering its name at the system prompt or just like you would run any other program. If you run the program and receive results different from what you thought you would, you need to go back to the first step. You must identify what caused the problem and correct it in the source code. When you make a change to the source code, you need to recompile and relink the program to create a corrected version of the executable file. You keep following this cycle until you get the program to execute exactly as you intended.

One final note on compiling and linking: Although compiling and linking are mentioned as two separate steps, many compilers, such as the DOS compilers mentioned earlier, do both as one step. Regardless of the method by which compiling and linking are accomplished, understand that these two processes, even when done with one command, are two separate actions.

The C Development Cycle

Step 1: Use an editor to write your source code. By tradition, C source code files have the extension .C (for example, MYPROG.C, DATABASE.C, and so on).

Step 2: Compile the program using a compiler. If the compiler doesn't find any errors in the program, it produces an object file. The compiler produces object files with a .OBJ extension and the same name as the source code file (for example, MYPROG.C compiles to MYPROG.OBJ). If the compiler finds errors, it reports them. You must return to step 1 to make corrections in your source code.

Step 3: Link the program using a linker. If no errors occur, the linker produces an executable program located in a disk file with an .EXE extension and the same name as the object file (for example, MYPROG.OBJ is linked to create MYPROG.EXE).

Step 4: Execute the program. You should test to determine whether it functions properly. If not, start again with step 1 and make modifications and additions to your source code.

For all but the simplest programs, you might go through this sequence many times before finishing your program. Even the most experienced programmers can't sit down and write a complete, error-free program in just one step! Because you'll be running through the edit-compile-link-test cycle many times, it's important to become familiar with your tools: the editor, compiler, and linker.

Your First C Program

You're probably eager to try your first program in C. To help you become familiar with your compiler, here's a quick program for you to work through. You might not understand everything at this point, but you should get a feel for the process of writing, compiling, and running a real C program.
This demonstration uses a program named HELLO.C, which does nothing more than display the words Hello, World! on-screen. This program, a traditional introduction to C programming, is a good one for you to learn. The source code for HELLO.C is in Listing 1.1. When you type in this listing, you won't include the line numbers or colons.

Listing 1.1. HELLO.C.

```
1: #include <stdio.h>
2:
3: main()
4: {
5:    printf("Hello, World!\n");
6:    return 0;
7: }
```

Be sure that you have installed your compiler as specified in the installation instructions provided with the software. Whether you are working with UNIX, DOS, or any other operating system, make sure you understand how to use the compiler and editor of your choice. Once your compiler and editor are ready, follow these steps to enter, compile, and execute HELLO.C.

Entering and Compiling HELLO.C

To enter and compile the HELLO.C program, follow these steps:

1. Make active the directory your C programs are in and start your editor. As mentioned previously, any text editor can be used, but most C compilers (such as Borland's Turbo C++ and Microsoft's Visual C/C++) come with an integrated development environment (IDE) that lets you enter, compile, and link your programs in one convenient setting. Check the manuals to see whether your compiler has an IDE available.
2. Use the keyboard to type the HELLO.C source code exactly as shown in Listing 1.1. Press Enter at the end of each line.
   : Don't enter the line numbers or colons. These are for reference only.
2. Save the source code. You should name the file HELLO.C.
3. Verify that HELLO.C is on disk by listing the files in the directory or folder. You should see HELLO.C within this listing.
4. Compile and link HELLO.C. Execute the appropriate command specified by your compiler's manuals. You should get a message stating that there were no errors or warnings.
5. Check the compiler messages. If you receive no errors or warnings, everything should be okay. If you made an error typing the program, the compiler will catch it and display an error message. For example, if you misspelled the word printf as prntf, you would see a message similar to the following:
   Error: undefined symbols:_prntf in hello.c (hello.OBJ)
6. Go back to step 2 if this or any other error message is displayed. Open the HELLO.C file in your editor. Compare your file's contents carefully with Listing 1.1, make any necessary corrections, and continue with step 3.
7. Your first C program should now be compiled and ready to run. If you display a directory listing of all files named HELLO (with any extension), you should see the following:
   HELLO.C, the source code file you created with your editor
   HELLO.OBJ or HELLO.O, which contains the object code for HELLO.C
   HELLO.EXE, the executable program created when you compiled and linked HELLO.C
8. To execute, or run, HELLO.EXE, simply enter hello. The message Hello, World! is displayed on-screen.

Congratulations! You have just entered, compiled, and run your first C program. Admittedly, HELLO.C is a simple program that doesn't do anything useful, but it's a start. In fact, most of today's expert C programmers started learning C in this same way--by compiling HELLO.C-- so you're in good company.

Compilation Errors

A compilation error occurs when the compiler finds something in the source code that it can't compile. A misspelling, typographical error, or any of a dozen other things can cause the compiler to choke. Fortunately, modern compilers don't just choke; they tell you what they're choking on and where it is! This makes it easier to find and correct errors in your source code. This point can be illustrated by introducing a deliberate error into HELLO.C. If you worked through that example (and you should have), you now have a copy of HELLO.C on your disk.

Using your editor, move the cursor to the end of the line containing the call to printf(), and erase the terminating semicolon. HELLO.C should now look like Listing 1.2.

Listing 1.2. HELLO.C with an error.

```
1: #include <stdio.h>
2:
3: main()
4: {
5:    printf("Hello, World!")
6:    return 0;
7: }
```

Next, save the file. You're now ready to compile it. Do so by entering the command for your compiler. Because of the error you introduced, the compilation is not completed. Rather, thecompiler displays a message similar to the following:

hello.c(6) : Error: `;' expected
Looking at this line, you can see that it has three parts:
hello.c The name of the file where the error was found
(6) :

The

line number where the error was found

Error: `;' expected A description of the error

This message is quite informative, telling you that in line 6 of HELLO.C the compiler expected to find a semicolon but didn't. However, you know that the semicolon was actually omitted from line 5, so there is a discrepancy. You're faced with the puzzle of why the compiler reports an error in line 6 when, in fact, a semicolon was omitted from line 5. The answer lies in the fact that C doesn't care about things like breaks between lines. The semicolon that belongs after the printf() statement could have been placed on the next line (although doing so would be bad programming practice). Only after encountering the next command (return) in line 6 is the compiler sure that the semicolon is missing. Therefore, the compiler reports that the error is in line 6.

This points out an undeniable fact about C compilers and error messages. Although the compiler is very clever about detecting and localizing errors, it's no Einstein. Using your knowledge of the C language, you must interpret the compiler's messages and determine the actual location of any errors that are reported. They are often found on the line reported by the compiler, but if not, they are almost always on the preceding line. You might have a bit of trouble finding errors at first, but you should soon get better at it.

❑: The errors reported might differ depending on the compiler. In most cases, the error message should give you an idea of what or where the problem is. Before leaving this topic, let's look at another example of a compilation error. Load HELLO.C into your editor again and make the following changes:

1. Replace the semicolon at the end of line 5.
2. Delete the double quotation mark just before the word Hello.

Save the file to disk and compile the program again. This time, the compiler should display error messages similar to the following:
hello.c(5) : Error: undefined identifier `Hello'
hello.c(7) : Lexical error: unterminated string
Lexical error: unterminated string

Lexical error: unterminated string
Fatal error: premature end of source file
The first error message finds the error correctly, locating it in line 5 at the word Hello. The error message undefined identifier means that the compiler doesn't know what to make of the word Hello, because it is no longer enclosed in quotes. However, what about the other four errors that are reported? These errors, the meaning of which you don't need to worry about now, illustrate the fact that a single error in a C program can sometimes cause multiple error messages.

The lesson to learn from all this is as follows: If the compiler reports multiple errors, and you can find only one, go ahead and fix that error and recompile. You might find that your single correction is all that's needed, and the program will compile without errors.

Linker Error Messages

Linker errors are relatively rare and usually result from misspelling the name of a C library function. In this case, you get an Error: undefined symbols: error message, followed by the misspelled name (preceded by an underscore). Once you correct the spelling, the problem should go away.

Summary
After reading this chapter, you should feel confident that selecting C as your programming language is a wise choice. C offers an unparalleled combination of power, popularity, and portability. These factors, together with C's close relationship to the C++ object-oriented language as well as Java, make C unbeatable.

This chapter explained the various steps involved in writing a C program--the process known as program development. You should have a clear grasp of the edit-compile-link-test cycle, as well as the tools to use for each step.

Errors are an unavoidable part of program development. Your C compiler detects errors in your source code and displays an error message, giving both the nature and the location of the error. Using this information, you can edit your source code to correct the error.

Remember, however, that the compiler can't always accurately report the nature and location of an error. Sometimes you need to use your knowledge of C to track down exactly what is causing a given error message

Components of a C Program

Every C program consists of several components combined in a certain way. Most of this book is devoted to explaining these various program components and how you use them. To get the overall picture, however, you should begin by seeing a complete (though small) C program with all its components identified. Today you will learn

- About a short C program and its components
- The purpose of each program component
- How to compile and run a sample program

A Short C Program

Listing 2.1 presents the source code for MULTIPLY.C. This is a very simple program. All it does is input two numbers from the keyboard and calculate their product. At this stage, don't worry about understanding the details of the program's workings. The point is to gain some familiarity with the parts of a C program so that you can better understand the listings presented later in this book.

Before looking at the sample program, you need to know what a function is, because functions are central to C programming. A function is an independent section of program code that performs a certain task and has been assigned a name. By referencing a function's name, your program can execute the code in the function. The program also can send information, called arguments, to the function, and the function can return information to the main part of the program. The two types of C functions are library functions, which are a part of the C compiler package, and user-defined functions, which you, the programmer, create. You will learn about both types of functions in this book.

❑ that, as with all the listings in this book, the line numbers in Listing 2.1 are not part of the program. They are included only for identification purposes, so don't type them.

Listing 2.1. MULTIPLY.C.

```
1:  /* Program to calculate the product of two numbers. */
2:  #include <stdio.h>
3:
4:  int a,b,c;
5:
```

```
6:  int product(int x, int y);
7:
8:  main()

9:  {
10:     /* Input the first number */
11:     printf("Enter a number between 1 and 100: ");
12:     scanf("%d", &a);
13:
14:     /* Input the second number */
15:     printf("Enter another number between 1 and 100: ");
16:     scanf("%d", &b);
17:
18:     /* Calculate and display the product */
19:     c = product(a, b);
20:     printf ("%d times %d = %d\n", a, b, c);
21:
22:     return 0;
23: }
24:
25: /* Function returns the product of its two arguments */
26: int product(int x, int y)
27: {
28:     return (x * y);
29: }
Enter a number between 1 and 100: 35
Enter another number between 1 and 100: 23
35 times 23 = 805[
```

The Program's Components

The following sections describe the various components of the preceding sample program. Line numbers are included so that you can easily identify the program parts being discussed.

The main() Function (Lines 8 Through 23)

The only component that is required in every C program is the main() function. In its simplest form, the main() function consists of the name main followed by a pair of empty parentheses (()) and a pair of braces ({}). Within the braces are statements that make up the main body of the program. Under normal circumstances, program execution starts at the first statement in main() and terminates at the last statement in main().

The #include Directive (Line 2)

The #include directive instructs the C compiler to add the contents of an include file into your program during compilation. An include file is a separate disk file that contains information needed by your program or the compiler. Several of these files (sometimes called header files) are supplied with your compiler. You never need to modify the information in these files; that's why they're kept separate from your source code. Include files should all have an .H extension (for example, STDIO.H).

You use the #include directive to instruct the compiler to add a specific include file to your program during compilation. The #include directive in this sample program means "Add the contents of the file STDIO.H." Most C programs require one or more include files. More information about include files is presented on Day 21, "Advanced Compiler Use."

The Variable Definition (Line 4)

A variable is a name assigned to a data storage location. Your program uses variables to store various kinds of data during program execution. In C, a variable must be defined before it can be used. A variable definition informs the compiler of the variable's name and the type of data it is to hold. In the sample program, the definition on line 4, int a,b,c;, defines three variables--named a, b, and c--that will each hold an integer value.

The Function Prototype (Line 6)

A function prototype provides the C compiler with the name and arguments of the functions contained in the program. It must appear before the function is used. A function prototype is distinct from a function definition, which contains the actual statements that make up the function. (Function definitions are discussed in more detail later in this chapter.)

Program Statements (Lines 11, 12, 15, 16, 19, 20, 22, and 28)

The real work of a C program is done by its statements. C statements display information on- screen, read keyboard input, perform mathematical operations, call functions, read disk files, and carry out all the other operations that a program needs to perform. Most of this book is devoted to teaching you the various C statements. For now, remember that in your source code, C statements are generally written one per line and always end with a semicolon. The statements in MULTIPLY.C are explained briefly in the following sections.

printf()

The printf() statement (lines 11, 15, and 20) is a library function that displays information on- screen. The printf() statement can display a simple text message (as in lines 11 and 15) or a message and the value of one or more program variables (as in line 20).

scanf()

The scanf() statement (lines 12 and 16) is another library function. It reads data from the keyboard and assigns that data to one or more program variables.

The program statement on line 19 calls the function named product(). In other words, it executes the program statements contained in the function product(). It also sends the arguments a and b to the function. After the statements in product() are completed, product() returns a value to the program. This value is stored in the variable named c.

return

Lines 22 and 28 contain return statements. The return statement on line 28 is part of the function product(). It calculates the product of the variables x and y and returns the result to the program statement that called product(). The return statement on line 22 returns a value of 0 to the operating system just before the program ends.

The Function Definition (Lines 26 Through 29)

A function is an independent, self-contained section of code that is written to perform a certain task. Every function has a name, and the code in each function is executed by including that function's name in a program statement. This is known as calling the function.

The function named product(), in lines 26 through 29, is a user-defined function. As the name implies, user-defined functions are written by the programmer during program development. This function is simple. All it does is multiply two values and return the answer to the program that called it. On Day 5, "Functions: The Basics," you will learn that the proper use of functions is an important part of good C programming practice.

❑ that in a real C program, you probably wouldn't use a function for a task as simple as multiplying two numbers. I've done this here for demonstration purposes only.

C also includes library functions that are a part of the C compiler package. Library functions perform most of the common tasks (such as screen, keyboard, and disk input/output) your program needs. In the sample program, printf() and scanf() are library functions.

Program Comments (Lines 1, 10, 14, 18, and 25)

Any part of your program that starts with /* and ends with */ is called a comment. The compiler ignores all comments, so they have absolutely no effect on how a program works. You can put anything you want into a comment, and it won't modify the way your program operates. A comment can span part of a line, an entire line, or multiple lines. Here are three examples:

```
/* A single-line comment */
int a,b,c; /* A partial-line comment */
/* a comment
spanning
multiple lines */
```

However, you shouldn't use nested comments (in other words, you shouldn't put one comment within another). Most compilers would not accept the following:

```
/*
/* Nested comment */
*/
```

Some compilers do allow nested comments. Although this feature might be tempting to use, you should avoid doing so. Because one of the benefits of C is portability, using a feature such as nested comments might limit the portability of your code. Nested comments also might lead to hard-to-find problems.
Many beginning programmers view program comments as unnecessary and a waste of time. This is a mistake! The operation of your program might be quite clear while you're writing it--particularly when you're writing simple programs. However, as your programs become larger and more complex, or when you need to modify a program you wrote six months ago, you'll find comments invaluable. Now is the time to develop the habit of using comments liberally to document all your programming structures and operations.

```
// This entire line is a comment
int x;  // Comment starts with slashes.
```

The two forward slashes signal that the rest of the line is a comment. Although many C compilers support this form of comment, you should avoid it if you're interested in portability. DO add abundant comments to your program's source code, especially near statements or functions that could be unclear to you or to someone who might have to modify it later.
DON'T add unnecessary comments to statements that are already clear. For example, entering

```
/* The following prints Hel o World! on the screen */
printf("Hello World!");
```

might be going a little too far, at least once you're completely comfortable with the printf() function and how it works.
DO learn to develop a style that will be helpful. A style that's too lean or cryptic doesn't help, nor does one that's so verbose that you're spending more time commenting than programming!

Braces (Lines 9, 23, 27, and 29)

You use braces ({}) to enclose the program lines that make up every C function--including the main() function. A group of one or more statements enclosed within braces is called a block. As you will see in later chapters, C has many uses for blocks.

Running the Program

Take the time to enter, compile, and run MULTIPLY.C. It provides additional practice in using your editor and compiler. Recall these steps from Day 1, "Getting Started with C":

1. Make your programming directory current.
2. Start your editor.
3. Enter the source code for MULTIPLY.C exactly as shown in Listing 2.1, but be sure to omit the line numbers and colons.
4. Save the program file.
5. Compile and link the program by entering the appropriate command(s) for your compiler. If no error messages are displayed, you can run the program by entering multiply at the command prompt.
6. If one or more error messages are displayed, return to step 2 and correct the errors.

A on Accuracy

A computer is fast and accurate, but it also is completely literal. It doesn't know enough to correct your simplest mistake; it takes everything you enter exactly as you entered it, not as you meant it!
This goes for your C source code as well. A simple typographical error in your program can cause the C compiler to choke, gag, and collapse. Fortunately, although the compiler isn'tsmart enough to correct your errors (and you'll make errors--everyone does!), it is smart enough to recognize them as errors and report them to you. (You saw in the preceding chapter how the compiler reports error messages and how you interpret them.)

A Review of the Parts of a Program

Now that all the parts of a program have been described, you should be able to look at any program and find some similarities. Look at Listing 2.2 and see whether you can identify the different parts.
Listing 2.2. LIST_IT.C.

```
1: /* LIST_IT.C--This program displays a listing with line numbers! */
2: #include <stdio.h>
3: #include <stdlib.h>
4:
5: void display_usage(void);
6: int line;
7:
8: main( int argc, char *argv[] )
9: {
10: char buffer[256];
11: FILE *fp;
12:
13: if( argc < 2 )
14: {
15: display_usage();
16: exit(1);
17: }
18:
19: if (( fp = fopen( argv[1], "r" )) == NULL )
{
fprintf( stderr, "Error opening file, %s!", argv[1] );
exit(1);
}

line = 1;

while( fgets( buffer, 256, fp ) != NULL )
fprintf( stdout, "%4d:\t%s", line++, buffer );

fclose(fp);
return 0;
}

void display_usage(void)
{
fprintf(stderr, "\nProper Usage is: " );
fprintf(stderr, "\n\nLIST_IT filename.ext\n" );
}
```

```
C:\>list_it list_it.c
1: /* LIST_IT.C - This program displays a listing with line numbers! */
2: #include <stdio.h>
3: #include <stdlib.h>
4:
5: void display_usage(void);
6: int line;
7:
8: main( int argc, char *argv[] )
9: {
10: char buffer[256];
11: FILE *fp;
12:
13: if( argc < 2 )
14: {

15: display_usage();
16: exit(1);
17: }
18:
19: if (( fp = fopen( argv[1], "r" )) == NULL )
20: {
21: fprintf( stderr, "Error opening file, %s!", argv[1] );
22: exit(1);
23: }
24:
25: line = 1;
26:
27: while( fgets( buffer, 256, fp ) != NULL )
28: fprintf( stdout, "%4d:\t%s", line++, buffer );
29:
30: fclose(fp);
31: return 0;
32: }
33:
```

```
34: void display_usage(void)
35: {
36: fprintf(stderr, "\nProper Usage is: " );
37: fprintf(stderr, "\n\nLIST_IT filename.ext\n" );
38: }
```

ANALYSIS: LIST_IT.C is similar to PRINT_IT.C, which you entered in exercise 7 of Day 1. Listing 2.2 displays saved C program listings on-screen instead of printing them on the Printer. Looking at this listing, you can summarize where the different parts are. The required main() function is in lines 8 through 32. Lines 2 and 3 have #include directives. Lines 6, 10, and 11 have variable definitions. A function prototype, void display_usage(void), is in line 5. This program has many statements (lines 13, 15, 16, 19, 21, 22, 25, 27, 28, 30, 31, 36, and 37). A function definition for display_usage() fills lines 34 through 38. Braces enclose blocks throughout the program. Finally, only line 1 has a comment. In most programs, you should probably include more than one comment line.

LIST_IT.C calls many functions. It calls only one user-defined function, display_usage(). The library functions that it uses are exit() in lines 16 and 22; fopen() in line 19; fprintf() in lines 21, 28, 36, and 37; fgets() in line 27; and fclose() in line 30. These library functions are covered in more detail throughout this book.

Summary
This chapter was short, but it's important, because it introduced you to the major components of a C program. You learned that the single required part of every C program is the main() function. You also learned that the program's real work is done by program statements that instruct the computer to perform your desired actions. This chapter also introduced you to variables and variable definitions, and it showed you how to use comments in your source code.
In addition to the main() function, a C program can use two types of subsidiary functions: library functions, supplied as part of the compiler package, and user-defined functions,created by the programmer.

Module 2: Types, Operators & Expressions

Objective: The participants will understand how to

- How size and capacity is used to store data.
- Write expressions and their execution.
- Variable naming convention
- How to frame conditions and understand their execution flow.

Chapter 1: Data Types

Definition and usage of variables
int, char, float, double types
Their optimal usage and capacity-size of Different notations for integer's values: octal, hexadecimal, decimal
Data types modifiers:long, short, unsigned
printf and display of different data types
Escape sequences

Chapter 2: Arithmatic Operators

Arithmetic operators +, -, *, /, %
Increment and decrement operators
Operator precedence and associatively
Assignment operators
Short hand assignment operators +-, -=, *=, /=, %=
Automatic promotions of data types and explicit type cast

Chapter 3: Non-Arithmatic Operators

Relational operators ==, !=, >, >=, <, <=
logical operators - &&, ||, !
(Ternary) Conditional operator

Data Types

Computer programs usually work with different types of data and need a way to store the values being used. These values can be numbers or characters. C has two ways of storing number values--variables and constants--with many options for each. A variable is a data storage location that has a value that can change during program execution. In contrast, a constant has a fixed value that can't change. Today you will learn
How to create variable names in C
The use of different types of numeric variables
The differences and similarities between character and numeric values
How to declare and initialize numeric variables
C's two types of numeric constants
Before you get to variables, however, you need to know a little about the operation of your computer's memory.

Computer Memory

A computer uses random-access memory (RAM) to store information while it's operating. RAM is located in integrated circuits, or chips, inside your computer. RAM is volatile, which means that it is erased and replaced with new information as often as needed. Being volatile also means that RAM "remembers" only while the computer is turned on and loses its information when you turn the computer off.

Each computer has a certain amount of RAM installed. The amount of RAM in a system is usually specified in kilobytes (KB) or megabytes (MB), such as 512KB, 640KB, 2MB, 4MB, or 8MB. One kilobyte of memory consists of 1,024 bytes. Thus, a system with 640KB of memory actually has 640 * 1,024, or 65,536, bytes of RAM. One megabyte is 1,024 kilobytes. A machine with 4MB of RAM would have 4,096KB or 4,194,304 bytes of RAM.

The byte is the fundamental unit of computer data storage. Day 20, "Working with Memory," has more information about bytes. For now, to get an idea of how many bytes it takes to store certain kinds of data, refer to Table 3.1.

Table 3.1. Memory space required to store data.

Data	**Bytes Required**
The letter x	1
The number 500	2
The number 241.105	4
The phrase Teach Yourself C	17
One typewritten page	Approximately 3,000

The RAM in your computer is organized sequentially, one byte following another. Each byte of memory has a unique address by which it is identified--an address that also distinguishes it from all other bytes in memory. Addresses are assigned to memory locations in order, starting at zero and increasing to the system limit. For now, you don't need to worry about addresses; it's all handled automatically by the C compiler.

What is your computer's RAM used for? It has several uses, but only data storage need concern you as a programmer. Data is the information with which your C program works. Whether your program is maintaining an address list, monitoring the stock market, keeping a household budget, or tracking the price of hog bellies, the information (names, stock prices, expense amounts, or hog futures) is kept in your computer's RAM while the program is running.

Now that you understand a little about the nuts and bolts of memory storage, you can get back to C programming and how C uses memory to store information.

Variables

A variable is a named data storage location in your computer's memory. By using a variable's name in your program, you are, in effect, referring to the data stored there.

Variable Names

To use variables in your C programs, you must know how to create variable names. In C, variable names must adhere to the following rules:
The name can contain letters, digits, and the underscore character (_).
The first character of the name must be a letter. The underscore is also a legal first character, but its use is not recommended.
Case matters (that is, upper- and lowercase letters). Thus, the names count and Count refer to two different variables.
C keywords can't be used as variable names. A keyword is a word that is part of the C language. (A complete list of 33 C keywords can be found in Appendix B, "Reserved Words.")

The following list contains some examples of legal and illegal C variable names:

Variable Name	**Legality**
Percent Legal	Legal
y2x5__fg7h	Legal
annual_profit	Legal but not advised
_1990_tax	
savings#account	Illegal: Contains the illegal character #
double	Illegal: Is a C keyword
9winter	Illegal: First character is a digit

Because C is case-sensitive, the name percent, PERCENT, and Percent would be considered three different variables. C programmers commonly use only lowercase letters in variable names, although this isn't required. Using all-uppercase letters is usually reserved for the names of constants (which are covered later in this chapter).

For many compilers, a C variable name can be up to 31 characters long. (It can actually be longer than that, but the compiler looks at only the first 31 characters of the name.) With this flexibility, you can create variable names that reflect the data being stored. For example, a program that calculates loan payments could store the value of the prime interest rate in a variable named interest_rate. The variable name helps make its usage clear. You could also have created a variable named x or even johnny_carson; it doesn't matter to the C compiler.
The use of the variable, however, wouldn't be nearly as clear to someone else looking at the source code. Although it might take a little more time to type descriptive variable names, the improvements in program clarity make it worthwhile.
Many naming conventions are used for variable names created from multiple words. You've seen one style: interest_rate. Using an underscore to separate words in a variable name makes it easy to interpret. The second style is called camel notation. Instead of using spaces, the first letter of each word is capitalized. Instead of interest_rate, the variable would be named InterestRate. Camel notation is gaining popularity, because it's easier to type a capital letter than an underscore. We use the underscore in this book because it's easier for most people to read. You should decide which style you want to adopt.
DO use variable names that are descriptive.
DO adopt and stick with a style for naming your variables.
DON'T start your variable names with an underscore unnecessarily.
DON'T name your variables with al capital letters unnecessarily.

Numeric Variable Types

C provides sev eral different types of numeric variables. You need different types of variables because different numeric values have varying memory storage requirements and differ in the ease with which certain mathematical operations can be performed on them. Small integers (for example, 1, 199, and -8) require less memory to store, and your computer can perform mathematical operations (addition, multiplication, and so on) with such numbers very quickly. In contrast, large integers and floating-point values (123,000,000 or 0.000000871256, for example) require more storage space and more time for mathematical operations. By using the appropriate variable types, you ensure that your program runs as efficiently as possible.
C's numeric variables fall into the following two main categories:
Integer variables hold values that have no fractional part (that is, whole numbers only). Integer variables come in two flavors: signed integer variables can hold positive or negative values, whereas unsigned integer variables can hold only positive values (and 0).

Floating-point variables hold values that have a fractional part (that is, real numbers).

Within each of these categories are two or more specific variable types. These are summarized in Table 3.2, which also shows the amount of memory, in bytes, required to hold a single variable of each type when you use a microcomputer with 16-bit architecture.

Table 3.2. C's numeric data types.

Variable Type	Keyword	Bytes Required	Range
Character	char	1	-128 to 127
Integer	int	2	-32768 to 32767
Short integer	short	2	-32768 to 32767
Long integer	long	4	-2,147,483,648 to 2,147,438,647
Unsigned character	unsigned char	1	0 to 255
Unsigned integer	unsigned int	2	0 to 65535
Unsigned short integer	unsigned short	2	0 to 65535
Unsigned long integer	unsigned long	4	0 to 4,294,967,295
Single-precision float			
floating-point			
Double-precision double		1	2E-38
floating-point		3	4E381
1Approximate range; precision = 7 digits. 4		2	2E-308
2Approximate range; precision = 19 digits.8		1	8E3082

Approximate range means the highest and lowest values a given variable can hold. (Space limitations prohibit listing exact ranges for the values of these variables.) Precision means the accuracy with which the variable is stored. (For example, if you evaluate 1/3, the answer is 0.33333... with 3s going to infinity. A variable with a precision of 7 stores seven 3s.)

Looking at Table 3.2, you might notice that the variable type int and short are identical. Why are two different types necessary? The int and short variable types are indeed identical on 16- bit IBM PC-compatible systems, but they might be different on other types of hardware. On a VAX system, a short and an int aren't the same size. Instead, a short is 2 bytes, whereas an int is 4. Remember that C is a flexible, portable language, so it provides different keywords for the two types. If you're working on a PC, you can use int and short interchangeably. No special keyword is needed to make an integer variable signed; integer variables are signed by default. You can, however, include the signed keyword if you

wish. The keywords shown in Table 3.2 are used in variable declarations, which are discussed in the next section.
Listing 3.1 will help you determine the size of variables on your particular computer. Don't be surprised if your output doesn't match the output presented after the listing.

Listing 3.1. A program that displays the size of variable types.

```
1: /* SIZEOF.C--Program to tell the size of the C variable */
2: /* type in bytes */
3:
4: #include <stdio.h>
5:
6: main()
7: {
8:
9: printf( "\nA char is %d bytes", sizeof( char ));
10: printf( "\nAn int is %d bytes", sizeof( int ));
11: printf( "\nA short is %d bytes", sizeof( short ));
12: printf( "\nA long is %d bytes", sizeof( long ));
13: printf( "\nAn unsigned char is %d bytes", sizeof( unsigned char ));
14: printf( "\nAn unsigned int is %d bytes", sizeof( unsigned int ));
15: printf( "\nAn unsigned short is %d bytes", sizeof( unsigned short ));
16: printf( "\nAn unsigned long is %d bytes", sizeof( unsigned long ));
17: printf( "\nA float is %d bytes", sizeof( float ));
18: printf( "\nA double is %d bytes\n", sizeof( double ));
19:
20: return 0;
21: }
```

```
A char is 1 bytes
An int is 2 bytes
A short is 2 bytes
A long is 4 bytes
An unsigned char is 1 bytes
An unsigned int is 2 bytes
An unsigned short is 2 bytes
An unsigned long is 4 bytes
A float is 4 bytes
A double is 8 bytes
```

ANALYSIS: As the preceding output shows, Listing 3.1 tells you exactly how many bytes each variable type on your computer takes. If you're using a 16-bit PC, your numbers should match those in Table 3.2.
Don't worry about trying to understand all the individual components of the program. Although some items are new, such as sizeof(), others should look familiar. Lines 1 and 2 are comments about the name of the program and a brief description. Line 4 includes the standard input/output header file to help print the information on-screen. This is a simple program, in that it contains only a single function, main() (lines 7 through 21). Lines 9 through 18 are the bulk of the program. Each of these lines prints a textual description with the size of each of the variable types, which is done using the sizeof operator. Day 19, "Exploring the C Function Library," covers the sizeof operator in detail. Line 20 of the program returns the value 0 to the operating system before ending the program. Although size of the data types can vary depending on your computer platform, C does make some guarantees, thanks to the ANSI Standard. There are five things you can count on:

- The size of a char is one byte.
- The size of a short is less than or equal to the size of an int.
- The size of an int is less than or equal to the size of a long.
- The size of an unsigned is equal to the size of an int.
- The size of a float is less than or equal to the size of a double.

Variable Declarations

Before you can use a variable in a C program, it must be declared. A variable declaration tells the compiler the name and types of a variable and optionally initializes the variable to a specific value. If your program attempts to use a variable that hasn't been declared, the compiler generates an error message. A variable declaration has the following form:

```
typename varname;
```

typename specifies the variable type and must be one of the keywords listed in Table 3.2.
varname is the variable name, which must follow the rules mentioned earlier. You can declare multiple variables of the same type on one line by separating the variable names with commas:

```
int count, number, start;    /* three integer variables */
float percent, total;        /* two float variables */
```

On Day 12, "Understanding Variable Scope," you'll learn that the location of variable declarations in the source code is important, because it affects the ways in which your program can use the variables. For now, you can place all the variable declarations together just before the start of the main() function.

The typedef Keyword

The typedef keyword is used to create a new name for an existing data type. In effect, typedef creates a synonym. For example, the statement typedef int integer; creates integer as a synonym for int. You then can use integer to define variables of type int, as in this example:

```
integer count;
```

❑that typedef doesn't create a new data type; it only lets you use a different name for a predefined data type. The most common use of typedef concerns aggregate data types, as explained on Day 11, "Structures." An aggregate data type consists of a combination of data types presented in this chapter.

Initializing Numeric Variables

When you declare a variable, you instruct the compiler to set aside storage space for the variable. However, the value stored in that space--the value of the variable--isn't defined. It might be zero, or it might be some random "garbage" value. Before using a variable, you should always initialize it to a known value. You can do this independently of the variable declaration by using an assignment statement, as in this example:

```
int count;   /* Set aside storage space for count */
count = 0;   /* Store 0 in count */
```

❑ that this statement uses the equal sign (=), which is C's assignment operator and is discussed further on Day 4, "Statements, Expressions, and Operators." For now, you need to be aware that the equal sign in programming is not the same as the equal sign in algebra. If you write x = 12 in an algebraic statement, you are stating a fact: "x equals 12." In C, however, it means something quite different: "Assign the value 12 to the variable named x."

You can also initialize a variable when it's declared. To do so, follow the variable name in the declaration statement with an equal sign and the desired initial value:

```
int count = 0;
double percent = 0.01, taxrate = 28.5;
```

Be careful not to initialize a variable with a value outside the allowed range. Here are two examples of out-of-range initializations:

```
int weight = 100000;
unsigned int value = -2500;
```

The C compiler doesn't catch such errors. Your program might compile and link, but you might get unexpected results when the program is run.

DO understand the number of bytes that variable types take for your computer.

DO use typedef to make your programs more readable.
DO initialize variables when you declare them whenever possible.
DON'T use a variable that hasn't been initialized. Results can be unpredictable.
DON'T use a float or double variable if you're only storing integers. Although they will work, using them is inefficient.
DON'T try to put numbers into variable types that are too small to hold them.
DON'T put negative numbers into variables with an unsigned type.

Summary
This chapter explored numeric variables, which are used by a C program to store data during program execution. You've seen that there are two broad classes of numeric variables, integer and floating-point. Within each class are specific variable types. Which variable type--int, long, float, or double--you use for a specific application depends on the nature of the data to be stored in the variable. You've also seen that in a C program, you must declare a variable before it can be used. A variable declaration informs the compiler of the name and type of a variable.
This chapter also covered C's two constant types, literal and symbolic. Unlike variables, the value of a constant can't change during program execution. You type literal constants into your source code whenever the value is needed. Symbolic constants are assigned a name that is used wherever the constant value is needed. Symbolic constants can be created with the #define directive or with the const keyword.

Statements,Expressions and Operators

C programs consist of statements, and most statements are composed of expressions and operators. You need to understand these three topics in order to be able to write C programs.
Today you will learn
What a statement is
What an expression is
C's mathematical, relational, and logical operators
What operator precedence is
The if statement

Statements

A statement is a complete direction instructing the computer to carry out some task. In C, statements are usually written one per line, although some statements span multiple lines. C statements always end with a semicolon (except for preprocessor directives such as #define and #include, which are discussed later, "Advanced Compiler Use"). You've already been introduced to some of C's statement types. For example:
```
x = 2 + 3;
```

is an assignment statement. It instructs the computer to add 2 and 3 and to assign the result to the variable x. Other types of statements will be introduced as needed throughout this book.

Escape Sequence

The term white space refers to spaces, tabs, and blank lines in your source code. The C compiler isn't sensitive to white space. When the compiler reads a statement in your source code, it looks for the characters in the statement and for the terminating semicolon, but it ignores white space. Thus, the statement
```
x=2+3;
```
is equivalent to this statement:
```
x = 2 + 3;
```
It is also equivalent to this:
```
x       =
2
```

```
+
3;
```

This gives you a great deal of flexibility in formatting your source code. You shouldn't use formatting like the previous example, however. Statements should be entered one per line with a standardized scheme for spacing around variables and operators. If you follow the formatting conventions used in this book, you should be in good shape. As you become more experienced, you might discover that you prefer slight variations. The point is to keep your source code readable.

However, the rule that C doesn't care about white space has one exception: Within literal string constants, tabs and spaces aren't ignored; they are considered part of the string. A string is a series of characters. Literal string constants are strings that are enclosed within quotes and interpreted literally by the compiler, space for space. Although it's extremely bad form, the following is legal:

```
printf(
"Hello, world!"
);
```

This, however, is not legal:

```
printf("Hello,
world!");
```

To break a literal string constant line, you must use the backslash character (\) just before the break. Thus, the following is legal:

```
printf("Hello,world!");
```

Sequence	**Meaning**
\a	Bell (alert)
\b	Backspace
\n	Newline
\t	Horizontal tab
\\	Backslash
\?	Question mark
\'	Single quotation

Null Statements

If you place a semicolon by itself on a line, you create a null statement--a statement that doesn't perform any action. This is perfectly legal in C. Later in this book, you will learn how the null statement can be useful.

Compound Statements

A compound statement, also called a block, is a group of two or more C statements enclosed in braces. Here's an example of a block:

```
{
   printf("Hello, ");
   printf("world!");
}
```

In C, a block can be used anywhere a single statement can be used. Many examples of this appear throughout this book.

❑ that the enclosing braces can be positioned in different ways. The following is equivalent to the preceding example:

```
{printf("Hello, ");
printf("world!");}
```

It's a good idea to place braces on their own lines, making the beginning and end of blocks clearly visible. Placing braces on their own lines also makes it easier to see whether you've left one out.

DO stay consistent with how you use white space in statements.

DO put block braces on their own lines. This makes the code easier to read.

DO line up block braces so that it's easy to find the beginning and end of a block.
DON'T spread a single statement across multiple lines if there's no need to do so. Limit statements to one line if possible.

Expressions

In C, an expression is anything that evaluates to a numeric value. C expressions come in all levels of complexity.

Simple Expressions

The simplest C expression consists of a single item: a simple variable, literal constant, or symbolic constant. Here are four expressions:

Expression	Description
PI	A symbolic constant (defined in the program)
20	A literal constant
Rate	A variable
-1.25	Another literal constant

A literal constant evaluates to its own value. A symbolic constant evaluates to the value it was given when you created it using the #define directive. A variable evaluates to the current value assigned to it by the program.

Operators

An operator is a symbol that instructs C to perform some operation, or action, on one or more operands. An operand is something that an operator acts on. In C, all operands are expressions. C operators fall into several categories:

The assignment operator
Mathematical operators
Relational operators
Logical operators

The Assignment Operator

The assignment operator is the equal sign (=). Its use in programming is somewhat different from its use in regular math. If you write
x = y;
in a C program, it doesn't mean "x is equal to y." Instead, it means "assign the value of y to x."
In a C assignment statement, the right side can be any expression, and the left side must be a variable name. Thus, the form is as follows:
variable = expression;
When executed, expression is evaluated, and the resulting value is assigned to variable.

Mathematical Operators

C's mathematical operators perform mathematical operations such as addition and subtraction. C has two unary mathematical operators and five binary mathematical operators.

Unary Mathematical Operators
The unary mathematical operators are so named because they take a single operand. C has two unary mathematical operators, listed in Table 4.1.
Table 4.1. C's unary mathematical operators.

Operator	Symbol	Action	Examples
Increment	++	Increments the operand by one	++x, x++
Decrement	- -	Decrements the operand by one	--x, x--

The increment and decrement operators can be used only with variables, not with constants. The operation performed is to add one to or subtract one from the operand. In other words, the statements
++x;
--y;
are the equivalent of these statements:
x = x + 1;
y = y - 1;

You should □ from Table 4.1 that either unary operator can be placed before its operand (prefix mode) or after its operand (postfix mode). These two modes are not equivalent. They differ in terms of when the increment or decrement is performed:

- When used in prefix mode, the increment and decrement operators modify their operand before it's used.
- When used in postfix mode, the increment and decrement operators modify their operand after it's used.

An example should make this clearer. Look at these two statements:
x = 10;
y = x++;
After these statements are executed, x has the value 11, and y has the value 10. The value of x was assigned to y, and then x was incremented. In contrast, the following statements result in both y and x having the value 11. x is incremented, and then its value is assigned to y.
x = 10;

y = ++x;
Remember that = is the assignment operator, not a statement of equality. As an analogy, think of = as the "photocopy" operator. The statement y = x means to copy x into y.
Subsequent changes to x, after the copy has been made, have no effect on y.
Listing 4.1 illustrates the difference between prefix mode and postfix mode.

Listing 4.1. UNARY.C: Demonstrates prefix and postfix modes.

```
/* Demonstrates unary operator prefix and postfix modes */

#include <stdio.h>

int a, b;

main()
{
   /* Set a and b both equal to 5 */

   a = b = 5;

   /* Print them, decrementing each time. */
   /* Use prefix mode for b, postfix mode for a */

   printf("\n%d  %d", a--, --b);
   printf("\n%d  %d", a--, --b);
   printf("\n%d  %d", a--, --b);
19:     printf("\n%d   %d", a--, --b);
20:     printf("\n%d   %d\n", a--, --b);
21:
22:     return 0;
23: }
```

```
5   4
4   3
3   2
2   1
1   0
```

ANALYSIS: This program declares two variables, a and b, in line 5. In line 11, the variables are set to the value of 5. With the execution of each printf() statement (lines 16 through 20), both a and b are decremented by 1. After a is printed, it is decremented, whereas b is decremented before it is printed.

Binary Mathematical Operators

C's binary operators take two operands. The binary operators, which include the common mathematical operations found on a calculator, are listed in Table 4.2.

Table 4.2. C's binary mathematical operators.

Operator	Symbol	Action	Example
Addition	+	Adds two operands	x + y
Subtraction	-	Subtracts the second operand from the first operand	x - y
Multiplication	*	Multiplies two operands	x * y
Division	/	Divides the first operand by the second operand	x / y
Modulus	%	Gives the remainder when the first operand is divided by the second operand	x % y

The first four operators listed in Table 4.2 should be familiar to you, and you should have little trouble using them. The fifth operator, modulus, might be new. Modulus returns the remainder when the first operand is divided by the second operand. For example, 11 modulus 4 equals 3 (that is, 4 goes into 11 two times with 3 left over). Here are some more examples:

100 modulus 9 equals 1
10 modulus 5 equals 0
40 modulus 6 equals 4
Listing 4.2 illustrates how you can use the modulus operator to convert a large number of seconds into hours,
minutes, and seconds.

Listing 4.2. SECONDS.C: Demonstrates the modulus operator.

```
1: /* Illustrates the modulus operator. */
2: /* Inputs a number of seconds, and converts to hours, */
3: /* minutes, and seconds. */
4:
5: #include <stdio.h>
6:
7: /* Define constants */
8:
9: #define SECS_PER_MIN 60
10: #define SECS_PER_HOUR 3600
11:
12: unsigned seconds, minutes, hours, secs_left, mins_left;
13:
14: main()
15: {
16: /* Input the number of seconds */
17:
18: printf("Enter number of seconds (< 65000): ");
19: scanf("%d", &seconds);
20:
21: hours = seconds / SECS_PER_HOUR;
22: minutes = seconds / SECS_PER_MIN;
23: mins_left = minutes % SECS_PER_MIN;
24: secs_left = seconds % SECS_PER_MIN;
25:
26: printf("%u seconds is equal to ", seconds);
27: printf("%u h, %u m, and %u s\n", hours, mins_left, secs_left);
28:
29: return 0;
30: }
```

```
Enter number of seconds (< 65000): 60
60 seconds is equal to 0 h, 1 m, and 0 s
Enter number of seconds (< 65000): 10000
10000 seconds is equal to 2 h, 46 m, and 40 s
```

ANALYSIS: SECONDS.C follows the same format that all the previous programs have followed. Lines 1 through 3 provide some comments to state what the program does. Line 4 is white space to make the program more readable. Just like the white space in statements and expressions, blank lines are ignored by the compiler. Line 5 includes the necessary header file for this program. Lines 9 and 10 define two constants, SECS_PER_MIN and SECS_PER_HOUR, that are used to make the statements in the program easier to read. Line 12 declares all the variables that will be used. Some people choose to declare each variable on a separate line rather than all on one. As with many elements of C, this is a matter of style. Either method is correct. Line 14 is the main() function, which contains the bulk of the program. To convert seconds to hours and minutes, the program must first get the values it needs to work with. To do this, line 18 uses the printf() function to display a statement on-screen, followed by line 19, which uses the scanf() function to get the number that the user entered. The scanf() statement then stores the number of seconds to be converted into the variable seconds. The printf() and scanf() functions are covered in more detail on Day 7, "Fundamentals of Input and Output." Line 21 contains an expression to determine the number of hours by dividing the number of seconds by the constant SECS_PER_HOUR. Because hours is an integer variable, the remainder value is ignored. Line 22 uses the same logic to determine the total number of minutes for the seconds entered. Because the total number of minutes figured in line 22 also contains minutes for the hours, line 23 uses the modulus operator to divide the hours and keep the remaining minutes. Line 24 carries out a similar calculation for determining the number of seconds that are left. Lines 26 and 27 are similar to what you have seen before. They take the values that have been calculated in the expressions and display them. Line 29 finishes the program by returning 0 to the operating system before exiting. Operator Precedence and Parentheses In an expression that contains more than one operator, what is the order in which operations are performed? The importance of this question is il ustrated by the following assignment statement:

```
x = 4 + 5 * 3;
```

Performing the addition first results in the following, and x is assigned the value 27:

```
x = 9 * 3;
```

In contrast, if the multiplication is performed first, you have the following, and x is assigned the value 19:

```
x = 4 + 15;
```

Clearly, some rules are needed about the order in which operations are performed. This order, called operator precedence, is strictly spelled out in C. Each operator has a specific precedence. When an expression is evaluated, operators with higher precedence are performed first. Table 4.3 lists the precedence of C's mathematical operators. Number 1 is the highest precedence and thus is evaluated first.

Table 4.3. The precedence of C's mathematical operators.

Operators	**Relative Precedence**
++ --	1
* / %	2
+ -	3

Looking at Table 4.3, you can see that in any C expression, operations are performed in the following order:

Unary increment and decrement

Multiplication, division, and modulus

Addition and subtraction

If an expression contains more than one operator with the same precedence level, the operators are performed in left-to-right order as they appear in the expression. For example, in the following expression, the % and * have the same precedence level, but the % is the leftmost operator, so it is performed first:

```
12 % 5 * 2
```

The expression evaluates to 4 (12 % 5 evaluates to 2; 2 times 2 is 4).

Returning to the previous example, you see that the statement x = 4 + 5 * 3; assigns the value 19 to x because the multiplication is performed before the addition.What if the order of precedence doesn't evaluate your expression as needed? Using the previous example, what if you wanted to add 4 to 5 and then multiply the sum by 3? C uses parentheses to modify the evaluation order. A subexpression enclosed in parentheses is evaluated first, without regard to operator precedence. Thus, you could write

```
x = (4 + 5) * 3;
```

The expression 4 + 5 inside parentheses is evaluated first, so the value assigned to x is 27. You can use multiple and nested parentheses in an expression. When parentheses are nested, evaluation proceeds from the innermost expression outward. Look at the following complex expression:

```
x = 25 - (2 * (10 + (8 / 2)));
```

The evaluation of this expression proceeds as follows:

1. The innermost expression, 8 / 2, is evaluated first, yielding the value 4:
25 - (2 * (10 + 4))

2. Moving outward, the next expression, 10 + 4, is evaluated, yielding the value 14:
25 - (2 * 14)

3. The last, or outermost, expression, 2 * 14, is evaluated, yielding the value 28:
25 – 28

4. The final expression, 25 - 28, is evaluated, assigning the value -3 to the variable x:
x = -3

You might want to use parentheses in some expressions for the sake of clarity, even when they aren't needed for modifying operator precedence. Parentheses must always be in pairs, or the compiler generates an error message.Order of Subexpression Evaluation As was mentioned in the previous section, if C expressions contain more than one operator with the same precedence level, they are evaluated left to right. For example, in the expression

w * x / y * z
w is multiplied by x, the result of the multiplication is then divided by y, and the result of the division is then multiplied by z.
Across precedence levels, however, there is no guarantee of left-to-right order. Look at this expression:
w * x / y + z / y
Because of precedence, the multiplication and division are performed before the addition. However, C doesn't specify whether the subexpression w * x / y is to be evaluated before or after z / y. It might not be clear to you why this matters. Look at another example:
w * x / ++y + z / y
If the left subexpression is evaluated first, y is incremented when the second expression is evaluated. If the right expression is evaluated first, y isn't incremented, and the result is different. Therefore, you should avoid this sort of indeterminate expression in your programming. Near the end of this chapter, the section "Operator Precedence Revisited" lists the precedence of all of C's operators.
DO use parentheses to make the order of expression evaluation clear.
DON'T overload an expression. It is often more clear to break an expression
into two or more statements. This is especially true when you're using the unary operators (--) or (++).

Relational Operators

C's relational operators are used to compare expressions, asking questions such as, "Is x greater than 100?" or "Is y equal to 0?" An expression containing a relational operator evaluates to either true (1) or false (0). C's six relational operators are listed in Table 4.4.

Table 4.5 shows some examples of how relational operators might be used. These examples use literal constants, but the same principles hold with variables.
❑: "True" is considered the same as "yes," which is also considered the same as 1. "False" is considered the same as "no," which is considered the same as 0.

Table 4.4. C's relational operators.

Operator	Symbol	Question Asked	Example
Equal			
Greater than			
Less than			

==
>
<

Is operand 1 equal to operand 2?
Is operand 1 greater than operand 2?
Is operand 1 less than operand 2?

x == y
x > y
x < y

Greater than or equal to >=

Is operand 1 greater than or equal to operand 2? x >= y

Less than or equal to

<=

Is operand 1 less than or equal to operand 2? x <= y

Not equal

!=

Is operand 1 not equal to operand 2?

x != y

Table 4.5. Relational operators in use.

Expression	How It Reads	What It Evaluates To
5 == 1	Is 5 equal to 1?	0 (false)
5 > 1	Is 5 greater than 1?	1 (true)
5 != 1	Is 5 not equal to 1?	1 (true)
(5 + 10) == (3 * 5)	Is (5 + 10) equal to (3 * 5)?	1 (true)

DO learn how C interprets true and false. When working with relational operators, true is equal to 1, and false is equal to 0.
DON'T confuse ==, the relational operator, with =, the assignment operator.
This is one of the most common errors that C programmers make.

The Precedence of Relational Operators

Like the mathematical operators discussed earlier in this chapter, the relational operators each have a precedence that determines the order in which they are performed in a multiple-operator expression. Similarly, you can use parentheses to modify precedence in expressions that use relational operators. The section "Operator Precedence Revisited" near the end of this chapter lists the precedence of all of C's operators.

Logical Operators

C's logical operators let you combine two or more relational expressions into a single expression that evaluates to either true or false. Table 4.7 lists C's three logical operators.

Table 4.7. C's logical operators.

Operator	Symbol	Example
AND	exp1 && exp2	
OR	exp1 || exp2	
NOT	exp1	

The way these logical operators work is explained in Table 4.8.

Table 4.8. C's logical operators in use.

Expression	What It Evaluates To
(exp1 && exp2)	True (1) only if both exp1 and exp2 are true; false (0) otherwise
(exp1 || exp2)	True (1) if either exp1 or exp2 is true; false (0) only if both are false
(!exp1)	False (0) if exp1 is true; true (1) if exp1 is false

You can see that expressions that use the logical operators evaluate to either true or false, depending on the true/false value of their operand(s). Table 4.9 shows some actual code examples.

Table 4.6. The order of precedence of C's relational operators.

Operators	Relative Precedence
< <= > >=	1
!= ==	2

Thus, if you write

```
x == y > z
```

it is the same as

```
x == (y > z)
```

because C first evaluates the expression y > z, resulting in a value of 0 or 1. Next, C determines whether x is equal to the 1 or 0 obtained in the first step. You will rarely, if ever, use this sort of construction, but you should know about it.

Module 3: Control Flow

Module Objective: the participants will understand

- How to direct the sequence of execution using statements
- Have an understanding of the iterative process.

Chapter 1: Decision Constructs
If Statement
If else statement
If else ladder
switch statement
break

Chapter 2: Looping Constructs
while statement
while loop
do while loop
for loop

Expression

(5 == 5) && (6 != 2)
(5 > 1) || (6 < 1)
(2 == 1) && (5 == 5)
!(5 == 4)

What It Evaluates To

True (1), because both operands are true
True (1), because one operand is true
False (0), because one operand is false
True (1), because the operand is false

You can create expressions that use multiple logical operators. For example, to ask the question "Is x equal to 2, 3, or 4?" you would write
(x == 2) || (x == 3) || (x == 4)
The logical operators often provide more than one way to ask a question. If x is an integer variable, the preceding question also could be written in either of the following ways:
(x > 1) && (x < 5)
(x >= 2) && (x <= 4)

More on True/False Values

You've seen that C's relational expressions evaluate to 0 to represent false and to 1 to represent true. It's important to be aware, however, that any numeric value is interpreted as either true or false when it is used in a C expression or statement that is expecting a logical value (that is, a true or false value). The rules are as follows:

- A value of zero represents false.
- Any nonzero value represents true.

This is illustrated by the following example, in which the value of x is printed:

```
x = 125;
if (x)
   printf("%d", x);
```

Because x has a nonzero value, the if statement interprets the expression (x) as true. You can further generalize this because, for any C expression, writing (expression) is equivalent to writing

```
(expression != 0)
```

Both evaluate to true if expression is nonzero and to false if expression is 0. Using the not (!) operator, you can also write (!expression) which is equivalent to (expression == 0)

The Precedence of Operators

As you might have guessed, C's logical operators also have a precedence order, both among themselves and in relation to other operators. The ! operator has a precedence equal to the unary mathematical operators ++ and --. Thus, ! has a higher precedence than all the relational operators and all the binary mathematical operators.
In contrast, the && and || operators have much lower precedence, lower than all the mathematical and relational operators, although && has a higher precedence than ||. As with all of C's operators, parentheses can be used to modify the evaluation order when using the logical operators. Consider the following example:
You want to write a logical expression that makes three individual comparisons:
1. Is a less than b?
2. Is a less than c?
3. Is c less than d?

You want the entire logical expression to evaluate to true if condition 3 is true and if either condition 1 or condition 2 is true. You might write

```
a < b || a < c && c < d
```

However, this won't do what you intended. Because the && operator has higher precedence than ||, the expression is equivalent to

```
a < b || (a < c && c < d)
```

and evaluates to true if (a < b) is true, whether or not the relationships (a < c) and (c < d) are true. You need to write

```
(a < b || a < c) && c < d
```

which forces the || to be evaluated before the &&. This is shown in Listing 4.6, which evaluates the expression written both ways. The variables are set so that, if written correctly, the expression should evaluate to false (0).

Listing 4.6. Logical operator precedence.

```
#include <stdio.h>

/* Initialize variables. Note that c is not less than d, */
/* which is one of the conditions to test for. */
/* Therefore, the entire expression should evaluate as false.*/

int a = 5, b = 6, c = 5, d = 1;
int x;
```

```
9:
10: main()
11: {
12: /* Evaluate the expression without parentheses */
 13:
14: x = a < b || a < c && c < d;
15: printf("\nWithout parentheses the expression evaluates as %d", x);
16:
17: /* Evaluate the expression with parentheses */
18:
19: x = (a < b || a < c) && c < d;
20: printf("\nWith parentheses the expression evaluates as %d\n", x);
21: return 0;
22: }
```

```
Without parentheses the expression evaluates as 1
With parentheses the expression evaluates as 0
```

ANALYSIS: Enter and run this listing. Note that the two values printed for the expression are different. This program initializes four variables, in line 7, with values to be used in the comparisons. Line 8 declares x to be used to store and print the results. Lines 14 and 19 use the logical operators. Line 14 doesn't use parentheses, so the results are determined by operator precedence. In this case, the results aren't what you wanted. Line 19 uses parentheses to change the order in which the expressions are evaluated.Compound Assignment Operators C's compound assignment operators provide a shorthand method for combining a binary mathematical operation with an assignment operation. For example, say you want to increase the value of x by 5, or, in other words, add 5 to x and assign the result to x. You could write

```
x = x + 5;
```

Using a compound assignment operator, which you can think of as a shorthand method of assignment, you would write

```
x += 5;
```

In more general notation, the compound assignment operators have the following syntax (where op represents a binary operator):

```
exp1 op= exp2
```

This is equivalent to writing

```
exp1 = exp1 op exp2;
```

You can create compound assignment operators using the five binary mathematical operators discussed earlier in this chapter. Table 4.10 lists some examples.

Table 4.10. Examples of compound assignment operators.

When You Write This...	**It Is Equivalent To This**
x *= y	x = x * y
y = y - z + 1	y -= z + 1
a = a / b	a /= b
x += y / 8	x = x + y / 8
y %= 3	y = y % 3

The compound operators provide a convenient shorthand, the advantages of which are particularly evident when the variable on the left side of the assignment operator has a long name. As with all other assignment statements, a compound assignment statement is an expression and evaluates to the value assigned to the left side. Thus, executing the following statements results in both x and z having the value 14:

```
x = 12;
z = x += 2;
```

The Conditional Operator

The conditional operator is C's only ternary operator, meaning that it takes three operands. Its syntax is
exp1 ? exp2 : exp3;

If exp1 evaluates to true (that is, nonzero), the entire expression evaluates to the value of exp2. If exp1 evaluates to false (that is, zero), the entire expression evaluates as the value of exp3. For example, the following statement assigns the value 1 to x if y is true and assigns 100 to x if y is false:
x = y ? 1 : 100;

Likewise, to make z equal to the larger of x and y, you could write
z = (x > y) ? x : y;

Perhaps you've noticed that the conditional operator functions somewhat like an if statement.
The preceding statement could also be written like this:

```
if (x > y)
z = x;
else
z = y;
```

The conditional operator can't be used in all situations in place of an if...else construction, but the conditional operator is more concise. The conditional operator can also be used in places you can't use an if statement, such as inside a single printf() statement:
printf("The larger value is %d", ((x > y) ? x : y));

The Comma Operator

The comma is frequently used in C as a simple punctuation mark, serving to separate variable declarations, function arguments, and so on. In certain situations, the comma acts as an operator rather than just as a separator. You can form an expression by separating two subexpressions with a comma. The result is as follows:

- Both expressions are evaluated, with the left expression being evaluated first.
- The entire expression evaluates to the value of the right expression.

For example, the following statement assigns the value of b to x, then increments a, and then increments b:
x = (a++ , b++);
Because the ++ operator is used in postfix mode, the value of b--before it is incremented--is assigned to x. Using parentheses is necessary because the comma operator has low precedence, even lower than the assignment operator.

DO use (expression == 0) instead of (!expression). When compiled, these two expressions evaluate the same; however, the first is more readable.

DO use the logical operators && and || instead of nesting if statements.
DON'T confuse the assignment operator (=) with the equal to (==) operator.

Operator Precedence Revisited

Table 4.11 lists all the C operators in order of decreasing precedence. Operators on the same line have the same precedence.

Table 4.11. C operator precedence.

Level	Operators	Size of
1	() [] -> .	10 |
2	! ~ ++ -- * (indirection) &	11 &&

3 4 5 << 6 7 8 9	(address-of) (type) +(unary) - (unary) * (multiplication) / % +- >> < <= > >= == != & (bitwise AND) ^	12 13 ?: 14 15 , || = += -= *= /= %= &= ^= |= <<= >>=

() is the function operator; [] is the array operator.

TIP: This is a good table to keep referring to until you become familiar with the order of precedence. You might find that you need it later.

Summary
You learned what a C statement is, that white space doesn't matter to a C compiler, and that statements always end with a semicolon. You also learned that a compound statement (or block), which consists of two or more statements enclosed in braces, can be used anywhere a single statement can be used.
Many statements are made up of some combination of expressions and operators. Remember that an expression is anything that evaluates to a numeric value. Complex expressions can contain many simpler expressions, which are called subexpressions.
Operators are C symbols that instruct the computer to perform an operation on one or more expressions. Some operators are unary, which means that they operate on a single operand. Most of C's operators are binary, however, operating on two operands. One operator, the conditional operator, is ternary. C's operators have a defined hierarchy of precedence that determines the order in which operations are performed in an expression that contains multiple operators.
The C operators covered in this chapter fall into three categories:
• Mathematical operators perform arithmetic operations on their operands (for example, addition).

• Relational operators perform comparisons between their operands (for example, greater than).
• Logical operators operate on true/false expressions. Remember that C uses 0 and 1 to represent false and true, respectively, and that any nonzero value is interpreted as being true.

You've also been introduced to C's if statement, which lets you control program execution
based on the evaluation of relational expressions

The if Statement

Relational operators are used mainly to construct the relational expressions used in if and while statements, covered in detail on Day 6, "Basic Program Control." For now, We look for the basics of the if statement to show how relational operators are used to make program control statements.
You might be wondering what a program control statement is. Statements in a C program normally execute from top to bottom, in the same order as they appear in your source code file. A program control statement modifies the order of statement execution. Program control statements can cause other program statements to execute multiple times or to not execute at all, depending on the circumstances. The if statement is one of C's program control statements. Others, such as do and while, are covered on Day 6.

In its basic form, the if statement evaluates an expression and directs program execution depending on the result of that evaluation. The form of an if statement is as follows:

```
if (expression)
   statement;
```

If expression evaluates to true, statement is executed. If expression evaluates to false, statement is not executed. In either case, execution then passes to whatever code follows the if statement. You could say that execution of statement depends on the result of expression.

❑ that both the line if (expression) and the line statement; are considered to comprise the complete if statement; they are not separate statements.

An if statement can control the execution of multiple statements through the use of a compound statement, or block. As defined earlier in this chapter, a block is a group of two or more statements enclosed in braces. A block can be used anywhere a single statement can be used. Therefore, you could write an if statement as follows:

```
if (expression)
{
statement1;
statement2;
   /* additional code goes here */
statement;
}
```

DO remember that if you program too much in one day, you'll get C sick.
DO indent statements within a block to make them easier to read. This includes the statements within a block in an if statement.
DON'T make the mistake of putting a semicolon at the end of an if statement.

An if statement should end with the conditional statement that follows it. In the following, statement1 executes whether or not x equals 2, because each line is evaluated as a separate statement, not together as intended:

```
if( x == 2);          /* semicolon does not belong!  */
statement1;
```

In your programming, you will find that if statements are used most often with relational expressions; in other words, "Execute the following statement(s) only if such-and-such a condition is true." Here's an example:

```
if (x > y)
   y = x;
```

This code assigns the value of x to y only if x is greater than y. If x is not greater than y, no assignment takes place. Listing 4.3 illustrates the use of if statements.

Listing 4.3. LIST0403.C: Demonstrates if statements.

```
/* Demonstrates the use of if statements */

#include <stdio.h>

int x, y;

main()
{
    /* Input the two values to be tested */

     printf("\nInput an integer value for x: ");
     scanf("%d", &x);
     printf("\nInput an integer value for y: ");
     scanf("%d", &y);

     /* Test values and print result */
17:
18:     if (x == y)
19:         printf("x is equal to y\n");
20:
21:     if (x > y)
22:         printf("x is greater than y\n");
23:
24:     if (x < y)
25:         printf("x is smaller than y\n");
26:
27:     return 0;
28: }
```

```
Input an integer value for x: 100
Input an integer value for y: 10
x is greater than y
Input an integer value for x: 10
Input an integer value for y: 100
x is smaller than y
Input an integer value for x: 10
Input an integer value for y: 10
x is equal to y
```

LIST0403.C shows three if statements in action (lines 18 through 25). Many of the lines in this program should be familiar. Line 5 declares two variables, x and y, and lines 11 through 14 prompt the user for values to be placed into these variables. Lines 18 through 25 use if statements to determine whether x is greater than, less than, or equal to y.

❑that line 18 uses an if statement to see whether x is equal to y. Remember that ==, the equal operator, means "is equal to" and should not be confused with =, the assignment operator. After the program checks to see whether the variables are equal, in line 21 it checks to see whether x is greater than y, followed by a check in line 24 to see whether x is less than y. If you think this is inefficient, you're right. In the next program, you will see how to avoid this inefficiency. For now, run the program with different values for x and y to see the results.

❑: You will notice that the statements within an if clause are indented. This is a common practice to aid readability.

The else Clause

An if statement can optionally include an else clause. The else clause is included as follows:

```
if (expression)
   statement1;
else
   statement2;
```

If expression evaluates to true, statement1 is executed. If expression evaluates to false, statement2 is executed. Both statement1 and statement2 can be compound statements or blocks. Listing 4.4 shows Listing 4.3 rewritten to use an if statement with an else clause.

Listing 4.4. An if statement with an else clause.

```
/* Demonstrates the use of if statement with else clause */

#include <stdio.h>

int x, y;

```

```
7:  main()
8:  {
9:     /* Input the two values to be tested */
10:
11:     printf("\nInput an integer value for x: ");
12:     scanf("%d", &x);
13:     printf("\nInput an integer value for y: ");
14:     scanf("%d", &y);
15:
16:     /* Test values and print result */
17:
18:     if (x == y)
19:         printf("x is equal to y\n");
20:     else
21:         if (x > y)
22:             printf("x is greater than y\n");
23:         else
24:             printf("x is smaller than y\n");
25:
26:     return 0;
27:  }
```

```
Input an integer value for x: 99
Input an integer value for y: 8
x is greater than y
Input an integer value for x: 8
Input an integer value for y: 99
x is smaller than y
Input an integer value for x: 99
Input an integer value for y: 99
x is equal to y
```

ANALYSIS: Lines 18 through 24 are slightly different from the previous listing. Line 18 still checks to see whether x equals y. If x does equal y, x is equal to y appears on-screen, just as in Listing 4.3 (LIST0403.C). However, the program then ends, and lines 20 through 24 aren't executed. Line 21 is executed only if x is not equal to y , or, to be more accurate, if the expression "x equals y" is false. If x does not equal y, line 21 checks to see whether x is greater than y. If so, line 22 prints x is greater than y; otherwise (else), line 24 is executed.

Listing 4.4 uses a nested if statement. Nesting means to place (nest) one or more C statements inside another C statement. In the case of Listing 4.4, an if statement is part of the first if statement's else clause.

The if Statement

Form 1

```
if( expression )
   statement1;
next_statement;
```

This is the if statement in its simplest form. If expression is true, statement1 is executed. If expression is not true, statement1 is ignored.

Form 2

```
if( expression )
   statement1;
```

```
else
   statement2;
next_statement;
```

This is the most common form of the if statement. If expression is true, statement1 is executed; otherwise, statement2 is executed.

Form 3

```
if( expression1 )
   statement1;
else if( expression2 )
   statement2;
else
   statement3;
next_statement;
```

This is a nested if. If the first expression, expression1, is true, statement1 is executed before the program continues with the next_statement. If the first expression is not true, the second expression, expression2, is checked. If the first expression is not true, and the second is true, statement2 is executed. If both expressions are false, statement3 is executed. Only one of the three statements is executed.

Example 1

```
if( salary > 45,0000 )
   tax = .30;
else
   tax = .25;
```

Example 2

```
if( age < 18 )
   printf("Minor");
else if( age < 65 )
   printf("Adult");
else
   printf( "Senior Citizen");
```

Evaluating Relational Expressions

Remember that expressions using relational operators are true C expressions that evaluate, by definition, to a value. Relational expressions evaluate to a value of either false (0) or true (1). Although the most common use of relational expressions is within if statements and other conditional constructions, they can be used as purely numeric values. This is illustrated in Listing 4.5.

Listing 4.5. Evaluating relational expressions.

```
/* Demonstrates the evaluation of relational expressions */

#include <stdio.h>

int a;

main()
{
   a = (5 == 5);          /* Evaluates to 1 */
   printf("\na = (5 == 5)\na = %d", a);

   a = (5 != 5);          /* Evaluates to 0 */
   printf("\na = (5 != 5)\na = %d", a);

15:     a = (12 == 12) + (5 != 1); /* Evaluates to 1 + 1 */
16:     printf("\na = (12 == 12) + (5 != 1)\na = %d\n", a);
17:     return 0;
18: }
```

```
a = (5 == 5)
a = 1
a = (5 != 5)
a = 0
a = (12 == 12) + (5 != 1)
a = 2
```

ANNALYSIS: The output from this listing might seem a little confusing at first. Remember, the most common mistake people make when using the relational operators is to use a single equal sign--the assignment operator--instead of a double equal sign. The following expression evaluates to 5 (and also assigns the value 5 to x):

```
x = 5
```

In contrast, the following expression evaluates to either 0 or 1 (depending on whether x is equal to 5) and doesn't change the value of x:

```
x == 5
```

If by mistake you write

```
if (x = 5)
   printf("x is equal to 5");
```

the message always prints because the expression being tested by the if statement always evaluates to true, no matter what the original value of x happens to be.

Looking at Listing 4.5, you can begin to understand why a takes on the values that it does. In line 9, the value 5 does equal 5, so true (1) is assigned to a. In line 12, the statement "5 does not equal 5" is false, so 0 is assigned to a.

To reiterate, the relational operators are used to create relational expressions that ask questions about relationships between expressions. The answer returned by a relational expression is a numeric value of either 1 (representing true) or 0 (representing false).

First, all the relational operators have a lower precedence than the mathematical operators do. Thus, if you write the following, 2 is added to x, and the result is compared to y:

```
if (x + 2 > y)
```

This is the equivalent of the following line, which is a good example of using parentheses for the sake of clarity:

```
if ((x + 2) > y)
```

Although they aren't required by the C compiler, the parentheses surrounding (x + 2) make it clear that it is the sum of x and 2 that is to be compared with y.

There is also a two-level precedence within the relational operators, as shown in Table 4.6.

DON'T put assignment statements in if statements. This can be confusing to other people who look at your code. They might think it's a mistake andchange your assignment to the logical equal statement.

DON'T use the "not equal to" operator (!=) in an if statement containing an else. It's almost always clearer do use the "equal to" operator (==) with an else. For instance, the following code:

```
if ( x != 5 )
statement1;
else
statement2;
```

would be better written as this:

```
if (x == 5 )
statement2;
else
statement1;
```

switch Statement

C's most flexible program control statement is the switch statement, which lets your program execute different statements based on an expression that can have more than two values. Earlier control statements, such as if, were limited to evaluating an expression that could have only two values: true or false. To control program flow based on more than two values, you had to use multiple nested if statements, as shown in Listing 13.4. The switch statement makes such nesting unnecessary.

The general form of the switch statement is as follows:

```
switch (expression)
{
   case  template_1: statement(s);
   case  template_2: statement(s);
   ...
   case  template_n: statement(s);
   default: statement(s);
}
```

In this statement, *expression* is any expression that evaluates to an integer value: type long, int, or char. The switch statement evaluates *expression* and compares the value against the templates following each case label, and then one of the following happens:

- If a match is found between *expression* and one of the templates, execution is transferred to the statement that follows the case label.
- If no match is found, execution is transferred to the statement following the optional default label.
- If no match is found and there is no default label, execution passes to the first statement following the switch statement's closing brace.

The switch statement is demonstrated in Listing 13.5, which displays a message based on the user's input.

Listing 13.5. Using the switch statement.

```
/* Demonstrates the switch statement. */

#include <stdio.h>

main()
{
    int reply;

    puts("Enter a number between 1 and 5:");
    scanf("%d", &reply);

    switch (reply)
    {
        case 1:
            puts("You entered 1.");
        case 2:
            puts("You entered 2.");
        case 3:
            puts("You entered 3.");
        case 4:
            puts("You entered 4.");
        case 5:
            puts("You entered 5.");
        default:
            puts("Out of range, try again.");
    }

    return 0;
}
```

```
Enter a number between 1 and 5:
2
You entered 2.
You entered 3.
You entered 4.
You entered 5.
Out of range, try again.
```

ANALYSIS: Well, that's certainly not right, is it? It looks as though the switch statement finds the first matching template and then executes everything that follows (not just the statements associated with the template). That's exactly what does happen, though. That's how switch is supposed to work. In effect, it performs a goto to the matching template. To ensure that only the statements associated with the matching template are executed, include a break statement where needed. Listing 13.6 shows the program rewritten with break statements.
Now it functions properly.

Listing 13.6. Correct use of switch, including break statements as needed.

```
/* Demonstrates the switch statement correctly. */

#include <stdio.h>

main()
{
    int reply;

    puts("\nEnter a number between 1 and 5:");
    scanf("%d", &reply);

    switch (reply)
    {
      case 0:
          break;
      case 1:
        {
          puts("You entered 1.\n");
          break;
        }
      case 2:
        {
          puts("You entered 2.\n");
          break;
        }
      case 3:
        {
          puts("You entered 3.\n");
          break;
        }
      case 4:
        {
          puts("You entered 4.\n");
          break;
        }
     case 5:
        {
          puts("You entered 5.\n");
          break;
        }
     default:
        {
          puts("Out of range, try again.\n");
        }
   }            /* End of switch */

}
```

```
Enter a number between 1 and 5:
1
You entered 1.
Enter a number between 1 and 5:
6
Out of range, try again.
```

Compile and run this version; it runs correctly.
One common use of the switch statement is to implement the sort of menu shown in Listing

Looping Constructs

while Statement
while (*condition*)
statement(s)
condition is any valid C expression, usually a relational expression. When *condition* evaluates to false (zero), the while statement terminates, and execution passes to the first statement following *statement(s)*; otherwise, the first C statement in *statement(s)* is executed. *statement(s)* is the C statement(s) that is executed as long as *condition* remains true. A while statement is a C looping statement. It allows repeated execution of a statement or block of statements as long as the condition remains true (nonzero). If the condition is not true when the while command is first executed, the *statement(s)* is never executed.

Example 1

```
int x = 0;
while (x < 10)
{
   printf("\nThe value of x is %d", x );
   x++;
}
```

Example 2

```
/* get numbers until you get one greater than 99 */
int nbr=0;
while (nbr <= 99)
   scanf("%d", &nbr );
```

Example 3

```
/* Lets user enter up to 10 integer values       */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops                       */
```

```
int value[10];
int ctr = 0;
int nbr;
while (ctr < 10 && nbr != 99)
{
   puts("Enter a number, 99 to quit ");
   scanf("%d", &nbr);
   value[ctr] = nbr;
   ctr++;
}
```

Nesting while Statements

Just like the for and if statements, while statements can also be nested. Listing 6.4 shows an example of nested while statements. Although this isn't the best use of a while statement, the example does present some new ideas.

Listing 6.4. Nested while statements.

```
/* Demonstrates nested while statements */

#include <stdio.h>

int array[5];

main()
{
int ctr = 0,
nbr = 0;

printf("This program prompts you to enter 5 numbers\n");
printf("Each number should be from 1 to 10\n");

while ( ctr < 5 )
{
nbr = 0;
while (nbr < 1 || nbr > 10)
{
printf("\nEnter number %d of 5: ", ctr + 1 );
scanf("%d", &nbr );
}

array[ctr] = nbr;
ctr++;
}

for (ctr = 0; ctr < 5; ctr++)
printf("Value %d is %d\n", ctr + 1, array[ctr] );

return 0;
}
```

```
This program prompts you to enter 5 numbers
Each number should be from 1 to 10
Enter number 1 of 5: 3
Enter number 2 of 5: 6
Enter number 3 of 5: 3
Enter number 4 of 5: 9
Enter number 5 of 5: 2
Value 1 is 3
Value 2 is 6
Value 3 is 3
Value 4 is 9
Value 5 is 2
```

ANALYSIS: As in previous listings, line 1 contains a comment with a description of the program, and line 3 contains an #include statement for the standard input/output header file. Line 5 contains a declaration for an array (named array) that can hold five integer values. The function main() contains two additional local variables, ctr and nbr (lines 9 and 10). Notice that these variables are initialized to zero at the same time they are declared. Also notice that the comma operator is used as a separator at the end of line 9, allowing nbr to be declared as an int without restating the int type command. Stating declarations in this manner is a common practice for many C programmers. Lines 12 and 13 print messages stating what the program does and what is expected of the user. Lines 15 through 26 contain the first while command and its statements. Lines 18 through 22 also contain a nested while loop with its own statements that are all part of the outer while. This outer loop continues to execute while ctr is less than 5 (line 15). As long as ctr is less than 5, line 17 sets nbr to 0, lines 18 through 22 (the nested while statement) gather a number in variable nbr, line 24 places the number in array, and line 25 increments ctr. Then the loop starts again. Therefore, the outer loop gathers five numbers and places each into array, indexed by ctr. The inner loop is a good use of a while statement. Only the numbers from 1 to 10 are valid, so until the user enters a valid number, there is no point continuing the program. Lines 18 through 22 prevent continuation.
This while statement states that while the number is less than 1 or greater than 10, the program should print a message to enter a number, and then get the number. Lines 28 and 29 print the values that are stored in array. Notice that because the while statements are done with the variable ctr, the for command can reuse it. Starting at zero and incrementing by one, the for loops five times, printing the value of ctr plus one (because the count started at zero) and printing the corresponding value in array.For additional practice, there are two things you can change in this program. The first is the values that the program accepts. Instead of 1 to 10, try making it accept from 1 to 100. You can also change the number of values that it accepts. Currently, it allows for five numbers. Try making it accept 10.

DON'T use the following convention if it isn't necessary:

```
while (x)
```

Instead, use this convention:

```
while (x != 0)
```

Although both work, the second is clearer when you're debugging (trying to find problems in) the code. When compiled, these produce virtually the same code.
DO use the for statement instead of the while statement if you need to initialize and increment within your loop. The for statement keeps the initialization, condition, and increment statements all together. The while statement does not.
The do...while Loop

C's third loop construct is the do...while loop, which executes a block of statements as long as a specified condition is true. The do...while loop tests the condition at the end of the loop rather than at the beginning, as is done by the for loop and the while loop.
The structure of the do...while loop is as follows:

```
do
   statement
while (condition);
```

condition is any C expression, and statement is a single or compound C statement. When program execution reaches a do...while statement, the following events occur:

1. The statements in statement are executed.

2. condition is evaluated. If it's true, execution returns to step 1. If it's false, the loop terminates.

The statements associated with a do...while loop are always executed at least once. This is because the test condition is evaluated at the end, instead of the beginning, of the loop. In contrast, for loops and while loops evaluate the test condition at the start of the loop, so the associated statements are not executed at all if the test condition is initially false.
The do...while loop is used less frequently than while and for loops. It is most appropriate when the statement(s) associated with the loop must be executed at least once. You could, of course, accomplish the same thing with a while loop by making sure that the test condition is true when execution first reaches the loop. A do...while loop probably would be more straightforward, however.

Listing 6.5 shows an example of a do...while loop.

Listing 6.5. A simple do...while loop.

```
/* Demonstrates a simple do...while statement */

#include <stdio.h>

int get_menu_choice( void );

main()
{
    int choice;

     choice = get_menu_choice();

     printf("You chose Menu Option %d\n", choice );

     return 0;
}

int get_menu_choice( void )
{
     int selection = 0;

     do
     {
         printf("\n" );
         printf("\n1 - Add a Record" );
         printf("\n2 - Change a record");
         printf("\n3 - Delete a record");
         printf("\n4 - Quit");
         printf("\n" );
         printf("\nEnter a selection: " );

         scanf("%d", &selection );

      }while ( selection < 1 || selection > 4 );

      return selection;
}
```

```
1 - Add a Record
2 - Change a record
3 - Delete a record
4 - Quit
Enter a selection: 8
```

```
1 - Add a Record
2 - Change a record
3 - Delete a record
4 - Quit
Enter a selection: 4
You chose Menu Option 4
```

ANALYSIS: This program provides a menu with four choices. The user selects one of the four choices, and then the program prints the number selected. Programs later in this book use and expand on this concept. For now, you should be able to follow most of the listing. The main() function (lines 7 through 16) adds nothing to what you already know.

: The body of main() could have been written into one line, like this:

```
printf( "You chose Menu Option %d", get_menu_option() );
```

If you were to expand this program and act on the selection, you would need the value returned by get_menu_choice(), so it is wise to assign the value to a variable (such as choice).

Lines 18 through 37 contain get_menu_choice(). This function displays a menu on-screen (lines 24 through 30) and then gets a selection. Because you have to display a menu at least once to get an answer, it is appropriate to use a do...while loop. In the case of this program, the menu is displayed until a valid choice is entered. Line 34 contains the while part of the do...while statement and validates the value of the selection, appropriately named selection. If the value entered is not between 1 and 4, the menu is redisplayed, and the user is prompted for a new value. When a valid selection is entered, the program continues to line 36, which returns the value in the variable selection.

The do...while Statement

```
do
{
statement(s)
}while (condition);
```

condition is any valid C expression, usually a relational expression. When *condition* evaluates to false (zero), the while statement terminates, and execution passes to the first statement following the while statement; otherwise, the program loops back to the do, and the C statement(s) in *statement(s)* is executed.

statement(s) is either a single C statement or a block of statements that are executed the first time through the loop and then as long as *condition* remains true.

A do...while statement is a C looping statement. It allows repeated execution of a statement or block of statements as long as the condition remains true (nonzero). Unlike the while statement, a do...while loop executes its statements at least once.

Example 1

```
/* prints even though condition fails! */
int x = 10;
do
{
   printf("\nThe value of x is %d", x );
}while (x != 10);
```

Example 2

```
/* gets numbers until the number is greater than 99 */
int nbr;
do
{
   scanf("%d", &nbr );
}while (nbr <= 99);
```

Example 3

```
/* Enables user to enter up to 10 integer values       */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops                             */
int value[10];
int ctr = 0;
int nbr;
do
{
   puts("Enter a number, 99 to quit ");
   scanf( "%d", &nbr);
   value[ctr] = nbr;
   ctr++;
}while (ctr < 10 && nbr != 99);
```

Nested Loops

The term *nested loop* refers to a loop that is contained within another loop. You have seen examples of some nested statements. C places no limitations on the nesting of loops, except that each inner loop must be enclosed completely in the outer loop; you can't have overlapping loops. Thus, the following is not allowed:

```
for ( count = 1; count < 100; count++)
{
   do
   {
      /* the do...while loop */
} /* end of for loop */
   }while (x != 0);
```

If the do...while loop is placed entirely in the for loop, there is no problem:

```
for (count = 1; count < 100; count++)
{
   do
   {
      /* the do...while loop */
   }while (x != 0);

} /* end of for loop */
```

When you use nested loops, remember that changes made in the inner loop might affect the outer loop as well. Note, however, that the inner loop might be independent from any variables in the outer loop; in this example, they are not. In the previous example, if the inner do...while loop modifies the value of count, the number of times the outer for loop executes is affected.
Good indenting style makes code with nested loops easier to read. Each level of loop should be indented one step further than the last level. This clearly labels the code associated with each loop.

DON'T try to overlap loops. You can nest them, but they must be entirely
within each other.
DO use the do...while loop when you know that a loop should be executed at
least once.

for Statement
for (*initial*; *condition*; *increment*)
statement(s)
initial is any valid C expression. It is usually an assignment statement that sets a variable to a particular value.

condition is any valid C expression. It is usually a relational expression. When *condition* evaluates to false (zero), the for statement terminates, and execution passes to the first statement following *statement(s)*; otherwise, the C *statement(s)* in *statement(s)* are executed.
increment is any valid C expression. It is usually an expression that increments a variable initialized by the initial expression.
statement(s) are the C statements that are executed as long as the condition remains true.A for statement is a looping statement. It can have an initialization, test condition, and increment as parts of its command. The for statement executes the initial expression first. It then checks the condition. If the condition is true, the statements execute. Once the statements are completed, the increment expression is evaluated. The for statement then rechecks the condition and continues to loop until the condition is false.

Example 1

```
/* Prints the value of x as it counts from 0 to 9 */
int x;
for (x = 0; x <10; x++)
    printf( "\nThe value of x is %d", x );
```

Example 2

```
/*Obtains values from the user until 99 is entered */
int nbr = 0;
for ( ; nbr != 99; )
   scanf( "%d", &nbr );
```

Example 3

```
/* Lets user enter up to 10 integer values       */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops                          */
int value[10];
int ctr,nbr=0;
for (ctr = 0; ctr < 10 && nbr != 99; ctr++)
{
    puts("Enter a number, 99 to quit ");
    scanf("%d", &nbr);
    value[ctr] = nbr;
}
```

Nesting for Statements

A for statement can be executed within another for statement. This is called *nesting*. (You saw this on Day 4 with the if statement.) By nesting for statements, you can do some complex programming. Listing 6.2 is not a complex program, but it illustrates the nesting of two for statements.

Listing 6.2. Nested for statements.

```
/* Demonstrates nesting two for statements */

#include <stdio.h>

void draw_box( int, int);

main()
{
    draw_box( 8, 35 );

     return 0;
}

```

```
14:  void draw_box( int row, int column )
15:  {
16:      int col;
17:      for ( ; row > 0; row--)
18:      {
19:          for (col = column; col > 0; col--)
20:              printf("X");
21:
22:          printf("\n");
23:      }
24:  }
```

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

ANALYSIS: The main work of this program is accomplished on line 20. When you run this program, 280 Xs are printed on-screen, forming an 8*35 square. The program has only one command to print an X, but it is nested in two loops.

In this listing, a function prototype for draw_box() is declared on line 5. This function takes two type int variables, row and column, which contain the dimensions of the box of Xs to be drawn. In line 9, main() calls draw_box() and passes the value 8 as the row and the value 35 as the column.

Looking closely at the draw_box() function, you might see a couple things you don't readily understand. The first is why the local variable col was declared. The second is why the second printf() in line 22 was used. Both of these will become clearer after you look at the two for loops.

Line 17 starts the first for loop. The initialization is skipped because the initial value of row was passed to the function. Looking at the condition, you see that this for loop is executed until the row is 0. On first executing line 17, row is 8; therefore, the program continues to line 19.

Line 19 contains the second for statement. Here the passed parameter, column, is copied to a local variable, col, of type int. The value of col is 35 initially (the value passed via column), and column retains its original value. Because col is greater than 0, line 20 is executed, printing an X. col is then decremented, and the loop continues. When col is 0, the for loop ends, and control goes to line 22. Line 22 causes the on-screen printing to start on a new line. (Printing is covered in detail on Day 7, "Fundamentals of Input and Output.") After moving to a new line on the screen, control reaches the end of the first for loop's statements, thus executing the increment expression, which subtracts 1 from row, making it 7. This puts control back at line 19. Notice that the value of col was 0 when it was last used. If column had been used instead of col, it would fail the condition test, because it will never be greater than 0. Only the first line would be printed. Take the initializer out of line 19 and change the two col variables to column to see what actually happens.

DON'T put too much processing in the for statement. Although you can use the comma separator, it is often clearer to put some of the functionality into the body of the loop.

DO remember the semicolon if you use a for with a null statement. Put the semi-colon placeholder on a separate line, or place a space between it and the end of the for statement. It's clearer to put it on a separate line.

```
for (count = 0; count < 1000; array[count] = 50) ;
   /* note space! */
```

Functions:The Basics

Functions are central to C programming and to the philosophy of C program design. You've already been introduced to some of C's library functions, which are complete functions supplied as part of your compiler. This chapter covers user-defined functions, which, as the name implies, are functions that you, the programmer, create. Today you will learn

- What a function is and what its parts are
- About the advantages of structured programming with functions
- How to create a function
- How to declare local variables in a function
- How to return a value from a function to the program
- How to pass arguments to a function

What Is a Function?
This chapter approaches the question "What is a function?" in two ways. First, it tells you what functions are, and then it shows you how they're used.

A Function Defined
First the definition: A function is a named, independent section of C code that performs a specific task and optionally returns a value to the calling program. Now let's look at the parts of this definition:

•*Afunction is named*. Each function has a unique name. By using that name in another part of the program, you can execute the statements contained in the function. This is known as *calling* the function. A function can be called from within another function.
•*A function is independent*. A function can perform its task without interference from or interfering with other parts of the program.
•*A function performs a specific task*. This is the easy part of the definition. A task is a discrete job that your program must perform as part of its overall operation, such as sending a line of text to a printer, sorting an array into numerical order, or calculating a cube root.
•*A function can return a value to the calling program*. When your program calls a function, the statements it contains are executed. If you want them to, these statements can pass information back to the calling program.
That's all there is to the "telling" part. Keep the previous definition in mind as you look at the next section.

A Function Illustrated

Listing 5.1. A program that uses a function to calculate the cube of a number.

```
/* Demonstrates a simple function */
#include <stdio.h>

long cube(long x);

long input, answer;

main()
{
   printf("Enter an integer value: ");
   scanf("%d", &input);
   answer = cube(input);
   /* : %ld is the conversion specifier for */
   /* a long integer */
   printf("\nThe cube of %ld is %ld.\n", input, answer);

17:    return 0;
18: }
19:
20: /* Function: cube() - Calculates the cubed value of a variable */
21: long cube(long x)
22: {
23:    long x_cubed;
24:
25:    x_cubed = x * x * x;
26:    return x_cubed;
27: }
```

```
Enter an integer value: 100
The cube of 100 is 1000000.
Enter an integer value: 9
The cube of 9 is 729.
Enter an integer value: 3
The cube of 3 is 27.
```

□**:** The following analysis focuses on the components of the program that relate directly to the function rather than explaining the entire program.

ANALYSIS: Line 4 contains the function prototype, a model for a function that will appear later in the program. A function's prototype contains the name of the function, a list of variables that must be passed to it, and the type of variable it returns, if any. Looking at line 4, you can tell that the function is named cube, that it requires a variable of the type long, and that it will return a value of type long. The variables to be passed to the function are called arguments, and they are enclosed in parentheses following the function's name. In this example, the function's argument is long x. The keyword before the name of the function indicates the type of variable the function returns. In this case, a type long variable is returned. Line 12 calls the function cube and passes the variable input to it as the function's argument. The function's return value is assigned to the variable answer. Notice that both input and answer are declared on line 6 as long variables, in keeping with the function prototype on line 4.

The function itself is called the function definition. In this case, it's called cube and is contained in lines 21 through 27. Like the prototype, the function definition has several parts. The function starts out with a function header on line 21. The function header is at the start of a function, and it gives the function's name (in this case, the name is cube). The header also gives the function's return type and describes its arguments. Note that the function header is identical to the function prototype (minus the semicolon). The body of the function, lines 22 through 27, is enclosed in braces. The body contains statements, such as on line 25, that are executed whenever the function is called. Line 23 is a variable declaration that looks like the declarations you have seen before, with one difference:
it's local. Local variables are declared within a function body. (Local declarations are discussed further , "Understanding Variable Scope.") Finally, the function concludes with a return statement on line 26, which signals the end of the function. A return statement also passes a value back to the calling program. In this case, the value of the variable x_cubed is returned.If you compare the structure of the cube() function with that of the main() function, you'll see that they are the same. main() is also a function. Other functions that you already have used are printf() and scanf(). Although printf() and scanf() are library functions (as opposed to user-defined functions), they are functions that can take arguments and return values just like the functions you create.

How a Function Works

A C program doesn't execute the statements in a function until the function is called by another part of the program. When a function is called, the program can send the function information in the form of one or more arguments. An argument is program data needed by the function to perform its task. The statements in the function then execute, performing whatever task each was designed to do. When the function's statements have finished, execution passes back to the same location in the program that called the function. Functions can send information back to the program in the form of a return value.

Functions
Function Prototype
return_type function_name(*arg-type name-1*,...,*arg-type name-n*);
Function Definition

```
return_type function_name( arg-type name-1,...,arg-type name-n)
{
  /* statements; */
}
```

A function prototype provides the compiler with a description of a function that will be defined at a later point in the program. The prototype includes a return type indicating the type of variable that the function will return. It also includes the function name, which should describe what the function does. The prototype also contains the variable types of the arguments (arg type) that will be passed to the function. Optionally, it can contain the names of the variables that will be passed. A prototype should always end with a semicolon.
A function definition is the actual function. The definition contains the code that will be executed. The first line of a function definition, called the function header, should be identical to the function prototype, with the exception of the semicolon. A function header shouldn't end with a semicolon. In addition, although the argument variable names were optional in the prototype, they must be included in the function header. Following the header is the function body, containing the statements that the function will perform. The function body should start with an opening bracket and end with a closing bracket. If the function return type is anything other than void, a return statement should be included, returning a value matching the return type.

Function Prototype Examples

```
double squared( double number );
void print_report( int report_number );
int get_menu_choice( void );
```

Function Definition Examples

```
double squared( double number )         /* function header */
{                                /* opening bracket */
  return(  number * number );         /* function body   */
}                                /* closing bracket */
void print_report( int report_number )

{
  if( report_number == 1 )
     puts( "Printing Report 1" );
  else
     puts( "Not printing Report 1" );
}
```

Functions and Structured Programming
By using functions in your C programs, you can practice structured programming, in which individual program tasks are performed by independent sections of program code. "Independent sections of program code" sounds just like part of the definition of functions given earlier, doesn't it? Functions and structured programming are closely related.

The Advantages of Structured Programming

Why is structured programming so great? There are two important reasons:

• It's easier to write a structured program, because complex programming problems are broken into a number of smal er, simpler tasks. Each task is performed by a function in which code and variables are isolated from the rest of the program. You can progress more quickly by dealing with these relatively simple tasks one at a time.
• It's easier to debug a structured program. If your program has a *bug* (something that causes it to work improperly), a structured design makes it easy to isolate the problem to a specific section of code (a specific function).
A related advantage of structured programming is the time you can save. If you write a function to perform a certain task in one program, you can quickly and easily use it in another program that needs to execute the same task. Even if the new program needs to accomplish a slightly different task, you'll often find that modifying a function you created earlier is easier than writing a new one from scratch. Consider how much you've used the two functions printf() and scanf() even though you probably haven't seen the code they contain. If your functions have been created to perform a single task, using them in other programs is much easier.

Planning a Structured Program
If you're going to write a structured program, you need to do some planning first. This planning should take place before you write a single line of code, and it usually can be done with nothing more than pencil and paper. Your plan should be a list of the specific tasks your program performs. Begin with a global idea of the program's function. If you were planning a program to manage your name and address list, what would you want the program to do?

Here are some obvious things:

• Enter new names and addresses.
• Modify existing entries.
• Sort entries by last name.
• Print mailing labels.
With this list, you've divided the program into four main tasks, each of which can be assigned to a function. Now you can go a step further, dividing these tasks into subtasks. For example, the "Enter new names and addresses" task can be subdivided into these subtasks:
• Read the existing address list from disk.
• Prompt the user for one or more new entries.
• Add the new data to the list.
• Save the updated list to disk.
Likewise, the "Modify existing entries" task can be subdivided as follows:
• Read the existing address list from disk.
• Modify one or more entries.
• Save the updated list to disk.

You might have noticed that these two lists have two subtasks in common--the ones dealing with reading from and saving to disk. You can write one function to "Read the existing address list from disk," and that function can be called by both the "Enter new names and addresses" function and the "Modify existing entries" function. The same is true for "Save the updated list to disk." Already you should see at least one advantage of structured programming. By carefully dividing the program into tasks, you can identify parts of the program that share common tasks. You can write "double-duty" disk access functions, saving yourself time and making your program smaller and more efficient.
When you follow this planned approach, you quickly make a list of discrete tasks that your program needs to perform. Then you can tackle the tasks one at a time, giving all your attention to one relatively simple task. When that function is written and working properly, you can move on to the next task. Before you know it, your program starts to take shape.

The Top-Down Approach
By using structured programming, C programmers take the top-down approach. where the program's structure resembles an inverted tree. Many times, most of the real work of the program is performed by

the functions at the tips of the "branches." The functions closer to the "trunk" primarily direct program execution among these functions.
As a result, many C programs have a small amount of code in the main body of the program--that is, in main(). The bulk of the program's code is found in functions. In main(), all you might find are a few dozen lines of code that direct program execution among the functions. Often, a menu is presented to the person using the program. Program execution is branched according to the user's choices. Each branch of the menu uses a different function.Now that you know what functions are and why they're so important, the time has come for you to learn how to write your own.
DO plan before starting to code. By determining your program's structure ahead of time, you can save time writing the code and debugging it.
DON'T try to do everything in one function. A single function should perform a single task, such as reading information from a file.

Writing a Function
The first step in writing a function is knowing what you want the function to do. Once you know that, the actual mechanics of writing the function aren't particularly difficult.

The Function Header
The first line of every function is the function header, which has three components, each serving a specific function.

The Function Return Type

The function return type specifies the data type that the function returns to the calling program. The return type can be any of C's data types: char, int, long, float, or double. You can also define a function that doesn't return a value by using a return type of void. Here are some examples:

```
int func1(...)          /* Returns a type int.   */
float func2(...)        /* Returns a type float. */
void func3(...)         /* Returns nothing.      */
```

The Function Name
You can name a function anything you like, as long as you follow the rules for C variable names . A function name must be unique (not assigned to any other function or variable). It's a good idea to assign a name that reflects what the function does.

The Parameter List
Many functions use arguments, which are values passed to the function when it's called. A function needs to know what kinds of arguments to expect--the data type of each argument. You can pass a function any of C's data types. Argument type information is provided in the function header by the parameter list.For each argument that is passed to the function, the parameter list must contain one entry. This entry specifies the data type and the name of the parameter. For example, here's the header from the function in Listing 5.1:
long cube(long x)

The parameter list consists of long x, specifying that this function takes one type long argument, represented by the parameter x. If there is more than one parameter, each must be separated by a comma. The function header void func1(int x, float y, char z) specifies a function with three arguments: a type int named x, a type float named y, and a type char named z. Some functions take no arguments, in which case the parameter list should consist of void, like this:

void *func2*(void)

❑**:** You do not place a semicolon at the end of a function header. If you mistakenly include one, the compiler will generate an error message.

Sometimes confusion arises about the distinction between a parameter and an argument. A parameter is an entry in a function header; it serves as a "placeholder" for an argument. A function's parameters are fixed; they do not change during program execution.
An argument is an actual value passed to the function by the calling program. Each time a function is called, it can be passed different arguments. In C, a function must be passed with the same number and type of arguments each time it's called, but the argument values can be different. In the function, the argument is accessed by using the corresponding parameter name.

An example will make this clearer. Listing 5.2 presents a very simple program with one function that is called twice.

Listing 5.2. The difference between arguments and parameters.

```
1:  /* Illustrates the difference between arguments and parameters. */
2:
3:  #include <stdio.h>
4:
5:  float x = 3.5, y = 65.11, z;
6:
7:  float half_of(float k);
8:
9:  main()
10: {
11:     /* In this call, x is the argument to half_of(). */
12:     z = half_of(x);
13:     printf("The value of z = %f\n", z);
14:
15:     /* In this call, y is the argument to half_of(). */
16:     z = half_of(y);
17:     printf("The value of z = %f\n", z);
18:
19:     return 0;
20: }
21:
22: float half_of(float k)
23: {
24:     /* k is the parameter. Each time half_of() is called, */
25:     /* k has the value that was passed as an argument. */
26:
27:     return (k/2);
28: }
```

```
The value of z = 1.750000
The value of z = 32.555000
```

ANALYSIS: Looking at Listing 5.2, you can see that the half_of() function prototype is declared on line 7. Lines 12 and 16 call half_of(), and lines 22 through 28 contain the actual function. Lines 12 and 16 each send a different argument to half_of(). Line 12 sends x, which contains a value of 3.5, and line 16 sends y, which contains a value of 65.11. When the program runs, it prints the correct number for each. The values in x and y are passed into the argument k of half_of(). This is like copying the values from x to k, and then from y to k. half_of() then returns this value after dividing it by 2 (line 27).

DO use a function name that describes the purpose of the function.
DON'T pass values to a function that it doesn't need.
DON'T try to pass fewer (or more) arguments to a function than there are parameters. In C programs, the number of arguments passed must match the number of parameters.

The Function Body

The function body is enclosed in braces, and it immediately follows the function header. It's here that the real work is done. When a function is called, execution begins at the start of the function body and terminates (returns to the calling program) when a return statement is encountered or when execution reaches the closing brace.

Local Variables

You can declare variables within the body of a function. Variables declared in a function arecalled local variables. The term local means that the variables are private to that particular function and are distinct from other variables of the same name declared elsewhere in the program. This will be explained shortly; for now, you should learn how to declare local variables.

A local variable is declared like any other variable, using the same variable types and rules for names that you learned on Day 3. Local variables can also be initialized when they are declared. You can declare any of C's variable types in a function. Here is an example of four local variables being declared within a function:

```
int func1(int y)
{
   int a, b = 10;
   float rate;
   double cost = 12.55;
   /* function code goes here... */
}
```

The preceding declarations create the local variables a, b, rate, and cost, which can be used by the code in the function. □ that the function parameters are considered to be variable declarations, so the variables, if any, in the function's parameter list also are available.

When you declare and use a variable in a function, it is totally separate and distinct from any other variables that are declared elsewhere in the program. This is true even if the variables have the same name.

Listing 5.3 demonstrates this independence.

Listing 5.3. A demonstration of local variables.

```
/* Demonstrates local variables. */

#include <stdio.h>

int x = 1, y = 2;

void demo(void);

main()
{
printf("\nBefore calling demo(), x = %d and y = %d.", x, y);
demo();
printf("\nAfter calling demo(), x = %d and y = %d\n.", x, y);

return 0;
}

void demo(void)
{
/* Declare and initialize two local variables. */

int x = 88, y = 99;

/* Display their values. */
25:
26: printf("\nWithin demo(), x = %d and y = %d.", x, y);
27: }
```

```
Before calling demo(), x = 1 and y = 2.
Within demo(), x = 88 and y = 99.
After calling demo(), x = 1 and y = 2.
```

ANALYSIS: Listing 5.3 is similar to the previous programs in this chapter. Line 5 declares variables x and y. These are declared outside of any functions and therefore are considered. global. Line 7 contains the prototype for our demonstration function, named demo(). It doesn't take any parameters, so it has void in the prototype. It also doesn't return any values, giving it a type of void. Line 9 starts our main() function, which is very simple. First, printf() is called on line 11 to display the values of x and y, and then the demo() function is called. Notice that demo() declares its own local versions of x and y on line 22. Line 26 shows that the local variables take precedence over any others. After the demo function is called, line 13 again prints the values of x and y. Because you are no longer in demo(), the original global values are printed. When you declare and use a variable in a function, it is totally separate and distinct from any other variables that are declared elsewhere in the program. This is true even if the variables have the same name. Three rules govern the use of variables in functions:

To use a variable in a function, you must declare it in the function header or the function body (except for global variables, which are covered on Day 12. In order for a function to obtain a value from the calling program, the value must be passed as an argument. In order for a calling program to obtain a value from a function, the value must be explicitly returned from the function.

To be honest, these "rules" are not strictly applied, because you'll learn how to get around them later in this book. However, follow these rules for now, and you should stay out of trouble.

Keeping the function's variables separate from other program variables is one way in which functions are independent. A function can perform any sort of data manipulation you want, using its own set of local variables. There's no worry that these manipulations will have an unintended effect on another part of the program.

Function Statements

There is essentially no limitation on the statements that can be included within a function. The only thing you can't do inside a function is define another function. You can, however, use all other C statements, including loops (these are covered on Day 6, "Basic Program Control"), if statements, and assignment statements. You can call library functions and other user-defined functions.
What about function length? C places no length restriction on functions, but as a matter of practicality, you should keep your functions relatively short. Remember that in structured programming, each function is supposed to perform a relatively simple task. If you find that a function is getting long, perhaps you're trying to perform a task too complex for one function alone. It probably can be broken into two or more smaller functions.
How long is too long? There's no definite answer to that question, but in practical experience it's rare to find a function longer than 25 or 30 lines of code. You have to use your own judgment. Some programming tasks require longer functions, whereas many functions are only a few lines long. As you gain programming experience, you will become more adept at determining what should and shouldn't be broken into smaller functions.

Returning a Value

To return a value from a function, you use the return keyword, followed by a C expression. When execution reaches a return statement, the expression is evaluated, and execution passes the value back to the calling program. The return value of the function is the value of the expression. Consider this function:

```
int func1(int var)
{
   int x;
```

```
    /* Function code goes here... */
    return x;
}
```

When this function is called, the statements in the function body execute up to the return statement. The return terminates the function and returns the value of x to the calling program. The expression that follows the return keyword can be any valid C expression.
A function can contain multiple return statements. The first return executed is the only one that has any effect. Multiple return statements can be an efficient way to return different values from a function, as demonstrated in Listing 5.4.

Listing 5.4. Using multiple return statements in a function.

```
1:   /* Demonstrates using multiple return statements in a function. */
2:
3:   #include <stdio.h>
4:
5:   int x, y, z;
6:
7:   int larger_of( int , int );
8:
9:   main()
10:  {
11:      puts("Enter two different integer values: ");
12:      scanf("%d%d", &x, &y);
13:
14:      z = larger_of(x,y);
15:
16:      printf("\nThe larger value is %d.", z);
17:
18:      return 0;
19:  }
20:
21:  int larger_of( int a, int b)
22:  {
23:      if (a > b)
24:          return a;
25:      else
26:          return b;
27:  }
```

```
Enter two different integer values:
200 300
The larger value is 300.
Enter two different integer values:
300
200
The larger value is 300.
```

ANALLYSIS: As in other examples, Listing 5.4 starts with a comment to describe what the program does (line 1). The STDIO.H header file is included for the standard input/output functions that allow the program to display information to the screen and get user input. Line 7 is the function prototype for larger_of(). Notice that it takes two int variables for parameters and returns an int. Line 14 calls larger_of() with x and y. The function larger_of() contains the multiple return statements. Using an if statement, the function checks to see whether a is bigger than b on line 23. If it is, line 24 executes a return statement, and the function immediately ends. Lines 25 and 26 are ignored in this case. If a isn't bigger than b, line 24 is skipped, the else clause is instigated, and the return on line 26 executes. You should be able to see that, depending on the arguments passed to the function larger_of(), either the first or the second return statement is executed, and the appropriate value is passed back to the calling function.

One final note on this program. Line 11 is a new function that you haven't seen before. puts()--meaning *put string*--is a simple function that displays a string to the standard output, usually the computer screen. (Strings are covered on Day 10, "Characters and Strings." For now, know that they are just quoted text.)
Remember that a function's return value has a type that is specified in the function header and function prototype. The value returned by the function must be of the same type, or the compiler generates an error message.

NOTE: Structured programming suggests that you have only one entry and one exit in a function. This means that you should try to have only one return statement within your function. At times, however, a program might be much easier to read and maintain with more than one return statement. In such cases, maintainability should take precedence.

The Function Prototype

A program must include a prototype for each function it uses. You saw an example of a function prototype on line 4 of Listing 5.1, and there have been function prototypes in the other listings as well. What is a function prototype, and why is it needed?
You can see from the earlier examples that the prototype for a function is identical to the function header, with a semicolon added at the end. Like the function header, the function prototype includes information about the function's return type, name, and parameters. The prototype's job is to tell the compiler about the function's return type, name, and parameters. With this information, the compiler can check every time your source code calls the function and verify that you're passing the correct number and type of arguments to the function and using the return value correctly. If there's a mismatch, the compiler generates an error message.
Strictly speaking, a function prototype doesn't need to exactly match the function header. The parameter names can be different, as long as they are the same type, number, and in the same order. There's no reason for the header and prototype not to match; having them identical makes source code easier to understand. Matching the two also makes writing a program easier. When you complete a function definition, use your editor's cut-and-paste feature to copy the function header and create the prototype. Be sure to add a semicolon at the end.
Where should function prototypes be placed in your source code? They should be placed before the start of main() or before the first function is defined. For readability, it's best to group all prototypes in one location.

DON'T try to return a value that has a type different from the function's type.
DO use local variables whenever possible.
DON'T let functions get too long. If a function starts getting long, try to break it into separate, smaller tasks.
DO limit each function to a single task.
DON'T have multiple return statements if they aren't needed. You should try to have one return when possible; however, sometimes having multiple return statements is easier and clearer.

Passing Arguments to a Function

To pass arguments to a function, you list them in parentheses following the function name. The number of arguments and the type of each argument must match the parameters in the function header and prototype. For example, if a function is defined to take two type int arguments, you must pass it exactly two int arguments--no more, no less--and no other type.

If you try to pass a function an incorrect number and/or type of argument, the compiler will detect it, based on the information in the function prototype.If the function takes multiple arguments, the arguments listed in the function call are assigned to the function parameters in order: the first argument to the first parameter, the second argument to the second parameter, and so on.
Each argument can be any valid C expression: a constant, a variable, a mathematical or logical expression, or even another function (one with a return value). For example, if half(), square(), and third() are all functions with return values, you could write

```
x = half(third(square(half(y))));
```

The program first calls half(), passing it y as an argument. When execution returns from half(), the program calls square(), passing half()'s return value as an argument. Next, third() is called with square()'s return value as the argument. Then, half() is called again, this time with third()'s return value as an argument. Finally, half()'s return value is assigned to the variable x.
The following is an equivalent piece of code:

```
a = half(y);
b = square(a);
c = third(b);
x = half(c);
```

Calling Functions

There are two ways to call a function. Any function can be called by simply using its name and argument list alone in a statement, as in the following example. If the function has a return value, it is discarded.

```
wait(12);
```

The second method can be used only with functions that have a return value. Because these functions evaluate to a value (that is, their return value), they are valid C expressions and can be used anywhere a C expression can be used. You've already seen an expression with a return value used as the right side of an assignment statement. Here are some more examples.

In the following example, half_of() is a parameter of a function:

```
printf("Half of %d is %d.", x, half_of(x));
```

First, the function half_of() is called with the value of x, and then printf() is called using the values x and half_of(x).

In this second example, multiple functions are being used in an expression:

```
y = half_of(x) + half_of(z);
```

Although half_of() is used twice, the second call could have been any other function. The following code shows the same statement, but not all on one line:

```
a = half_of(x);
b = half_of(z);
y = a + b;
```

The final two examples show effective ways to use the return values of functions. Here, a function is being used with the if statement:

```
if ( half_of(x) > 10 )
{
   /* statements; */          /* these could be any statements! */
}
```

If the return value of the function meets the criteria (in this case, if half_of() returns a value greater than 10), the if statement is true, and its statements are executed. If the returned value doesn't meet the criteria, the if's statements are not executed.

The following example is even better:

```
if ( do_a_process() != OKAY )
{
   /* statements; */          /* do error routine */
}
```

Again, we haven't given the actual statements, nor is do_a_process() a real function;

however, this is an important example that checks the return value of a process to see whether it ran all right. If it didn't, the statements take care of any error handling or cleanup. This is commonly used with accessing information in files, comparing values, and allocating memory.If you try to use a function with a void return type as an expression, the compiler generates an error message.

DO pass parameters to functions in order to make the function generic and thus reusable.
DO take advantage of the ability to put functions into expressions.
DON'T make an individual statement confusing by putting a bunch of functions in it. You should put functions into your statements only if they don't make the code more confusing.

Recursion

The term *recursion* refers to a situation in which a function calls itself either directly or indirectly. *Indirect recursion* occurs when one function calls another function that then calls the first function. C allows recursive functions, and they can be useful in some situations.

For example, recursion can be used to calculate the factorial of a number. The factorial of a number x is written x! and is calculated as follows:
x! = x * (x-1) * (x-2) * (x-3) * ... * (2) * 1
However, you can also calculate x! like this:
x! = x * (x-1)!

Going one step further, you can calculate (x-1)! using the same procedure:
(x-1)! = (x-1) * (x-2)!

You can continue calculating recursively until you're down to a value of 1, in which case you're finished. The program in Listing 5.5 uses a recursive function to calculate factorials. Because the program uses unsigned integers, it's limited to an input value of 8; the factorial of 9 and larger values are outside the allowed range for integers.

Listing 5.5. Using a recursive function to calculate factorials.

```
/* Demonstrates function recursion. Calculates the */
/* factorial of a number. */

#include <stdio.h>

unsigned int f, x;
unsigned int factorial(unsigned int a);

main()
{
    puts("Enter an integer value between 1 and 8: ");
    scanf("%d", &x);

14:     if( x > 8 || x < 1)
15:     {
16:         printf("Only values from 1 to 8 are acceptable!");
17:     }
18:     else
19:     {
20:         f = factorial(x);
21:         printf("%u factorial equals %u\n", x, f);
22:     }
23:
24:     return 0;
25: }
26:
27: unsigned int factorial(unsigned int a)
28: {
29:     if (a == 1)
30:         return 1;
31:     else
32:     {
33:         a *= factorial(a-1);
34:         return a;
35:     }
36: }
```

```
Enter an integer value between 1 and 8:
6
6 factorial equals 720
```

ANALYSIS: The first half of this program is like many of the other programs you have worked with so far. It starts with comments on lines 1 and 2. On line 4, the appropriate header file is included for the input/output routines. Line 6 declares a couple of unsigned integer values. Line 7 is a function prototype for the factorial function. Notice that it takes an unsigned int as its parameter and returns an unsigned int. Lines 9 through 25 are the main() function. Lines 11 and 12 print a message asking for a value from 1 to 8 and then accept an entered value. Lines 14 through 22 show an interesting if statement. Because a value greater than 8 causes a problem, this if statement checks the value. If it's greater than 8, an error message is printed; otherwise, the program figures the factorial on line 20 and prints the result on line 21. When you know there could be a problem, such as a limit on the size of a number, add code to detect the problem and prevent it.

Our recursive function, factorial(), is located on lines 27 through 36. The value passed is assigned to a. On line 29, the value of a is checked. If it's 1, the program returns the value of 1. If the value isn't 1, a is set equal to itself times the value of factorial(a-1). The program calls the factorial function again, but this time the value of a is (a-1). If (a-1) isn't equal to 1, factorial() is called again with ((a-1)-1), which is the same as (a-2). This process continues until the if statement on line 29 is true. If the value of the factorial is 3, the factorial is evaluated to the following:

3 * (3-1) * ((3-1)-1)

DO understand and work with recursion before you use it.

DON'T use recursion if there will be several iterations. (An iteration is the repetition of a program statement.) Recursion uses many resources, because the function has to remember where it is.

Where the Functions Belong

You might be wondering where in your source code you should place your function definitions. For now, they should go in the same source code file as main() and after the end of main().You can keep your user-defined functions in a separate source-code file, apart from main(). This technique is useful with large programs and when you want to use the same set of functions in more than one program.

Summary

This chapter introduced you to functions, an important part of C programming. Functions are independent sections of code that perform specific tasks. When your program needs a task performed, it calls the function that performs that task. The use of functions is essential for structured programming--a method of program design that emphasizes a modular, top-down approach. Structured programming creates more efficient programs and also is much easier for you, the programmer, to use.

You also learned that a function consists of a header and a body. The header includes information about the function's return type, name, and parameters. The body contains local variable declarations and the C statements that are executed when the function is called. Finally, you saw that local variables--those declared within a function--are totally independent of any other program variables declared elsewhere.

The #include Directive

You have already learned how to use the #include preprocessor directive to include header files in your program. When it encounters an #include directive, the preprocessor reads the specified file and inserts it at the location of the directive. You can't use the * or ? wildcards to read in a group of files with one #include directive. You can, however, nest #include directives. In other words, an included file can contain #include directives, which can contain #include directives, and so on. Most compilers limit the number of levels deep that you can nest, but you usually can nest up to 10 levels.

There are two ways to specify the filename for an #include directive. If the filename is enclosed in angle brackets, such as #include <stdio.h> (as you have seen throughout this book), the preprocessor first looks for the file in the standard directory. If the file isn't found, or no standard directory is specified, the preprocessor looks for the file in the current directory.

"What is the standard directory?" you might be asking. In DOS, it's the directory or directories specified by the DOS INCLUDE environment variable. Your DOS documentation contains complete information on the DOS environment. To summarize, however, you set an environment variable with a SET command (usually, but not necessarily, in your AUTOEXEC.BAT file). Most compilers automatical y set the INCLUDE variable in the AUTOEXEC.BAT file when the compiler is installed.

The second method of specifying the file to be included is enclosing the filename in double quotation marks: #include "myfile.h". In this case, the preprocessor doesn't search the standard directories; instead, it looks in the directory containing the source code file being compiled. Generally speaking, header files that you write should be kept in the same directory as the C source code files, and they are included by using double quotation marks. The standard directory is reserved for header files supplied with your compiler.

#define Preprocessor Directive

The #define preprocessor directive has two uses: creating symbolic constants and creating macros.

Simple Substitution Macros Using #define

You learned about substitution macros on Day 3, "Storing Data: Variables and Constants,"although the term used to describe them in that chapter was *symbolic constants*. You create a substitution macro by using #define to replace text with other text. For example, to replace text1 with text2, you write

```
#define text1 text2
```

This directive causes the preprocessor to go through the entire source code file, replacing every occurrence of text1 with text2. The only exception occurs if text1 is found within double quotation marks, in which case no change is made.

The most frequent use for substitution macros is to create symbolic constants, as explained on Day 3. For example, if your program contains the following lines:

```
#define MAX 1000
x = y * MAX;
z = MAX - 12;
```

during preprocessing, the source code is changed to read as follows:

Module 4: Functions and Program Structure

Module Objective: the participants will understand

- How to Break large computing tasks into smaller ones
- Understand scope and lifetime of a variable in ‘C’ program
- Understand the pre processing stage of a ‘C’ program

Chapter 1: Functions

Standard Input output function
Writing functions
Prototype declaration
Function definition
Calling functions
Pitfalls of missing or not matching function declaration and definition
Return types
Returning void
Formal parameters and function arguments
Parameter passing Pass-by-value
Nested function call
Inbuilt functions
Recursive functions
Mathematical Functions

Chapter 2: Variables
Scoping Variables
Local variables
Global variables
Storage Specifiers

- auto
- extern
- register
- static

Chapter 3: Role of Preprocessors
Including files using **#include**
Macro definitions **#define**

Module 5: Arrays and Pointers

Module Objective: The participants will understand

- How to store, manage and manipulate information of same type
- Understand memory architecture that caters to applications which requires dynamic scenario How to use library string functions.

Chapter 1: Arrays
Single dimension array
Operators [] for accessing an array
Array initialization and rules
Address arithmetic on arrays

Chapter 2: Pointers in C Language
Pointer as address variable
Parameter passing – Passing address by value
Pointers in relation to arrays and addresses
Pointer manipulation using **[]** and *
Pitfalls of pointer usage
Void * and its importance

Chapter 3: Working with Strings
Char array and null terminator '\0'
Basic string functions – strcpy, strlen, strcmp, strcat
gets()
puts()
Using strcmpi, strupr, strlwr, strrev, strset, strstr

Arrays: The Basics
Before we cover the for statement, let's take a short detour and learn the basics of arrays. The for statement and arrays are closely linked in C, so it's difficult to define one without explaining the other. To help you understand the arrays used in the for statement examples to come, a quick treatment of arrays follows.
An *array* is an indexed group of data storage locations that have the same name and are distinguished from each other by a *subscript,* or *index*--a number following the variable name, enclosed in brackets. (This will become clearer as you continue.) Like other C variables, arrays must be declared. An array declaration includes both the data type and the size of the array (the number of elements in the array). For example, the following statement declares an array named data that is type int and has 1,000 elements:

```
int data[1000];
```

The individual elements are referred to by subscript as data[0] through data[999]. The first element is data[0], not data[1]. In other languages, such as BASIC, the first element of an array is 1; this is not true in C.
Each element of this array is equivalent to a normal integer variable and can be used the same way. The subscript of an array can be another C variable, as in this example:

```
int data[1000];
int count;
count = 100;
```

data[count] = 12;   /* The same as data[100] = 12 */
This has been a quick introduction to arrays. However, you should now be able to understand how arrays are used in the program examples later in this chapter. If every detail of arrays isn't clear to you, don't worry. You will learn more about arrays on Day 8.

DON'T declare arrays with subscripts larger than you will need; it wastes memory.
DON'T forget that in C, arrays are referenced starting with subscript 0, not 1.

What Is an Array?

An *array* is a collection of data storage locations, each having the same data type and the same name. Each storage location in an array is called an *array element*. Why do you need arrays in your programs? This question can be answered with an example. If you're keeping track of your business expenses for 1998 and filing your receipts by month, you could have a separate folder for each month's receipts, but it would be more convenient to have a single folder with 12 compartments.
Extend this example to computer programming. Imagine that you're designing a program to keep track of your business expenses. The program could declare 12 separate variables, one for each month's expense total. This approach is analogous to having 12 separate folders for your receipts. Good programming practice, however, would utilize an array with 12 elements, storing each month's total in the corresponding array element. This approach is comparable to filing your receipts in a single folder with 12 compartments.

Single-Dimensional Arrays

A *single-dimensional array* has only a single subscript. A *subscript* is a number in brackets that follows an array's name. This number can identify the number of individual elements in the array. An example should make this clear. For the business expenses program, you could use the following line to declare an array of type float:
float expenses[12];

The array is named expenses, and it contains 12 elements. Each of the 12 elements is the exact equivalent of a single float variable. All of C's data types can be used for arrays. C array elements are always numbered starting at 0, so the 12 elements of expenses are numbered 0 through 11. In the preceding example, January's expense total would be stored in expenses[0], February's in expenses[1], and so on.
When you declare an array, the compiler sets aside a block of memory large enough to hold the entire array.

The location of array declarations in your source code is important. As with nonarray variables, the declaration's location affects how your program can use the array. The effect of a declaration's location is covered in more detail on Day 12, "Understanding Variable Scope." For now, place your array declarations with other variable declarations, just before the start of main().
An array element can be used in your program anywhere a nonarray variable of the same type can be used. Individual elements of the array are accessed by using the array name followed by the element subscript enclosed in square brackets. For example, the following statement stores the value 89.95 in the second array element (remember, the first array element is expenses[0], not expenses[1]):
expenses[1] = 89.95;
Likewise, the statement
expenses[10] = expenses[11];
assigns a copy of the value that is stored in array element expenses[11] into array element expenses[10].
When you refer to an array element, the array subscript can be a literal constant, as in these examples. However, your programs might often use a subscript that is a
C integer variable or expression, or even another array element. Here are some examples:
float expenses[100];

```
int a[10];
/* additional statements go here */
expenses[i] = 100;        /* i is an integer variable */
expenses[2 + 3] = 100;    /* equivalent to expenses[5] */
expenses[a[2]] = 100;     /* a[] is an integer array */
```

That last example might need an explanation. If, for instance, you have an integer array named a[] and the value 8 is stored in element a[2], writing

```
expenses[a[2]]
```

has the same effect as writing

```
expenses[8];
```

When you use arrays, keep the element numbering scheme in mind: In an array of *n* elements, the allowable subscripts range from 0 to *n*-1. If you use the subscript value *n,* you might get program errors. The C compiler doesn't recognize whether your program uses an array subscript that is out of bounds. Your program compiles and links, but out-of-range subscripts generally produce erroneous results.

WARNING: Remember that array elements start with 0, not 1. Also remember that the last element is one less than the number of elements in the array. For example, an array with 10 elements contains elements 0 through 9.

Sometimes you might want to treat an array of n elements as if its elements were numbered 1 through *n*. For instance, in the previous example, a more natural method might be to store January's expense total in expenses[1], February's in expenses[2], and so on. The simplest way to do this is to declare the array with one more element than needed, and ignore element 0. In this case, you would declare the array as follows. You could also store some related data in element 0 (the yearly expense total, perhaps).

```
float expenses[13];
```

The program EXPENSES.C in Listing 8.1 demonstrates the use of an array. This is a simple program with no real practical use; it's for demonstration purposes only.

Listing 8.1. EXPENSES.C demonstrates the use of an array.

```
/* EXPENSES.C - Demonstrates use of an array */

#include <stdio.h>

/* Declare an array to hold expenses, and a counter variable */

float expenses[13];
int count;

main()
{
/* Input data from keyboard into array */

for (count = 1; count < 13; count++)
{
printf("Enter expenses for month %d: ", count);
scanf("%f", &expenses[count]);
}

/* Print array contents */

for (count = 1; count < 13; count++)
{
printf("Month %d = $%.2f\n", count, expenses[count]);
}
return 0;
}
```

```
Enter expenses for month 1: 100
Enter expenses for month 2: 200.12
Enter expenses for month 3: 150.50
Enter expenses for month 4: 300
Enter expenses for month 5: 100.50
Enter expenses for month 6: 34.25
Enter expenses for month 7: 45.75
Enter expenses for month 8: 195.00
Enter expenses for month 9: 123.45
Enter expenses for month 10: 111.11
Enter expenses for month 11: 222.20
Enter expenses for month 12: 120.00
Month 1 = $100.00
Month 2 = $200.12
Month 3 = $150.50
Month 4 = $300.00
Month 5 = $100.50
Month 6 = $34.25
Month 7 = $45.75
Month 8 = $195.00
Month 9 = $123.45
Month 10 = $111.11
Month 11 = $222.20
Month 12 = $120.00
```

ANAALYSIS: When you run EXPENSES.C, the program prompts you to enter expenses for months 1 through 12. The values you enter are stored in an array. You must enter a value for each month. After the 12th value is entered, the array contents are displayed on-screen.

The flow of the program is similar to listings you've seen before. Line 1 starts with a comment that describes what the program does. Notice that the name of the program, EXPENSES.C, is included. When the name of the program is included in a comment, you know which program you're viewing. This is helpful when you're reviewing printouts of a listing. Line 5 contains an additional comment explaining the variables that are being declared. In line 7, an array of 13 elements is declared. In this program, only 12 elements are needed, one for each month, but 13 have been declared. The for loop in lines 14 through 18 ignores element 0. This lets the program use elements 1 through 12, which relate directly to the 12 months. Going back to line 8, a variable, count, is declared and is used throughout the program as a counter and an array index. The program's main() function begins on line 10. As stated earlier, this program uses a for loop to print a message and accept a value for each of the 12 months. Notice that in line 17, the scanf() function uses an array element. In line 7, the expenses array was declared as float, so %f is used. The address-of operator (&) also is placed before the array element, just as if it were a regular type float variable and not an array element. Lines 22 through 25 contain a second for loop that prints the values just entered. An additional formatting command has been added to the printf() function so that the expenses values print in a more orderly fashion. For now, know that %.2f prints a floating number with two digits to the right of the decimal. Additional formatting commands are covered in more detail on Day 14, "Working with the Screen, Printer, and Keyboard."

DON'T forget that array subscripts start at element 0.
DO use arrays instead of creating several variables that store the same thing.
For example, if you want to store total sales for each month of the year, create an array with 12 elements to hold sales rather than creating a sales variable for each month.

Multidimensional Arrays

A multidimensional array has more than one subscript. A two-dimensional array has two subscripts, a three-dimensional array has three subscripts, and so on. There is no limit to the number of dimensions a C array can have. (There *is* a limit on total array size, as discussed later in this chapter.)

For example, you might write a program that plays checkers. The checkerboard contains 64 squares arranged in eight rows and eight columns. Your program could represent the board as a two-dimensional array, as follows:

```
int checker[8][8];
```

The resulting array has 64 elements: checker[0][0], checker[0][1], checker[0][2]...checker[7][6], checker[7][7].

Similarly, a three-dimensional array could be thought of as a cube. Four-dimensional arrays (and higher) are probably best left to your imagination. All arrays, no matter how many dimensions they have, are stored sequentially in memory. More detail on array storage is presented on Day 15, "Pointers: Beyond the Basics."

Naming and Declaring Arrays

The rules for assigning names to arrays are the same as for variable names, covered on Day 3, "Storing Data: Variables and Constants." An array name must be unique. It can't be used for another array or for any other identifier (variable, constant, and so on). As you have probably realized, array declarations follow the same form as declarations of nonarray variables, except that the number of elements in the array must be enclosed in square brackets immediately following the array name.

When you declare an array, you can specify the number of elements with a literal constant (as was done in the earlier examples) or with a symbolic constant created with the #define directive.

Thus, the following:

```
#define MONTHS 12
int array[MONTHS];
```

is equivalent to this statement:

```
int array[12];
```

With most compilers, however, you can't declare an array's elements with a symbolic con-stant created with the const keyword:

```
const int MONTHS = 12;
int array[MONTHS];        /* Wrong! */
```

Listing 8.2, GRADES.C, is another program demonstrating the use of a single-dimensional array. GRADES.C uses an array to store 10 grades.

Listing 8.2. GRADES.C stores 10 grades in an array.

```
/*GRADES.C - Sample program with array */
/* Get 10 grades and then average them */

#include <stdio.h>

#define MAX_GRADE 100
#define STUDENTS 10

int grades[STUDENTS];

int idx;
int total = 0; /* used for average */

main()
{
for( idx=0;idx< STUDENTS;idx++)
{
printf( "Enter Person %d's grade: ", idx +1);
scanf( "%d", &grades[idx] );
20:
21: while ( grades[idx] > MAX_GRADE )
22: {
23: printf( "\nThe highest grade possible is %d",
24 MAX_GRADE );
25: printf( "\nEnter correct grade: " );
26: scanf( "%d", &grades[idx] );
27: }
28:
29: total += grades[idx];
30: }
31:
32: printf( "\n\nThe average score is %d\n", ( total / STUDENTS) );
33:
34: return (0);
35: }
```

```
Enter Person 1's grade: 95
Enter Person 2's grade: 100
Enter Person 3's grade: 60
Enter Person 4's grade: 105
The highest grade possible is 100
Enter correct grade: 100
Enter Person 5's grade: 25
Enter Person 6's grade: 0
Enter Person 7's grade: 85
Enter Person 8's grade: 85
Enter Person 9's grade: 95
Enter Person 10's grade: 85
The average score is 73
```

ANALYSIS: Like EXPENSES.C, this listing prompts the user for input. It prompts for 10 people's grades. Instead of printing each grade, it prints the average score. As you learned earlier, arrays are named like regular variables. On line 9, the array for this program is named grades. It should be safe to assume that this array holds grades. On lines 6 and 7, two constants, MAX_GRADE and STUDENTS, are defined. These constants can be changed easily. Knowing that STUDENTS is defined as 10, you then know that the grades array has 10 elements. Two other variables are declared, idx and total. An abbreviation of index, idx is used as a counter and array subscript. A running total of all grades is kept in total. The heart of this program is the for loop in lines 16 through 30. The for statement initializes idx to 0, the first subscript for an array. It then loops as long as idx is less than the number of students. Each time it loops, it increments idx by 1. For each loop, the program prompts for a person's grade (lines 18 and 19). Notice that in line 18, 1 is added to idx in order to count the people from 1 to 10 instead of from 0 to 9. Because arrays start with subscript 0, the first grade is put in grade[0]. Instead of confusing users by asking for Person 0's grade, they are asked for Person 1's grade. Lines 21 through 27 contain a while loop nested within the for loop. This is an edit check that ensures that the grade isn't higher than the maximum grade, MAX_GRADE. Users are prompted to enter a correct grade if they enter a grade that is too high. You should check program data whenever you can. Line 29 adds the entered grade to a total counter. In line 32, this total is used to print the average score (total/STUDENTS).

DO use #define statements to create constants that can be used when declaring arrays. Then you can easily change the number of elements in the array. In GRADES.C, for example, you could change the number of students in the #define, and you wouldn't have to make any other changes in the program.
DO avoid multidimensional arrays with more than three dimensions.
Remember, multidimensional arrays can get very big very quickly.

Initializing Arrays

You can initialize all or part of an array when you first declare it. Follow the array declaration with an equal sign and a list of values enclosed in braces and separated by commas. The listed values are assigned in order to array elements starting at number 0. For example, the following code assigns the value 100 to array[0], 200 to array[1], 300 to array[2], and 400 to array[3]:

```
int array[4] = { 100, 200, 300, 400 };
```

If you omit the array size, the compiler creates an array just large enough to hold the initialization values. Thus, the following statement would have exactly the same effect as the previous array declaration statement:

```
int array[] = { 100, 200, 300, 400 };
```

You can, however, include too few initialization values, as in this example:

```
int array[10] = { 1, 2, 3 };
```

If you don't explicitly initialize an array element, you can't be sure what value it holds when the program runs. If you include too many initializers (more initializers than array elements), the compiler detects an error.

Array Element Storage

As you might remember from Day 8, the elements of an array are stored in sequential memory locations with the first element in the lowest address. Subsequent array elements (those with an index greater than 0) are stored in higher addresses. How much higher depends on the array's data type (char, int, float, and so forth).

Take an array of type int. As you learned earlier, "Storing Data: Variables and Constants," a single int variable can occupy two bytes of memory. Each array element is therefore located two bytes above the preceding element, and the address of each array element is two higher than the address of the preceding element. A type float, on the other hand, can occupy four bytes. In an array of type float, each array element is located four bytes above the preceding element, and the address of each array element is four higher than the address of the preceding element.

why the following relationships are true:

Figure 9.7

```
1: x == 1000
2: &x[0] == 1000
3: &x[1] = 1002
4: expenses == 1250
5: &expenses[0] == 1250
6: &expenses[1] == 1254
```

x without the array brackets is the address of the first element (x[0]). You can also see that x[0] is at the address of 1000. Line 2 shows this too. It can be read like this: "The address of the first element of the array x is equal to 1000." Line 3 shows that the address of the second element (subscripted as 1 in an array) is 1002. Again, Figure 9.7 can confirm this. Lines 4, 5, and 6 are virtually identical to 1, 2, and 3, respectively. They vary in the difference between the addresses of the two array elements. In the type int array x, the difference is two bytes, and in the type float array, expenses, the difference is four bytes.

How do you access these successive array elements using a pointer? You can see from these examples that a pointer must be increased by 2 to access successive elements of a type int array, and by 4 to access successive elements of a type float array. You can generalize and say that to access successive elements of an array of a particular data type, a pointer must be increased by sizeof(*datatype*). Remember that the sizeof() operator returns the size in bytes of a C data type.

Listing 9.2 illustrates the relationship between addresses and the elements of different type arrays by declaring arrays of type int, float, and double and by displaying the addresses of successive elements.

Listing 9.2. Displaying the addresses of successive array elements.

```
/* Demonstrates the relationship between addresses and */
/* elements of arrays of different data types. */

#include <stdio.h>
```

```
5:
6: /* Declare three arrays and a counter variable. */
7:
8: int i[10], x;
9: float f[10];
10: double d[10];
11:
12: main()
13: {
14: /* Print the table heading */
15:
16: printf("\t\tInteger\t\tFloat\t\tDouble");
17:
18: printf("\n=================================");
19: printf("======================");
20:
21: /* Print the addresses of each array element. */
22:
23: for (x = 0; x < 10; x++)
24: printf("\nElement %d:\t%ld\t\t%ld\t\t%ld", x, &i[x],
25: &f[x], &d[x]);
26:
27: printf("\n=================================");
28: printf("======================\n");
29:
30: return 0;
31: }
```

```
Integer Float Double
=============================================
Element 0: 1392 1414 1454
Element 1: 1394 1418 1462
Element 2: 1396 1422 1470
Element 3: 1398 1426 1478
Element 4: 1400 1430 1486
Element 5: 1402 1434 1494
Element 6: 1404 1438 1502
Element 7: 1406 1442 1510
Element 8: 1408 1446 1518
Element 9: 1410 1450 1526
=============================================
```

ANALYSIS: The exact addresses that your system displays might be different from these, but the relationships are the same. In this output, there are two bytes between int elements, four bytes between float elements, and eight bytes between double elements. (Note: Some machines use different sizes for variable types. If your machine differs, the preceding output might have different-size gaps; however, they will be consistent gaps.) This listing takes advantage of the escape characters discussed on Day 7, "Fundamentals of Input and Output." The printf() calls in lines 16 and 24 use the tab escape character (\t) to help format the table by aligning the columns. Looking more closely at Listing 9.2, you can see that three arrays are created in lines 8, 9, and 10. Line 8 declares array i of type int, line 9 declares array f of type float, and line 10 declares array d of type double. Line 16 prints the column headers for the table that will be displayed. Lines 18 and 19, along with lines 27 and 28, print dashed lines across the top and bottom of the table data. This is a nice touch for a report. Lines 23, 24, and 25 are a for loop that prints each of the table's rows. The number of the element x is printed first. This is followed by the address of the element in each of the three arrays.

What Is a Pointer?

To understand pointers, you need a basic knowledge of how your computer stores infor-mation in memory. The following is a somewhat simplified account of PC memory storage.

Your Computer's Memory

A PC's RAM consists of many thousands of sequential storage locations, and each location is identified by a unique address. The memory addresses in a given computer range from 0 to a maximum value that depends on the amount of memory installed.

When you're using your computer, the operating system uses some of the system's memory. When you're running a program, the program's code (the machine-language instructions for the program's various tasks) and data (the information the program is using) also use some of the system's memory. This section examines the memory storage for program data.

When you declare a variable in a C program, the compiler sets aside a memory location with a unique address to store that variable. The compiler associates that address with the variable's name. When your program uses the variable name, it automatically accesses the proper memory location. The location's address is used, but it is hidden from you, and you need not be concerned with it.

Creating a Pointer

You should note that the address of the variable rate (or any other variable) is a number and can be treated like any other number in C. If you know a variable's address, you can create a second variable in which to store the address of the first. The first step is to declare a variable to hold the address of rate. Give it the name p_rate, for example. At first, p_rate is uninitialized. Storage has been allocated for p_rate, but its value is undetermined.The next step is to store the address of the variable rate in the variable p_rate. Because p_rate now contains the address of rate, it indicates the location where rate is stored in memory. In C parlance, p_rate points to rate, or is a pointer to rate.

To summarize, a pointer is a variable that contains the address of another variable. Now you can get down to the details of using pointers in your C programs.

Pointers and Simple Variables

In the example just given, a pointer variable pointed to a simple (that is, nonarray) variable. This section shows you how to create and use pointers to simple variables.

Declaring Pointers

A pointer is a numeric variable and, like all variables, must be declared before it can be used. Pointer variable names follow the same rules as other variables and must be unique. This chapter uses the convention that a pointer to the variable name is called p_name. This isn't necessary, however; you can name pointers anything you want (within C's naming rules).

A pointer declaration takes the following form:

typename **ptrname*;

typename is any of C's variable types and indicates the type of the variable that the pointer points to. The asterisk (*) is the indirection operator, and it indicates that *ptrname* is a pointer to type *typename* and not a variable of type *typename*. Pointers can be declared along with nonpointer variables. Here are some more examples:

```
char *ch1, *ch2;        /* ch1 and ch2 both are pointers to type char */
float *value, percent;   /* value is a pointer to type float, and
                         /* percent is an ordinary float variable */
```

❑**:** The * symbol is used as both the indirection operator and the multiplication operator. Don't worry about the compiler's becoming confused.The context in which * is used always provides enough information so that the compiler can figure out whether you mean indirection or multiplication.

Initializing Pointers

Now that you've declared a pointer, what can you do with it? You can't do anything with it until you make it point to something. Like regular variables, uninitialized pointers can be used, but the results are unpredictable and potentially disastrous. Until a pointer holds the address of a variable, it isn't useful. The address doesn't get stored in the pointer by magic; your program must put it there by using the address-of operator, the ampersand (&). When placed before the name of a variable, the address-of operator returns the address of the variable. Therefore, you initialize a pointer with a statement of the form
pointer = &*variable*;
The program statement to initialize the variable p_rate to point at the variable rate would be p_rate = &rate;   /* assign the address of rate to p_rate */
Before the initialization, p_rate didn't point to anything in particular. After the initialization, p_rate is a pointer to rate.

Using Pointers

Now that you know how to declare and initialize pointers, you're probably wondering how to use them. The indirection operator (*) comes into play again. When the * precedes the name of a pointer, it refers to the variable pointed to.

Let's continue with the previous example, in which the pointer p_rate has been initialized to point to the variable rate. If you write *p_rate, it refers to the variable rate. If you want to print the value of rate (which is 100 in the example), you could write
printf("%d", rate);
or this:
printf("%d", *p_rate);
In C, these two statements are equivalent. Accessing the contents of a variable by using the variable name is called *direct access*. Accessing the contents of a variable by using a pointer to the variable is called *indirect access* or *indirection*.
Pause a minute and think about this material. Pointers are an integral part of the C language, and it's essential that you understand them. Pointers have confused many people, so don't worry if you're feeling a bit puzzled. If you need to review, that's fine. Maybe the following summary can help. If you have a pointer named ptr that has been initialized to point to the variable var, the following are true:
*ptr and var both refer to the contents of var (that is, whatever value the program has stored there).
• ptr and &var refer to the address of var.

As you can see, a pointer name without the indirection operator accesses the pointer value itself, which is, of course, the address of the variable pointed to. Listing 9.1 demonstrates basic pointer use. You should enter, compile, and run this program.

Listing 9.1. Basic pointer use.

```
/* Demonstrates basic pointer use. */

#include <stdio.h>

/* Declare and initialize an int variable */

int var = 1;

/* Declare a pointer to int */

int *ptr;

main()
{
```

```
15:    /* Initialize ptr to point to var */
16:
17:    ptr = &var;
18:
19:    /* Access var directly and indirectly */
20:
21:    printf("\nDirect access, var = %d", var);
22:    printf("\nIndirect access, var = %d", *ptr);
23:
24:    /* Display the address of var two ways */
25:
26:    printf("\n\nThe address of var = %d", &var);
27:    printf("\nThe address of var = %d\n", ptr);
28:
29:    return 0;
30: }
```

```
Direct access, var = 1
Indirect access, var = 1
The address of var = 4264228
The address of var = 4264228
```

The address reported for var might not be 4264228 on your system.

ANALYSIS: In this listing, two variables are declared. In line 7, var is declared as an int and initialized to 1. In line 11, a pointer to a variable of type int is declared and named ptr. In line 17, the pointer ptr is assigned the address of var using the address-of operator (&). The rest of the program prints the values from these two variables to the screen. Line 21 prints the value of var, whereas line 22 prints the value stored in the location pointed to by ptr. In this program, this value is 1. Line 26 prints the address of var using the address-of operator. This is the same value printed by line 27 using the pointer variable, ptr.

DON'T use an uninitialized pointer. Results can be disastrous if you do.

Pointers and Variable Types

The previous discussion ignores the fact that different variable types occupy different amounts of memory. For the more common PC operating systems, an int takes two bytes, a float takes four bytes, and so on. Each individual byte of memory has its own address, so a multibyte variable actually occupies several addresses.

How, then, do pointers handle the addresses of multibyte variables? Here's how it works: The address of a variable is actually the address of the first (lowest) byte it occupies. This can be illustrated with an example that declares and initializes three variables:

```
int vint = 12252;
char vchar = 90;
float vfloat = 1200.156004;
```

Now let's declare and initialize pointers to these three variables:

```
int *p_vint;
char *p_vchar;
float *p_vfloat;
/* additional code goes here */
p_vint = &vint;
p_vchar = &vchar;
p_vfloat = &vfloat;
```

Each pointer is equal to the address of the first byte of the pointed-to variable. Thus, p_vint equals 1000, p_vchar equals 1003, and p_vfloat equals 1006. Remember, however, that each pointer was declared to point to a certain type of variable. The compiler knows that a pointer to type int points to the first of two bytes, a pointer to type float points to the first of four bytes, and so on. This is illustrated in Figure 9.6.

Pointers and Arrays

Pointers can be useful when you're working with simple variables, but they are more helpful with arrays. There is a special relationship between pointers and arrays in C. In fact, when you use the array subscript notation that you learned on Day 8, "Using Numeric Arrays," you're really using pointers without knowing it. The following sections explain how this works.

The Array Name as a Pointer

An array name without brackets is a pointer to the array's first element. Thus, if you've declared an array data[], data is the address of the first array element. "Wait a minute," you might be saying. "Don't you need the address-of operator to get an address?" Yes. You can also use the expression &data[0] to obtain the address of the array's first element. In C, the relationship (data == &data[0]) is true. You've seen that the name of an array is a pointer to the array. Remember that this is a pointer constant; it can't be changed and remains fixed for the duration of program execution. This makes sense: If you changed its value, it would point elsewhere and not to the array (which remains at a fixed location in memory). You can, however, declare a pointer variable and initialize it to point at the array. For example, the following code initializes the pointer variable p_array with the address of the first element of array[]:

```
int array[100], *p_array;
/* additional code goes here */
p_array = array;
```

Because p_array is a pointer variable, it can be modified to point elsewhere. Unlike array, p_array isn't locked into pointing at the first element of array[]. For example, it could be pointed at other elements of array[]. How would you do this? First, you need to look at how array elements are stored in memory.

Array Element Storage

As you might remember from Day 8, the elements of an array are stored in sequential memory locations with the first element in the lowest address. Subsequent array elements (those with an index greater than 0) are stored in higher addresses. How much higher depends on the array's data type (char, int, float, and so forth).

Take an array of type int. As you learned earlier, "Storing Data: Variables and Constants," a single int variable can occupy two bytes of memory. Each array element is therefore located two bytes above the preceding element, and the address of each array element is two higher than the address of the preceding element. A type float, on the other hand, can occupy four bytes. In an array of type float, each array element is located four bytes above the preceding element, and the address of each array element is four higher than the address of the preceding element.

why the following relationships are true:

Figure 9.7

```
1: x == 1000
2: &x[0] == 1000
3: &x[1] = 1002
4: expenses == 1250
5: &expenses[0] == 1250
6: &expenses[1] == 1254
```

x without the array brackets is the address of the first element (x[0]). You can also see that x[0] is at the address of 1000. Line 2 shows this too. It can be read like this: "The address of the first element of the array x is equal to 1000." Line 3 shows that the address of the second element (subscripted as 1 in an array) is 1002. Again, Figure 9.7 can confirm this. Lines 4, 5, and 6 are virtually identical to 1, 2, and 3, respectively. They vary in the difference between the addresses of the two array elements. In the type int array x, the difference is two bytes, and in the type float array, expenses, the difference is four bytes.

How do you access these successive array elements using a pointer? You can see from these examples that a pointer must be increased by 2 to access successive elements of a type int array, and by 4 to access successive elements of a type float array. You can generalize and say that to access successive elements of an array of a particular data type, a pointer must be increased by sizeof(*datatype*). Remember that the sizeof() operator returns the size in bytes of a C data type. Listing 9.2 illustrates the relationship between addresses and the elements of different type arrays by declaring arrays of type int, float, and double and by displaying the addresses of successive
elements.

Listing 9.2. Displaying the addresses of successive array elements.

```
/* Demonstrates the relationship between addresses and */
/* elements of arrays of different data types. */

#include <stdio.h>

/* Declare three arrays and a counter variable. */

int i[10], x;
float f[10];
double d[10];

main()
{
/* Print the table heading */

printf("\t\tInteger\t\tFloat\t\tDouble");

printf("\n================================");
printf("=====================");

/* Print the addresses of each array element. */

for (x = 0; x < 10; x++)
printf("\nElement %d:\t%ld\t\t%ld\t\t%ld", x, &i[x],
&f[x], &d[x]);

printf("\n================================");
printf("=====================\n");

return 0;
}
```

```
Integer Float Double
=============================================
Element 0: 1392 1414 1454
Element 1: 1394 1418 1462
Element 2: 1396 1422 1470
Element 3: 1398 1426 1478
Element 4: 1400 1430 1486
Element 5: 1402 1434 1494
Element 6: 1404 1438 1502
Element 7: 1406 1442 1510
Element 8: 1408 1446 1518
Element 9: 1410 1450 1526
=============================================
```

ANALYSIS: The exact addresses that your system displays might be different from these, but the relationships are the same. In this output, there are two bytes between int elements, four bytes between float elements, and eight bytes between double elements. (Note: Some machines use different sizes for variable types. If your machine differs, the preceding output might have different-size gaps; however, they will be consistent gaps.) This listing takes advantage of the escape characters discussed on Day 7, "Fundamentals of Input and Output." The printf() calls in lines 16 and 24 use the tab escape character (\t) to help format the table by aligning the columns. Looking more closely at Listing 9.2, you can see that three arrays are created in lines 8, 9, and 10. Line 8 declares array i of type int, line 9 declares array f of type float, and line 10 declares array d of type double. Line 16 prints the column headers for the table that will be displayed. Lines 18 and 19, along with lines 27 and 28, print dashed lines across the top and bottom of the table data. This is a nice touch for a report.
Lines 23, 24, and 25 are a for loop that prints each of the table's rows. The number of the element x is printed first. This is followed by the address of the element in each of the three arrays.

Passing Pointers to Functions

The default method of passing an argument to a function is by value. *Passing by value* means that the function is passed a copy of the argument's value. This method has three steps:

1. The argument expression is evaluated.
2. The result is copied onto the *stack,* a temporary storage area in memory.
3. The function retrieves the argument's value from the stack.

Figure 18.1 illustrates passing an argument by value. In this case, the argument is a simple type int variable, but the principle is the same for other variable types and more complex expressions. When a variable is passed to a function by value, the function has access to the variable's value but not to the original copy of the variable. As a result, the code in the function can't modify the original variable. This is the main reason why passing by value is the default method of passing arguments: Data outside a function is protected from inadvertent modification.
Passing arguments by value is possible with the basic data types (char, int, long, float, and double) and structures. There is another way to pass an argument to a function, however: by passing a pointer to the argument variable rather than the value of the variable itself. This method of passing an argument is called *passing by reference*.
As you learned in, "Understanding Pointers," passing by reference is the only way to pass an array to a function; passing an array by value is not possible. With other data types, however, you can use either method. If your program uses large structures, passing them by value might cause your program to run out of stack space. Aside from this consideration, passing an argument by reference instead of by value offers an advantage as well as a disadvantage:

- The advantage of passing by reference is that the function can modify the value of the argument variable.
- The disadvantage of passing by reference is that the function can modify the value of the argument variable.

"What?" you might be saying. "An advantage that's also a disadvantage?" Yes. It all depends on the specific situation. If your program requires that a function modify an argument variable, passing by reference is an advantage. If there is no such need, it is a disadvantage because of the possibility of inadvertent modifications.
You might be wondering why you don't use the function's return value to modify the argument variable. You can do this, of course, as shown in the following example:

```
x = half(x);
int half(int y)
{
return y/2;
}
```

Remember, however, that a function can return only a single value. By passing one or more arguments by reference, you allow a function to "return" more than one value to the calling program.

Listing 18.1 demonstrates passing by reference and the default passing by value. Its output clearly shows that a variable passed by value can't be changed by the function, whereas a variable passed by reference can be changed. Of course, a function doesn't need to modify a variable passed by reference. In such as case, there's no reason to pass by reference.

Listing 18.1. Passing by value and passing by reference.

```
1: /* Passing arguments by value and by reference. */
2:
3: #include <stdio.h>
4:
5: void by_value(int a, int b, int c);
6: void by_ref(int *a, int *b, int *c);
7:
8: main()
9: {
10: int x = 2, y = 4, z = 6;
11:
12: printf("\nBefore calling by_value(), x = %d, y = %d, z = %d.",
13: x, y, z);
14:
15: by_value(x, y, z);
16:
17: printf("\nAfter calling by_value(), x = %d, y = %d, z = %d.",
18: x, y, z);
19:
20: by_ref(&x, &y, &z);
21: printf("\nAfter calling by_ref(), x = %d, y = %d, z = %d.\n",
22: x, y, z);
23: return(0);
24: }
25:
26: void by_value(int a, int b, int c)
27: {
28: a = 0;
29: b = 0;
30: c = 0;
31: }
32:
33: void by_ref(int *a, int *b, int *c)
34: {
35: *a = 0;
36: *b = 0;
37: *c = 0;
38: }
```

```
Before calling by_value(), x = 2, y = 4, z = 6.
After calling by_value(), x = 2, y = 4, z = 6.
After calling by_ref(), x = 0, y = 0, z = 0.
```

ANALYSIS: This program demonstrates the difference between passing variables by value and passing them by reference. Lines 5 and 6 contain prototypes for the two functions called in the program. Notice that line 5 describes three arguments of type int for the by_value() function, but by_ref() differs on line 6 because it requires three pointers to type int variables as arguments. The function headers for these two functions on lines 26 and 33 follow the same format as the prototypes. The bodies of the two functions are similar, but not the same.

Both functions assign 0 to the three variables passed to them. In the by_value() function, 0 is assigned directly to the variables. In the by_ref() function, pointers are used, so the variables must be dereferenced. Each function is called once by main(). First, the three variables to be passed are assigned values other than 0 on line 10. Line 12 prints these values to the screen. Line 15 calls the first of the two functions, by_value(). Line 17 prints the three variables again. Notice that they are not changed. The by_value() function received the variables by value and therefore couldn't change their original content. Line 20 calls by_ref(), and line 22 prints the values again. This time, the values have all changed to 0. Passing the variables by reference gave by_ref() access to the actual contents of the variables. You can write a function that receives some arguments by reference and others by value. Just remember to keep them straight inside the function, using the indirection operator (*) to dereference arguments passed by reference.

DON'T pass large amounts of data by value if it isn't necessary. You could run out of stack space.

DO pass variables by value if you don't want the original value altered.
DON'T forget that a variable passed by reference should be a pointer. Also,
use the indirection operator to dereference the variable in the function.

Type void Pointers
You've seen the void keyword used to specify that a function either doesn't take arguments or doesn't return a value. The void keyword can also be used to create a generic pointer--a pointer that can point to any type of data object. For example, the statement
void *x;
declares x as a generic pointer. x points to something; you just haven't yet specified what. The most common use for type void pointers is in declaring function parameters. You might want to create a function that can handle different types of arguments. You can pass it a type int one time, a type float the next time, and so on. By declaring that the function takes a void pointer as an argument, you don't restrict it to accepting only a single data type. If you declare the function to take a void pointer as an argument, you can pass the function a pointer to anything.
Here's a simple example: You want a function that accepts a numeric variable as an argument and divides it by 2, returning the answer in the argument variable. Thus, if the variable x holds the value 4, after a call to half(x) the variable x is equal to 2. Because you want to modify the argument, you pass it by reference. Because you want to use the function with any of C's numeric data types, you declare the function to take a void pointer:

void half(void *x);

Now you can call the function, passing it any pointer as an argument. There's one more thing you need, however. Although you can pass a void pointer without knowing what data type it points to, you can't dereference the pointer. Before the code in the function can do anything with the pointer, it must know the data type. You do this with a *typecast,* which is nothing more than a way of telling the program to treat this void pointer as a pointer to type. If x is a void pointer, you typecast it as follows:
(type *)x
Here, type is the appropriate data type. To tell the program that x is a pointer to type int, write
(int *)x
To dereference the pointer--that is, to access the int that x points to--write
*(int *)x
Typecasts are covered in more detail in, "Working with Memory." Getting back to the original topic (passing a void pointer to a function), you can see that, to use the pointer, the function must know the data type to which it points. In the case of the function you're writing to divide its argument by two, there are four possibilities for type: int, long, float, and double. In addition to the void pointer to the variable to be divided by two, you must tell the function the type of variable to which the void pointer points. You can modify the function definition as follows:
void half(void *x, char type);
Based on the argument type, the function casts the void pointer x to the appropriate type. Then the pointer can be dereferenced, and the value of the pointed-to variable can be used. The final version of the half() function is shown in Listing 18.2.

Working with Strings

Characters and Strings

A *character* is a single letter, numeral, punctuation mark, or other such symbol. A *string* is any sequence of characters. Strings are used to hold text data, which is comprised of letters,numerals, punctuation marks, and other symbols. Clearly, characters and strings are extremely useful in many programming applications. Today you will learn

- How to use C's char data type to hold single characters
- How to create arrays of type char to hold multiple-character strings
- How to initialize characters and strings
- How to use pointers with strings
- How to print and input characters and strings

The char Data Type

C uses the char data type to hold characters. You saw in, "Storing Data: Variables and Constants," that char is one of C's numeric integer data types. If char is a numeric type, how can it be used to hold characters?
The answer lies in how C stores characters. Your computer's memory stores all data in numeric form. There is no direct way to store characters. However, a numeric code exists for each character. This is called the *ASCII code* or the *ASCII character set*. (*ASCII* stands for American Standard Code for Information Interchange.) The code assigns values between 0 and 255 for upper- and lowercase letters, numeric digits, punctuation marks, and other symbols. The ASCII character set is listed in Appendix A.

For example, 97 is the ASCII code for the letter a. When you store the character a in a type char variable, you're really storing the value 97. Because the allowable numeric range for type char matches the standard ASCII character set, char is ideally suited for storing characters.
At this point, you might be a bit puzzled. If C stores characters as numbers, how does your program know whether a given type char variable is a character or a number? As you'll learn later, declaring a variable as type char is not enough; you must do something else with the variable:
- If a char variable is used somewhere in a C program where a character is expected, it is interpreted as a character.
- If a char variable is used somewhere in a C program where a number is expected, it is interpreted as a number.

This gives you some understanding of how C uses a numeric data type to store character data. Now you can go on to the details.

Using Character Variables

Like other variables, you must declare chars before using them, and you can initialize them at the time of declaration. Here are some examples:

```
char a, b, c;       /* Declare three uninitialized char variables */
char code = `x';    /* Declare the char variable named code */
                    /* and store the character x there */
code = `!';         /* Store ! in the variable named code */
```

To create literal character constants, you enclose a single character in single quotation marks. The compiler automatical y translates literal character constants into the corresponding ASCII codes, and the numeric code value is assigned to the variable.
You can create symbolic character constants by using either the #define directive or the const keyword:

```
#define EX `x'
char code = EX;     /* Sets code equal to `x' */
const char A = `Z';
```

Now that you know how to declare and initialize character variables, it's time for a demonstration. Listing 10.1 illustrates the numeric nature of character storage using the printf() function you learned earlier, "Fundamentals of Input and Output." The function printf() can be used to print both characters and numbers. The format string %c instructs printf() to print a character, whereas %d instructs it to print a decimal integer. Listing 10.1 initializes two type char variables and prints each one, first as a character, and then as a number.

Listing 10.1. The numeric nature of type char variables.

```
/* Demonstrates the numeric nature of char variables */

#include <stdio.h>

/* Declare and initialize two char variables */

char c1 = `a';
char c2 = 90;

main()
{
    /* Print variable c1 as a character, then as a number */

    printf("\nAs a character, variable c1 is %c", c1);
    printf("\nAs a number, variable c1 is %d", c1);

    /* Do the same for variable c2 */

    printf("\nAs a character, variable c2 is %c", c2);
    printf("\nAs a number, variable c2 is %d\n", c2);

    return 0;
}
```

```
As a character, variable c1 is a
As a number, variable c1 is 97
As a character, variable c2 is Z
As a number, variable c2 is 90
```

ANALYSIS: You learned on Day 3 that the allowable range for a variable of type char goes only to 127, whereas the ASCII codes go to 255. The ASCII codes are actually divided into two parts. The standard ASCII codes go only to 127; this range includes all letters, numbers, punctuation marks, and other keyboard symbols. The codes from 128 to 255 are the extended ASCII codes and represent special characters such as foreign letters and graphics symbols (see Appendix A for a complete list). Thus, for standard text data, you can use type char variables; if you want to print the extended ASCII characters, you must use an unsigned char.

Listing 10.2 prints some of the extended ASCII characters.

Listing 10.2. Printing extended ASCII characters.

```
/* Demonstrates printing extended ASCII characters */

#include <stdio.h>

unsigned char x;    /* Must be unsigned for extended ASCII */
```

```
6:
7:  main()
8:  {
9:      /* Print extended ASCII characters 180 through 203 */
10:
11:      for (x = 180; x < 204; x++)
12:      {
13:          printf("ASCII code %d is character %c\n", x, x);
14:      }
15:
16:          return 0;
17: }
```

```
ASCII code 180 is character ¥
ASCII code 181 is character µ
ASCII code 182 is character [partialdiff]
ASCII code 183 is character [Sigma]
ASCII code 184 is character [Pi]
ASCII code 185 is character [pi]
ASCII code 186 is character [integral]
ASCII code 187 is character ª
ASCII code 188 is character º
ASCII code 189 is character [Omega]
ASCII code 190 is character æ
ASCII code 191 is character ø
ASCII code 192 is character ¿
ASCII code 193 is character ¡
ASCII code 194 is character ¬
ASCII code 195 is character [radical]
ASCII code 196 is character [florin]
ASCII code 197 is character ~
ASCII code 198 is character [Delta]
ASCII code 199 is character «
ASCII code 200 is character »
ASCII code 201 is character ...
ASCII code 202 is character g
ASCII code 203 is character À
```

ANALYSIS: Looking at this program, you see that line 5 declares an unsigned character variable, x. This gives a range of 0 to 255. As with other numeric data types, you must not initialize a char variable to a value outside the allowed range, or you might get unexpected results. In line 11, x is not initialized outside the range; instead, it is initialized to 180. In the for statement, x is incremented by 1 until it reaches 204. Each time x is incremented, line 13 prints the value of x and the character value of x. Remember that %c prints the character, or ASCII, value of x.
DO use %c to print the character value of a number.
DON'T use double quotations when initializing a character variable.
DO use single quotations when initializing a variable.
DON'T try to put extended ASCII character values into a signed char variable.
DO look at the ASCII chart in Appendix A to see the interesting characters that can be printed.

❑**:** Some computer systems might use a different character set; however, most use the same values for 0 to 127.

Strings

The C library has three functions for copying strings. Because of the way C handles strings, you can't simply assign one string to another, as you can in some other computer languages. You must copy the source string from its location in memory to the memory location of the destination string. The string-copying functions are strcpy(), strncpy(), and strdup(). Al of the string-copying functions require the header file STRING.H.

The strcpy() Function

The library function strcpy() copies an entire string to another memory location. Its prototype is as follows:
char *strcpy(char *destination, char *source);
The function strcpy() copies the string (including the terminating null character \0) pointed to by source to the location pointed to by destination. The return value is a pointer to the new string, destination. When using strcpy(), you must first allocate storage space for the destination string. The function has no way of knowing whether destination points to allocated space. If space hasn't been allocated, the function overwrites strlen(source) bytes of memory, starting at destination; this can cause unpredictable problems. The use of strcpy() is illustrated in Listing 17.2.

NOTE: When a program uses malloc() to allocate memory, as Listing 17.2 does, good programming practice requires the use of the free() function to free up the memory when the program is finished with it.

Listing 17.2. Before using strcpy(), you must allocate storage space for the destination string.

```
/* Demonstrates strcpy(). */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char source[] = "The source string.";

main()
{
    char dest1[80];
    char *dest2, *dest3;

    printf("\nsource: %s", source );

    /* Copy to dest1 is okay because dest1 points to */
    /* 80 bytes of allocated space. */

    strcpy(dest1, source);
    printf("\ndest1:  %s", dest1);

    /* To copy to dest2 you must allocate space. */

    dest2 = (char *)malloc(strlen(source) +1);
    strcpy(dest2, source);
    printf("\ndest2:  %s\n", dest2);

    /* Copying without allocating destination space is a no-no. */
    /* The following could cause serious problems. */

    /* strcpy(dest3, source); */
31:     return(0);
32: }
```

```
source: The source string.
dest1:  The source string.
dest2:  The source string.
```

ANALYSIS: This program demonstrates copying strings both to character arrays such as dest1 (declared on line 10) and to character pointers such as dest2 (declared along with dest3 on line 11). Line 13 prints the original source string. This string is then copied to dest1 with strcpy() on line 18. Line 24 copies source to dest2. Both dest1 and dest2 are printed to show that the function was successful. Notice that line 23 allocates the appropriate amount of space for dest2 with the malloc() function. If you copy a string to a character pointer that hasn't been allocated memory, you get unpredictable results.

String Length and Storage

You should remember from earlier chapters that, in C programs, a string is a sequence of characters, with its beginning indicated by a pointer and its end marked by the null character \0. At times, you need to know the length of a string (the number of characters between the start and the end of the string). This length is obtained with the library function strlen(). Its prototype, in STRING.H, is size_t strlen(char *str);
You might be puzzling over the size_t return type. This type is defined in STRING.H as unsigned, so the function strlen() returns an unsigned integer. The size_t type is used with many of the string functions. Just remember that it means unsigned. The argument passed to strlen is a pointer to the string of which you want to know the length. The function strlen() returns the number of characters between str and the next null character, not counting the null character. Listing 17.1 demonstrates strlen().

Listing 17.1. Using the strlen() function to determine the length of a string.

```
1: /* Using the strlen() function. */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: main()
7: {
8: size_t length;
9: char buf[80];
10:
11: while (1)

12: {
13: puts("\nEnter a line of text; a blank line terminates.");
14: gets(buf);
15:
16: length = strlen(buf);
17:
18: if (length != 0)
19: printf("\nThat line is %u characters long.", length);
20: else
21: break;
22: }
23: return(0);
24: }
```

```
Enter a line of text; a blank line terminates.
Just do it!
That line is 11 characters long.
Enter a line of text; a blank line terminates.
```

ANALYSIS:: This program does little more than demonstrate the use of strlen(). Lines 13 and 14 display a message and get a string called buf. Line 16 uses strlen() to assign the length of buf to the variable length. Line 18 checks whether the string was blank by checking for a length of 0. If the string wasn't blank, line 19 prints the string's size.

Concatenating Strings

If you're not familiar with the term *concatenation,* you might be asking, "What is it?" and "Is it legal?" Well, it means to join two strings--to tack one string onto the end of another--and, in most states, it is legal. The C standard library contains two string concatenation functions--strcat() and strncat()--both of which require the header file STRING.H.

The strcat() Function

The prototype of strcat() is
char *strcat(char *str1, char *str2);
The function appends a copy of str2 onto the end of str1, moving the terminating null character to the end of the new string. You must allocate enough space for str1 to hold the resulting string. The return value of strcat() is a pointer to str1. Listing 17.5 demonstrates strcat().

Listing 17.5. Using strcat() to concatenate strings.

```
/* The strcat() function. */

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

char str1[27] = "a";
char str2[2];

main()
{
    int n;

    /* Put a null character at the end of str2[]. */

    str2[1] = `\0';

    for (n = 98; n< 123; n++)
    {
        str2[0] = n;
        strcat(str1, str2);
        puts(str1);
    }
     return(0);
}
```

```
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
```

```
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

ANALYSIS: The ASCII codes for the letters b through z are 98 to 122. This program uses these ASCII codes in its demonstration of strcat(). The for loop on lines 17 through 22 assigns these values in turn to str2[0]. Because str2[1] is already the null character (line 15), the effect is to assign the strings "b", "c", and so on to str2. Each of these strings is concatenated with str1 (line 20), and then str1 is displayed on-screen (line 21).

Comparing Strings

Strings are compared to determine whether they are equal or unequal. If they are unequal, one string is "greater than" or "less than" the other. Determinations of "greater" and "less" are made with the ASCII codes of the characters. In the case of letters, this is equivalent to alphabetical order, with the one seemingly strange exception that all uppercase letters are "less than" the lowercase letters. This is true because the uppercase letters have ASCII codes 65 through 90 for A through Z, while lowercase a through z are represented by 97 through 122. Thus, "ZEBRA" would be considered to be less than "apple" by these C functions.
The ANSI C library contains functions for two types of string comparisons: comparing two entire strings, and comparing a certain number of characters in two strings.

Comparing Two Entire Strings

The function strcmp() compares two strings character by character. Its prototype is int strcmp(char *str1, char *str2);
The arguments str1 and str2 are pointers to the strings being compared. The function's return values are given in Table 17.1. Listing 17.7 demonstrates strcmp().

Table 17.1. The values returned by strcmp().

Return Value	Meaning
< 0	str1 is less than str2
0	str1 is equal to str2.
> 0	str1 is greater than str2

NOTE: On some UNIX systems, string comparison functions don't necessarily return -1 when the strings aren't the same. They will, however, always return a nonzero value for unequal strings.

Listing 17.7. Using strcmp() to compare strings.

```
1: /* The strcmp() function. */
2:
3: #include <stdio.h>
```

```
4: #include <string.h>
5:
6: main()
7: {
8: char str1[80], str2[80];
9: int x;
10:
11: while (1)
12: {
13:
14: /* Input two strings. */
15:
16: printf("\n\nInput the first string, a blank to exit: ");
17: gets(str1);
18:
19: if ( strlen(str1) == 0 )
20: break;
21:
22: printf("\nInput the second string: ");
23: gets(str2);
24:
25: /* Compare them and display the result. */
26:
27: x = strcmp(str1, str2);
28:
29: printf("\nstrcmp(%s,%s) returns %d", str1, str2, x);
30: }
31: return(0);
32: }
```

```
Input the first string, a blank to exit: First string

Input the second string: Second string

strcmp(First string,Second string) returns -1

Input the first string, a blank to exit: test string

Input the second string: test string

strcmp(test string,test string) returns 0

Input the first string, a blank to exit: zebra

Input the second string: aardvark

strcmp(zebra,aardvark) returns 1

Input the first string, a blank to exit:
```

NOTE: On some UNIX systems, string comparison functions don't necessarily return -1 when the strings aren't the same. They will, however, always return a nonzero value for unequal strings.

ANALYSIS: This program demonstrates strcmp(), prompting the user for two strings (lines 16, 17, 22, and 23) and displaying the result returned by strcmp() on line 29. Experiment with this program to get a feel for how strcmp() compares strings. Try entering two strings that are identical except for case, such as

Smith and SMITH. You'll see that strcmp() is case-sensitive, meaning that the program considers uppercase and lowercase letters to be different.

Displaying Messages with puts()

The puts() function can also be used to display text messages on-screen, but it can't display numeric variables. puts() takes a single string as its argument and displays it, automatically adding a newline at the end. For example, the statement

```
puts("Hello, world.");
```

performs the same action as

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

You can include escape sequences (including \n) in a string passed to puts(). They have the same effect as when they are used with printf() (see Table 7.1). Any program that uses puts() should include the header file STDIO.H.
❑ that STDIO.H should be included only once in a program.

DO use the puts() function instead of the printf() function whenever you want to print text but don't need to print any variables.
DON'T try to use conversion specifiers with the puts() statement.

The puts() Function

```
#include <stdio.h>
puts( string );
```

puts() is a function that copies a string to the standard output device, usually the display screen. When you use puts(), include the standard input/output header file (STDIO.H). puts() also appends a newline character to the end of the string that is printed. The for- mat string can contain escape sequences. Table 7.1 lists the most frequently used escape sequences.
The following are examples of calls to puts() and their output:
Example 1 Input
puts("This is printed with the puts() function!");
Example 1 Output
This is printed with the puts() function!
Example 2 Output
This prints on the first line.
This prints on the second line.
This prints on the third line.
If these were printf()s, all four lines would be on two lines!

Inputting Strings Using the gets() Function

The gets() function gets a string from the keyboard. When gets() is called, it reads all characters typed at the keyboard up to the first newline character (which you generate by pressing Enter). This function discards the newline, adds a null character, and gives the string to the calling program. The string is stored at the location indicated by a pointer to type char passed to gets(). A program that uses gets() must #include the file STDIO.H. Listing 10.5 presents an example.

Listing 10.5. Using gets() to input string data from the keyboard.

```
1:  /* Demonstrates using the gets() library function. */
2:
```

```
3:  #include <stdio.h>
4:
5:  /* Allocate a character array to hold input. */
6:
7:  char input[81];
8:
9:  main()
10: {
11:     puts("Enter some text, then press Enter: ");
12:     gets(input);
13:     printf("You entered: %s\n", input);
14:
15:     return 0;
16: }
```

```
Enter some text, then press Enter:

This is a test
You entered: This is a test
```

ANALYSIS: [endd] In this example, the argument to gets() is the expression input, which is the name of a type char array and therefore a pointer to the first array element. The array is declared with 81 elements in line 7. Because the maximum line length possible on most computer screens is 80 characters, this array size provides space for the longest possible input line (plus the null character that gets() adds at the end).
The gets() function has a return value, which was ignored in the previous example. gets() returns a pointer to type char with the address where the input string is stored. Yes, this is the same value that is passed to gets(), but having the value returned to the program in this way lets your program test for a blank line. Listing 10.6 shows how to do this.

Listing 10.6. Using the gets() return value to test for the input of a blank line.

```
/* Demonstrates using the gets() return value. */

#include <stdio.h>

/* Declare a character array to hold input, and a pointer. */

char input[81], *ptr;

main()
{
    /* Display instructions. */

    puts("Enter text a line at a time, then press Enter.");
    puts("Enter a blank line when done.");

    /* Loop as long as input is not a blank line. */

    while ( *(ptr = gets(input)) != NULL)
        printf("You entered %s\n", input);

    puts("Thank you and good-bye\n");

    return 0;
}
```

```
Enter text a line at a time, then press Enter.
Enter a blank line when done.
```

```
First string
You entered First string
Two
You entered Two
Bradley L. Jones
You entered Bradley L. Jones
Thank you and good-bye
```

ANALYSIS: Now you can see how the program works. If you enter a blank line (that is, if you simply press Enter) in response to line 18, the string (which contains 0 characters) is still stored with a null character at the end. Because the string has a length of 0, the null character is stored in the first position. This is the position pointed to by the return value of gets(), so if you test that position and find a null character, you know that a blank line was entered.

1. The gets() function accepts input from the keyboard until it reaches a newline character.
2. The input string, minus the newline and with a trailing null character, is stored in the memory location pointed to by input.
3. The address of the string (the same value as input) is returned to the pointer ptr.
4. An *assignment statement* is an expression that evaluates to the value of the variable on the left side of the assignment operator. Therefore, the entire expression ptr = gets(input) evaluates to the value of ptr. By enclosing this expression in parentheses and preceding it with the indirection operator (*), you obtain the value stored at the pointed-to address. This is, of course, the first character of the input string.
5. NULL is a symbolic constant defined in the header file STDIO.H. It has the value of the null character (0).
6. If the first character of the input string isn't the null character (if a blank line hasn't been entered), the comparison operator returns true, and the while loop executes. If the first character is the null character (if a blank line has been entered), the comparison operator returns false, and the while loop terminates.

When you use gets() or any other function that stores data using a pointer, be sure that the pointer points to allocated space. It's easy to make a mistake such as this:

```
char *ptr;
gets(ptr);
```

The pointer ptr has been declared but not initialized. It points somewhere, but you don't know where. The gets() function doesn't know this, so it simply goes ahead and stores the input string at the address contained in ptr. The string might overwrite something important, such as program code or the operating system. The compiler doesn't catch these kinds of mistakes, so you, the programmer, must be vigilant.

The gets() Function

```
#include <stdio.h>
char *gets(char *str);
```

The gets() function gets a string, str, from the standard input device, usually the keyboard.

The string consists of any characters entered until a newline character is read. At that point, a null is appended to the end of the string.

Then the gets() function returns a pointer to the string just read. If there is a problem getting the string, gets() returns null.

Example

```
/* gets() example */
#include <stdio.h>
char line[256];
void main()
{
  printf( "Enter a string:\n");
  gets( line );
  printf( "\nYou entered the following string:\n" );
  printf( "%s\n", line );
}
```

Inputting Strings Using the scanf() Function

The scanf() library function accepts numeric data input from the keyboard. This function can also input strings. Remember that scanf() uses a *format string* that tells it how to read the input. To read a string, include the specifier %s in scanf()'s format string. Like gets(), scanf() is passed a pointer to the string's storage location.
How does scanf() decide where the string begins and ends? The beginning is the first nonwhitespace character encountered. The end can be specified in one of two ways. If you use %s in the format string, the string runs up to (but not including) the next whitespace character (space, tab, or newline). If you use %ns (where n is an integer constant that specifies field width), scanf() inputs the next n characters or up to the next whitespace

character, whichever comes first.

You can read in multiple strings with scanf() by including more than one %s in the format string. For each %s in the format string, scanf() uses the preceding rules to find the requested number of strings in the input. For example:

```
scanf("%s%s%s", s1, s2, s3);
```

If in response to this statement you enter January February March, January is assigned to the string s1, February is assigned to s2, and March to s3.

What about using the field-width specifier? If you execute the statement

```
scanf("%3s%3s%3s", s1, s2, s3);
```

and in response you enter September, Sep is assigned to s1, tem is assigned to s2, and ber is assigned to s3.
What if you enter fewer or more strings than the scanf() function expects? If you enter fewer strings, scanf() continues to look for the missing strings, and the program doesn't continue until they're entered. For example, if in response to the statement

```
scanf("%s%s%s", s1, s2, s3);
```

you enter January February, the program sits and waits for the third string specified in the scanf() format string. If you enter more strings than requested, the unmatched strings remain pending (waiting in the keyboard buffer) and are read by any subsequent scanf() or other input statements. For example, if in response to the statements

```
scanf("%s%s", s1, s2);

scanf("%s", s3);
```

you enter January February March, the result is that January is assigned to the string s1, February is assigned to s2, and March is assigned to s3.

Module 6: Structures

Module Objective: The participants will understand

- How to organize complicated data, particularly in large 'C' programs.
- How structures are used.

Chapter 1: Structures
Structure declaration and definition
Structure members and initialization
Accessing structure members
Structure within Structure and initialization

Structures

Many programming tasks are simplified by the C data constructs cal ed *structures*. A structure is a data storage method designed by you, the programmer, to suit your programming needs exactly.

Simple Structures

A *structure* is a collection of one or more variables grouped under a single name for easy manipulation. The variables in a structure, unlike those in an array, can be of different variable types. A structure can contain any of C's data types, including arrays and other structures.
Each variable within a structure is cal ed a *member* of the structure. The next section shows a simple example.
You should start with simple structures. Note that the C language makes no distinction between simple and complex structures, but it's easier to explain structures in this way.

Defining and Declaring Structures

If you're writing a graphics program, your code needs to deal with the coordinates of points on the screen. Screen coordinates are written as an x value, giving the horizontal position, and a y value, giving the vertical position. You can define a structure named coord that contains both the x and y values of a screen location as follows:

```
struct coord {
   int x;
   int y;
};
```

The struct keyword, which identifies the beginning of a structure definition, must be followed immediately by the structure name, or *tag* (which follows the same rules as other C variable names). Within the braces following the structure name is a list of the structure's member variables. You must give a variable type and name for each member.
The preceding statements define a structure type named coord that contains two integer variables, x and y. They do not, however, actually create any instances of the structure coord. In other words, they don't *declare* (set aside storage for) any structures. There are two ways to declare structures. One is to follow the structure definition with a list of one or more variable names, as is done here:

```
struct coord {
   int x;
   int y;
} first, second;
```

These statements define the structure type coord and declare two structures, first and second, of type coord. first and second are each *instances* of type coord; first contains two integer members named x and y, and so does second.

This method of declaring structures combines the declaration with the definition. The second method is to declare structure variables at a different location in your source code from the definition. The following statements also declare two instances of type coord:

```
struct coord {
   int x;
   int y;
};
/* Additional code may go here */
struct coord first, second;
```

Accessing Structure Members

Individual structure members can be used like other variables of the same type. Structure members are accessed using the *structure member operator* (.), also called the *dot operator,*between the structure name and the member name. Thus, to have the structure named first refer to a screen location that has coordinates x=50, y=100, you could write

```
first.x = 50;
first.y = 100;
```

To display the screen locations stored in the structure second, you could write

```
printf("%d,%d", second.x, second.y);
```

At this point, you might be wondering what the advantage is of using structures rather than individual variables. One major advantage is that you can copy information between structures of the same type with a simple equation statement. Continuing with the preceding example, the statement

```
first = second;
```

is equivalent to this statement:

```
first.x = second.x;
first.y = second.y;
```

When your program uses complex structures with many members, this notation can be a great time-saver. Other advantages of structures will become apparent as you learn some advanced techniques. In general, you'll find structures to be useful whenever information of different variable types needs to be treated as a group. For example, in a mailing list database, each entry could be a structure, and each piece of information (name, address, city, and so on) could be a structure member.

The struct Keyword

struct *tag* {
structure_member(s);
   /* additional statements may go here */
} *instance*;

The struct keyword is used to declare structures. A structure is a collection of one or more variables (*structure_members*) that have been grouped under a single name for easy man- ipulation. The variables don't have to be of the same variable type, nor do they have to be simple variables. Structures also can hold arrays, pointers, and other structures.

The keyword struct identifies the beginning of a structure definition. It's followed by a tag that is the name given to the structure. Following the tag are the structure members, enclosed in braces. An *instance*, the actual declaration of a structure, can also be defined. If you define the structure without the instance, it's just a template that can be used later in a program to declare structures. Here is a template's format:

struct *tag* {
structure_member(s);
   /* additional statements may go here */
};

To use the template, you use the following format:

struct *tag instance*;

To use this format, you must have previously declared a structure with the given tag.

Example 1

```
/* Declare a structure template called SSN */
struct SSN {
   int first_three;
   char dash1;
   int second_two;
   char dash2;
   int last_four;
}
/* Use the structure template */
struct SSN customer_ssn;
```

Example 2

```
/* Declare a structure and instance together */
struct date {
   char month[2];
   char day[2];
   char year[4];
} current_date;
```

Example 3

```
/* Declare and initialize a structure */
struct time {
   int hours;
   int minutes;
   int seconds;
} time_of_birth = { 8, 45, 0 };
```

More-Complex Structures

Now that you have been introduced to simple structures, you can get to the more interesting and complex types of structures. These are structures that contain other structures as members and structures that contain arrays as members.

Structures That Contain Structures

As mentioned earlier, a C structure can contain any of C's data types. For example, a structure can contain other structures. The previous example can be extended to illustrate this. Assume that your graphics program needs to deal with rectangles. A rectangle can be defined by the coordinates of two diagonally opposite corners. You've already seen how to define a structure that can hold the two coordinates required for a single point. You need two such structures to define a rectangle. You can define a structure as follows (assuming, of course, that you have already defined the type coord structure):

```
struct rectangle {
   struct coord topleft;
   struct coord bottomrt;
};
```

This statement defines a structure of type rectangle that contains two structures of type coord.
These two type coord structures are named topleft and bottomrt.

The preceding statement defines only the type rectangle structure. To declare a structure, you must then include a statement such as struct rectangle mybox;

You could have combined the definition and declaration, as you did before for the type coord:

```
struct rectangle {
   struct coord topleft;
   struct coord bottomrt;
} mybox;
```

To access the actual data locations (the type int members), you must apply the member operator (.) twice. Thus, the expression
mybox.topleft.x
refers to the x member of the topleft member of the type rectangle structure named mybox.
To define a rectangle with coordinates (0,10),(100,200), you would write
mybox.topleft.x = 0;
mybox.topleft.y = 10;
mybox.bottomrt.x = 100;
mybox.bottomrt.y = 200;
Maybe this is getting a bit confusing. You might understand better if you look at Figure 11.1, which shows the relationship between the type rectangle structure, the two type coord structures it contains, and the two type int variables each type coord structure contains.

These structures are named as in the preceding example.
Let's look at an example of using structures that contain other structures. Listing 11.1 takes input from the user for the coordinates of a rectangle and then calculates and displays the rectangle's area. Note the program's assumptions, given in comments near the start of the program (lines 3 through 8).

Listing 11.1. A demonstration of structures that contain other structures.

```
1:  /* Demonstrates structures that contain other structures. */
2:
3:  /* Receives input for corner coordinates of a rectangle and
4:     calculates the area. Assumes that the y coordinate of the
5:     upper-left corner is greater than the y coordinate of the
6:     lower-right corner, that the x coordinate of the lower-
7:     right corner is greater than the x coordinate of the upper-
8:     left corner, and that all coordinates are positive. */
9:
10: #include <stdio.h>
11:
12: int length, width;
13: long area;
14:
15: struct coord{
16:     int x;
17:     int y;
18: };
19:
20: struct rectangle{
21:     struct coord topleft;
22:     struct coord bottomrt;
23: } mybox;
24:
25: main()
26: {
27:     /* Input the coordinates */
28:
29:     printf("\nEnter the top left x coordinate: ");
30:     scanf("%d", &mybox.topleft.x);
31:
32:     printf("\nEnter the top left y coordinate: ");
33:     scanf("%d", &mybox.topleft.y);
34:
35:     printf("\nEnter the bottom right x coordinate: ");
36:     scanf("%d", &mybox.bottomrt.x);
37:
```

```
38:    printf("\nEnter the bottom right y coordinate: ");
39:    scanf("%d", &mybox.bottomrt.y);
40:
41:    /* Calculate the length and width */
42:
43:    width = mybox.bottomrt.x - mybox.topleft.x;
44:    length = mybox.bottomrt.y - mybox.topleft.y;
45:
46:    /* Calculate and display the area */
47:
48:    area = width * length;
49:    printf("\nThe area is %ld units.\n", area);
50:
51:    return 0;
52: }
```

```
Enter the top left x coordinate: 1
Enter the top left y coordinate: 1
Enter the bottom right x coordinate: 10
Enter the bottom right y coordinate: 10
The area is 81 units.
```

ANALYSIS: The coord structure is defined in lines 15 through 18 with its two members, x and y. Lines 20 through 23 declare and define an instance, called mybox, of the rectangle structure. The two members of the rectangle structure are topleft and bottomrt, both structures of type coord. Lines 29 through 39 fill in the values in the mybox structure. At first it might seem that there are only two values to fill, because mybox has only two members. However, each of mybox's members has its own members. topleft and bottomrt have two members each, x and y from the coord structure. This gives a total of four members to be filled. After the members are filled with values, the area is calculated using the structure and member names. When using the x and y values, you must include the structure instance name. Because x and y are in a structure within a structure, you must use the instance names of both structures--mybox.bottomrt.x, mybox.bottomrt.y, mybox.topleft.x, and mybox.topleft.y--in the calculations. C places no limits on the nesting of structures. While memory allows, you can define structures that contain structures that contain structures that contain structures--well, you get the idea! Of course, there's a limit beyond which nesting becomes unproductive.Rarely are more than three levels of nesting used in any C program.

104

.

.