Documentations

Course : CSE -105
Credit Title : Structured Programming
Instructor : Md.Shamsujjoha (MSJ)

-1-

A Brief History of the C Language

- Why Use C?
- Preparing to Program
- The Program Development Cycle
  - Creating the Source Code
  - Compiling the Source Code
  - Linking to Create an Executable File
  - Completing the Development Cycle
- Your First C Program
  - Entering and Compiling HELLO.C
- Summary
- Q&A
- Workshop
  - Quiz
  - Exercises

Today you will learn

- Why C is the best choice among programming languages
- The steps in the program development cycle
- How to write, compile, and run your first C program
- About error messages generated by the compiler and linker

A Brief 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 pr^2 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.

The Program Development Cycle

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.

NOTE: 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.

Compiling the Source Code

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*.

NOTE: 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.

Linking to Create an Executable File

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*.

Figure 1.1 shows the progression from source code to object code to executable program.

Figure 1.1. *The C source code that you write is converted to object code by the compiler and then to an executable file by the 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 an .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.

Figure 1.2 shows the program development steps. 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.

Figure 1.2. *The steps involved in C program development.*

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.

NOTE: Don't enter the line numbers or colons. These are for reference only.

---

3. Save the source code. You should name the file HELLO.C.

4. Verify that HELLO.C is on disk by listing the files in the directory or folder. You should see HELLO.C within this listing.

5. 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.

6. 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)
```

7. 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.

8. 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
9. 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, the compiler 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.

NOTE: 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.

Q&A

Q If I want to give someone a program I wrote, which files do I need to give him?

A One of the nice things about C is that it is a compiled language. This means that after the source code is compiled, you have an executable program. This executable program is a stand-alone program. If you wanted to give HELLO to all your friends with computers, you could. All you need to give them is the executable program, HELLO.EXE. They don't need the source file, HELLO.C, or the object file, HELLO.OBJ. They don't need to own a C compiler, either.

Q After I create an executable file, do I need to keep the source file (.C) or object file (.OBJ)?

A If you get rid of the source file, you have no way to make changes to the program in the future, so you should keep this file. The object files are a different matter. There are reasons to keep object files, but they are beyond the scope of what you're doing now. For now, you can get rid of your object files once you have your executable file. If you need the object file, you can recompile the source file.
Most integrated development environments create files in addition to the source file (.C), the object file (.OBJ or .O), and the executable file. As long as you keep the source file (.C), you can always recreate the other files.

Q If my compiler came with an editor, do I have to use it?

A Definitely not. You can use any editor, as long as it saves the source code in text format. If the compiler came with an editor, you should try to use it. If you like a different editor better, use it. I use an editor that I purchased separately, even though all my compilers have their own editors. The editors that come with compilers are getting better. Some of them automatically format your C code. Others color-code different parts of your source file to make it easier to find errors.

Q Can I ignore warning messages?

A Some warning messages don't affect how the program runs, and some do. If the compiler gives you a warning message, it's a signal that something isn't right. Most compilers let you set the warning level. By setting the warning level, you can get only the most serious warnings, or you can get all the warnings, including the most minute. Some compilers even offer various levels in-between. In your programs, you should look at each warning and make a determination. It's always best to try to write all your programs with absolutely no warnings or errors. (With an error, your compiler won't create the executable file.)

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. Try to understand the quiz and exercise answers before continuing to the next chapter.

Quiz

1. Give three reasons why C is the best choice of programming language.
2. What does the compiler do?
3. What are the steps in the program development cycle?

4. What command do you need to enter in order to compile a program called PROGRAM1.C with your compiler?
5. Does your compiler do both the linking and compiling with just one command, or do you have to enter separate commands?
6. What extension should you use for your C source files?
7. Is FILENAME.TXT a valid name for a C source file?
8. If you execute a program that you have compiled and it doesn't work as you expected, what should you do?
9. What is machine language?
10. What does the linker do?

Exercises

1. Use your text editor to look at the object file created by Listing 1.1. Does the object file look like the source file? (Don't save this file when you exit the editor.)

2. Enter the following program and compile it. What does this program do? (Don't include the line numbers or colons.).

```
1: #include <stdio.h>
2:
3: int radius, area;
4:
5: main()
6: {
7:     printf( "Enter radius (i.e. 10): " );
8:     scanf( "%d", &radius );
9:     area = (int) (3.14159 * radius * radius);
10:    printf( "\n\nArea = %d\n", area );
11:    return 0;
12: }
```

3. Enter and compile the following program. What does this program do?

```
1: #include <stdio.h>
2:
3: int x,y;
4:
5: main()
6: {
7:     for ( x = 0; x < 10; x++, printf( "\n" ) )
8:         for ( y = 0; y < 10; y++ )
9:             printf( "X" );
10:
11:    return 0;
12: }
```

4. BUG BUSTER: The following program has a problem. Enter it in your editor and compile it. Which lines generate error messages?

```
1: #include <stdio.h>
2:
3: main();
4: {
5:     printf( "Keep looking!" );
6:     printf( "You\'ll find it!\n" );
7:     return 0;
8: }
```

5. BUG BUSTER: The following program has a problem. Enter it in your editor and compile it. Which lines generate problems?

```
1: #include <stdio.h>
2:
3: main()
4: {
5:     printf( "This is a program with a " );
6:     do_it( "problem!");
7:     return 0;
8: }
```

6. Make the following change to the program in exercise 3. Recompile and rerun this program. What does the program do now?

```
9: printf( "%c", 1 );
```

- 2 -

The Components of a C Program

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.

Note 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. More information about variables and variable definitions is presented on Day 3, "Storing Data: Variables and Constants."

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.

Note 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.

NOTE: Many people have started using a newer style of comments in their C programs. Within C++ and Java, you can use double forward slashes to signal a comment. Here are two examples:

```
// 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 Hello 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 Note 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't smart 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 )
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: }
```

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.

Q&A

Q What effect do comments have on a program?

A Comments are for the programmer. When the compiler converts the source code to object code, it throws the comments and the white space away. This means that they have no effect on the executable program. Comments do make your source file bigger, but this is usually of little concern. To summarize, you should use comments and white space to make your source code as easy to understand and main-tain as possible.

Q What is the difference between a statement and a block?

A A block is a group of statements enclosed in braces ({}). A block can be used in most places that a statement can be used.

Q How can I find out what library functions are available?

A Many compilers come with a manual dedicated specifically to documenting the library functions. They are usually in alphabetical order. Another way to find out what library functions are available is to buy a book that lists them. Appendix E, "Common C Functions," lists many of the available functions. After you begin to understand more of C, it would be a good idea to read these appendixes so that you don't rewrite a library function. (There's no use reinventing the wheel!)

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the term for a group of one or more C statements enclosed in braces?

2. What is the one component that must be present in every C program?

3. How do you add program comments, and why are they used?

4. What is a function?

5. C offers two types of functions. What are they, and how are they different?

6. What is the #include directive used for?

7. Can comments be nested?

8. Can comments be longer than one line?

9. What is another name for an include file?

10. What is an include file?

Exercises

1. Write the smallest program possible.

2. Consider the following program:

```
1:  /* EX2-2.C */
2:  #include <stdio.h>
3:
4:  void display_line(void);
5:
6:  main()
7:  {
8:      display_line();
9:      printf("\n Teach Yourself C In 21 Days!\n");
10:     display_line();
11:
12:     return 0;
13: }
14:
15: /* print asterisk line */
16: void display_line(void)
17: {
18:     int counter;
19:
20:     for( counter = 0; counter < 21; counter++ )
21:         printf("*" );
22: }
23: /* end of program */
```

a. What line(s) contain statements?

b. What line(s) contain variable definitions?

c. What line(s) contain function prototypes?

d. What line(s) contain function definitions?

e. What line(s) contain comments?

3. Write an example of a comment.

4. What does the following program do? (Enter, compile, and run it.)

```
1: /* EX2-4.C */
2: #include <stdio.h>
3:
4: main()
5: {
6:     int ctr;
7:
8:     for( ctr = 65; ctr < 91; ctr++ )
9:         printf("%c", ctr );
10:
11:    return 0;
12: }
13: /* end of program */
```

5. What does the following program do? (Enter, compile, and run it.)

```
1: /* EX2-5.C */
2: #include <stdio.h>
3: #include <string.h>
4: main()
5: {
6:     char buffer[256];
7:
8:      printf( "Enter your name and press <Enter>:\n");
9:      gets( buffer );
10:
11:     printf( "\nYour name has %d characters and spaces!",
12                     strlen( buffer ));
13:
14:     return 0;
15: }
```

Storing Data: Variables and Constants

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

If you already know how a computer's memory operates, you can skip this section. If you're not sure, however, read on. This information will help you better understand certain aspects of C programming. 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
y2x5__fg7h	Legal
annual_profit	Legal
_1990_tax	Legal but not advised
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 names 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 all capital letters unnecessarily.

Numeric Variable Types

C provides several 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 floating-point	float	4	1.2E-38 to 3.4E38[1]
Double-precision floating-point	double	8	2.2E-308 to 1.8E308[2]
[1]Approximate range; precision = 7 digits.			
[2]Approximate range; precision = 19 digits.			

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 types 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 I said the 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 type 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;
```

Note 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 */
```

Note 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.

Constants

Like a variable, a *constant* is a data storage location used by your program. Unlike a variable, the value stored in a constant can't be changed during program execution. C has two types of constants, each with its own specific uses.

Literal Constants

A *literal constant* is a value that is typed directly into the source code wherever it is needed. Here are two examples:

```
int count = 20;
float tax_rate = 0.28;
```

The 20 and the 0.28 are literal constants. The preceding statements store these values in the variables count and tax_rate. Note that one of these constants contains a decimal point, whereas the other does not. The presence or absence of the decimal point distinguishes floating-point constants from integer constants.

A literal constant written with a decimal point is a floating-point constant and is represented by the C compiler as a double-precision number. Floating-point constants can be written in standard decimal notation, as shown in these examples:

```
123.456
0.019
100.
```

Note that the third constant, 100., is written with a decimal point even though it's an integer (that is, it has no fractional part). The decimal point causes the C compiler to treat the constant as a double-precision value. Without the decimal point, it is treated as an integer constant.

Floating-point constants also can be written in scientific notation. You might recall from high school math that scientific notation represents a number as a decimal part multiplied by 10 to a positive or negative power. Scientific notation is particularly useful for representing extremely large and extremely small values. In C, scientific notation is written as a decimal number followed immediately by an E or e and the exponent:

1.23E2 1.23 times 10 to the 2nd power, or 123

4.08e6 4.08 times 10 to the 6th power, or 4,080,000

0.85e-4 0.85 times 10 to the -4th power, or 0.000085

A constant written without a decimal point is represented by the compiler as an integer number. Integer constants can be written in three different notations:

- A constant starting with any digit other than 0 is interpreted as a decimal integer (that is, the standard base-10 number system). Decimal constants can contain the digits 0 through 9 and a leading minus or plus sign. (Without a leading minus or plus, a constant is assumed to be positive.)
- A constant starting with the digit 0 is interpreted as an octal integer (the base-8 number system). Octal constants can contain the digits 0 through 7 and a leading minus or plus sign.
- A constant starting with 0x or 0X is interpreted as a hexadecimal constant (the base-16 number system). Hexadecimal constants can contain the digits 0 through 9, the letters A through F, and a leading minus or plus sign.

NOTE: See Appendix C, "Working with Binary and Hexadecimal Numbers," for a more complete explanation of decimal and hexadecimal notation.

Symbolic Constants

A *symbolic constant* is a constant that is represented by a name (symbol) in your program. Like a literal constant, a symbolic constant can't change. Whenever you need the constant's value in your program, you use its name as you would use a variable name. The actual value of the symbolic constant needs to be entered only once, when it is first defined.

Symbolic constants have two significant advantages over literal constants, as the following example shows. Suppose that you're writing a program that performs a variety of geometrical calculations. The program frequently needs the value ,, (3.14159) for its calculations. (You might recall from geometry class that ,, is the ratio of a circle's circumference to its diameter.) For example, to calculate the circumference and area of a circle with a known radius, you could write

```
circumference = 3.14159 * (2 * radius);
area = 3.14159 * (radius)*(radius);
```

The asterisk (*) is C's multiplication operator and is covered on Day 4. Thus, the first of these statements means "Multiply 2 times the value stored in the variable radius, and then multiply the result by 3.14159. Finally, assign the result to the variable named circumference."

If, however, you define a symbolic constant with the name PI and the value 3.14, you could write

```
circumference = PI * (2 * radius);
area = PI * (radius)*(radius);
```

The resulting code is clearer. Rather than puzzling over what the value 3.14 is for, you can see immediately that the constant PI is being used.

The second advantage of symbolic constants becomes apparent when you need to change a constant. Continuing with the preceding example, you might decide that for greater accuracy your program needs to use a value of PI with more decimal places: 3.14159 rather than 3.14. If you had used literal constants for PI, you would have to go through your source code and change each occurrence of the value from 3.14 to 3.14159. With a symbolic constant, you need to make a change only in the place where the constant is defined.

C has two methods for defining a symbolic constant: the #define directive and the const keyword. The #define directive is one of C's preprocessor directives, and it is discussed fully on Day 21, "Advanced Compiler Use." The #define directive is used as follows:

#define *CONSTNAME literal*

This creates a constant named *CONSTNAME* with the value of *literal*. *literal* represents a literal constant, as described earlier. *CONSTNAME* follows the same rules described earlier for variable names. By convention, the names of symbolic constants are uppercase. This makes them easy to distinguish from variable names, which by convention are lowercase. For the previous example, the required #define directive would be

#define PI 3.14159

Note that #define lines don't end with a semicolon (;). #defines can be placed anywhere in your source code, but they are in effect only for the portions of the source code that follow the #define directive. Most commonly, programmers group all #defines together, near the beginning of the file and before the start of main().

How a #define Works

The precise action of the #define directive is to instruct the compiler as follows: "In the source code, replace *CONSTNAME* with *literal*." The effect is exactly the same as if you had used your editor to go through the source code and make the changes manually. Note that #define doesn't replace instances of its target that occur as parts of longer names, within double quotes, or as part of a program comment. For example, in the following code, the instances of PI in the second and third lines would not get changed:

```
#define PI 3.14159
/* You have defined a constant for PI. */
#define PIPETTE 100
```

Defining Constants with the const Keyword

The second way to define a symbolic constant is with the const keyword. const is a modifier that can be applied to any variable declaration. A variable declared to be

const can't be modified during program execution--only initialized at the time of declaration. Here are some examples:

```
const int count = 100;
const float pi = 3.14159;
const long debt = 12000000, float tax_rate = 0.21;
```

const affects all variables on the declaration line. In the last line, debt and tax_rate are symbolic constants. If your program tries to modify a const variable, the compiler generates an error message, as shown here:

```
const int count = 100;
count = 200;        /* Does not compile! Cannot reassign or alter */
                    /* the value of a constant. */
```

What are the practical differences between symbolic constants created with the #define directive and those created with the const keyword? The differences have to do with pointers and variable scope. Pointers and variable scope are two very important aspects of C programming, and they are covered on Day 9, "Understanding Pointers," and Day 12.

Now let's look at a program that demonstrates variable declarations and the use of literal and symbolic constants. Listing 3.2 prompts the user to input his or her weight and year of birth. It then calculates and displays a user's weight in grams and his or her age in the year 2000. You can enter, compile, and run this program using the procedures explained on Day 1, "Getting Started with C."

Listing 3.2. A program that demonstrates the use of variables and constants.

```
1:    /* Demonstrates variables and constants */
2:    #include <stdio.h>
3:
4:    /* Define a constant to convert from pounds to grams */
5:    #define GRAMS_PER_POUND 454
6:
7:    /* Define a constant for the start of the next century */
8:    const int NEXT_CENTURY = 2000;
9:
10:   /* Declare the needed variables */
11:   long weight_in_grams, weight_in_pounds;
12    int year_of_birth, age_in_2000;
13:
14:   main()
15:   {
16:      /* Input data from user */
17:
18:      printf("Enter your weight in pounds: ");
19:      scanf("%d", &weight_in_pounds);
```

```
20:      printf("Enter your year of birth: ");
21:      scanf("%d", &year_of_birth);
22:
23:      /* Perform conversions */
24:
25:      weight_in_grams = weight_in_pounds * GRAMS_PER_POUND;
26:      age_in_2000 = NEXT_CENTURY - year_of_birth;
27:
28:      /* Display results on the screen */
29:
30:      printf("\nYour weight in grams = %ld", weight_in_grams);
31:      printf("\nIn 2000 you will be %d years old\n", age_in_2000);
32:
33:      return 0;
34:   }
```

Enter your weight in pounds: **175**
Enter your year of birth: **1960**
Your weight in grams = 79450
In 2000 you will be 40 years old

ANALYSIS: This program declares the two types of symbolic constants in lines 5 and 8. In line 5, a constant is used to make the value 454 more understandable. Because it uses GRAMS_PER_POUND, line 25 is easy to understand. Lines 11 and 12 declare the variables used in the program. Notice the use of descriptive names such as weight_in_grams. You can tell what this variable is used for. Lines 18 and 20 print prompts on-screen. The printf() function is covered in greater detail later. To allow the user to respond to the prompts, lines 19 and 21 use another library function, scanf(), which is covered later. scanf() gets information from the screen. For now, accept that this works as shown in the listing. Later, you will learn exactly how it works. Lines 25 and 26 calculate the user's weight in grams and his or her age in the year 2000. These statements and others are covered in detail in the next chapter. To finish the program, lines 30 and 31 display the results for the user.

DO use constants to make your programs easier to read.

DON'T try to assign a value to a constant after it has already been initialized.

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.

Q&A

Q long int variables hold bigger numbers, so why not always use them instead of int variables?

A A long int variable takes up more RAM than the smaller int. In smaller programs, this doesn't pose a problem. As programs get bigger, however, you should try to be efficient with the memory you use.

Q What happens if I assign a number with a decimal to an integer?

A You can assign a number with a decimal to an int variable. If you're using a constant variable, your compiler probably will give you a warning. The value assigned will have the decimal portion truncated. For example, if you assign 3.14 to an integer variable called pi, pi will only contain 3. The .14 will be chopped off and thrown away.

Q What happens if I put a number into a type that isn't big enough to hold it?

A Many compilers will allow this without signaling any errors. The number is wrapped to fit, however, and it isn't correct. For example, if you assign 32768 to a two-byte signed integer, the integer really contains the value -32768. If you assign the value 65535 to this integer, it really contains the value -1. Subtracting the maximum value that the field will hold generally gives you the value that will be stored.

Q What happens if I put a negative number into an unsigned variable?

A As the preceding answer indicated, your compiler might not signal any errors if you do this. The compiler does the same wrapping as if you assigned a number that was too big. For instance, if you assign -1 to an unsigned int variable that is two bytes long, the compiler will put the highest number possible in the variable (65535).

Q What are the practical differences between symbolic constants created with the #define directive and those created with the const keyword?

A The differences have to do with pointers and variable scope. Pointers and variable scope are two very important aspects of C programming and are covered on Days 9 and 12. For now, know that by using #define to create constants, you can make your programs much easier to read.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What's the difference between an integer variable and a floating-point variable?
2. Give two reasons for using a double-precision floating-point variable (type double) instead of a single-precision floating-point variable (type float).
3. What are five rules that the ANSI Standard states are always true when allocating size for variables?
4. What are the two advantages of using a symbolic constant instead of a literal constant?
5. Show two methods for defining a symbolic constant named MAXIMUM that has a value of 100.
6. What characters are allowed in C variable names?
7. What guidelines should you follow in creating names for variables and constants?
8. What's the difference between a symbolic and a literal constant?
9. What's the minimum value that a type int variable can hold?

Exercises

1. In what variable type would you best store the following values?
a. A person's age to the nearest year.
b. A person's weight in pounds.
c. The radius of a circle.
d. Your annual salary.
e. The cost of an item.
f. The highest grade on a test (assume it is always 100).
g. The temperature.
h. A person's net worth.
i. The distance to a star in miles.
2. Determine appropriate variable names for the values in exercise 1.
3. Write declarations for the variables in exercise 2.
4. Which of the following variable names are valid?
a. 123variable

b. x
c. total_score
d. Weight_in_#s
e. one.0
f. gross-cost
g. RADIUS
h. Radius
i. radius
j. this_is_a_variable_to_hold_the_width_of_a_box

- 4 -

Statements, Expressions, and Operators

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 on Day 21, "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.

Statements and White Space

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!");
```

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. Note 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.

Complex Expressions

Complex expressions consist of simpler expressions connected by operators. For example:

```
2 + 8
```

is an expression consisting of the subexpressions 2 and 8 and the addition operator +. The expression 2 + 8 evaluates, as you know, to 10. You can also write C expressions of great complexity:

1.25 / 8 + 5 * rate + rate * rate / cost

When an expression contains multiple operators, the evaluation of the expression depends on operator precedence. This concept is covered later in this chapter, as are details about all of C's operators.

C expressions get even more interesting. Look at the following assignment statement:

x = a + 10;

This statement evaluates the expression a + 10 and assigns the result to x. In addition, the entire statement x = a + 10 is itself an expression that evaluates to the value of the variable on the left side of the equal sign. This is illustrated in Figure 4.1.

Thus, you can write statements such as the following, which assigns the value of the expression a + 10 to both variables, x and y:

y = x = a + 10;

Figure 4.1. *An assignment statement is itself an expression.*

You can also write statements such as this:

x = 6 + (y = 4 + 5);

The result of this statement is that y has the value 9 and x has the value 15. Note the parentheses, which are required in order for the statement to compile. The use of parentheses is covered later in this chapter.

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 note 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.

```
1:   /* Demonstrates unary operator prefix and postfix modes */
2:
3:   #include <stdio.h>
4:
5:   int a, b;
6:
7:   main()
8:   {
9:       /* Set a and b both equal to 5 */
10:
11:      a = b = 5;
12:
13:      /* Print them, decrementing each time. */
14:      /* Use prefix mode for b, postfix mode for a */
15:
16:      printf("\n%d   %d", a--, --b);
17:      printf("\n%d   %d", a--, --b);
18:      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 illustrated 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.

NOTE: "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	==	Is operand 1 equal to operand 2?	x == y
Greater than	>	Is operand 1 greater than operand 2?	x > y
Less than	<	Is operand 1 less than operand 2?	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 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, I'll explain 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*. Note 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 */
   statementn;
}
```

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.

```
1:   /* Demonstrates the use of if statements */
2:
3:   #include <stdio.h>
4:
5:   int x, y;
6:
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:
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. Note 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.

NOTE: 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.

```
1:  /* Demonstrates the use of if statement with else clause */
2:
3:  #include <stdio.h>
4:
5:  int x, y;
6:
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.

```
1:   /* Demonstrates the evaluation of relational expressions */
2:
3:   #include <stdio.h>
4:
5:   int a;
6:
7:   main()
8:   {
9:       a = (5 == 5);          /* Evaluates to 1 */
10:      printf("\na = (5 == 5)\na = %d", a);
11:
12:      a = (5 != 5);          /* Evaluates to 0 */
13:      printf("\na = (5 != 5)\na = %d", a);
14:
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).

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.

First, all the relational operators have a lower precedence than the mathematical operators. 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.

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.

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 and change 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 to 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;
```

Logical Operators

Sometimes you might need to ask more than one relational question at once. For example, "If it's 7:00 a.m. and a weekday and not my vacation, ring the alarm." 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 4.9. Code examples of C's logical operators.

Expression	What It Evaluates To
(5 == 5) && (6 != 2)	True (1), because both operands are true
(5 > 1) || (6 < 1)	True (1), because one operand is true
(2 == 1) && (5 == 5)	False (0), because one operand is false
!(5 == 4)	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.

```
1:   #include <stdio.h>
2:
3:   /* Initialize variables. Note that c is not less than d, */
4:   /* which is one of the conditions to test for. */
5:   /* Therefore, the entire expression should evaluate as false.*/
6:
7:   int a = 5, b = 6, c = 5, d = 1;
8:   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 -= z + 1	y = y - z + 1
a /= b	a = 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.

As you'll learn in the next chapter, the most common use of the comma operator is in for statements.

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
1	() [] -> .
2	! ~ ++ -- * *(indirection)* & *(address-of)* (*type*)
	sizeof + *(unary)* - *(unary)*
3	* *(multiplication)* / %
4	+ -
5	<< >>
6	< <= > >=
7	== !=
8	& *(bitwise* AND*)*
9	^
10	|
11	&&
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

This chapter covered a lot of material. 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.

Q&A

Q What effect do spaces and blank lines have on how a program runs?

A White space (lines, spaces, tabs) makes the code listing more readable. When the program is compiled, white space is stripped and thus has no effect on the executable program. For this reason, you should use white space to make your program easier to read.

Q Is it better to code a compound if statement or to nest multiple if statements?

A You should make your code easy to understand. If you nest if statements, they are evaluated as shown in this chapter. If you use a single compound statement, the expressions are evaluated only until the entire statement evaluates to false.

Q What is the difference between unary and binary operators?

A As the names imply, unary operators work with one variable, and binary operators work with two.

Q Is the subtraction operator (-) binary or unary?

A It's both! The compiler is smart enough to know which one you're using. It knows which form to use based on the number of variables in the expression that is used. In the following statement, it is unary: x = -y;
versus the following binary use: x = a - b;

Q Are negative numbers considered true or false?

A Remember that 0 is false, and any other value is true. This includes negative numbers.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the following C statement called, and what is its meaning?

```
x = 5 + 8;
```

2. What is an expression?

3. In an expression that contains multiple operators, what determines the order in which operations are performed?

4. If the variable x has the value 10, what are the values of x and a after each of the following statements is executed separately?

```
a = x++;
a = ++x;
```

5. To what value does the expression 10 % 3 evaluate?

6. To what value does the expression 5 + 3 * 8 / 2 + 2 evaluate?

7. Rewrite the expression in question 6, adding parentheses so that it evaluates to 16.

8. If an expression evaluates to false, what value does the expression have?

9. In the following list, which has higher precedence?

a. == or <
b. * or +
c. != or ==
d. >= or >

10. What are the compound assignment operators, and how are they useful?

Exercises

1. The following code is not well-written. Enter and compile it to see whether it works.

```
#include <stdio.h>
int x,y;main(){ printf(
"\nEnter two numbers");scanf(
"%d %d",&x,&y);printf(
"\n\n%d is bigger",(x>y)?x:y);return 0;}
```

2. Rewrite the code in exercise 1 to be more readable.

3. Change Listing 4.1 to count upward instead of downward.

4. Write an if statement that assigns the value of x to the variable y only if x is between 1 and 20. Leave y unchanged if x is not in that range.

5. Use the conditional operator to perform the same task as in exercise 4.

6. Rewrite the following nested if statements using a single if statement and compound operators.

```
if (x < 1)
  if ( x > 10 )
    statement;
```

7. To what value do each of the following expressions evaluate?

a. (1 + 2 * 3)
b. 10 % 3 * 3 - (1 + 2)
c. ((1 + 2) * 3)
d. (5 == 5)
e. (x = 5)

8. If x = 4, y = 6, and z = 2, determine whether each of the following evaluates to true or false.

a. if(x == 4)
b. if(x != y - z)
c. if(z = 1)
d. if(y)

9. Write an if statement that determines whether someone is legally an adult (age 21), but not a senior citizen (age 65).

10. BUG BUSTER: Fix the following program so that it runs correctly.

```
/* a program with problems... */
#include <stdio.h>
int x= 1:
main()
{
  if( x = 1);
    printf(" x equals 1" );
  otherwise
    printf(" x does not equal 1");
  return 0;
}
```

- 5 -

Functions: The Basics

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:

- *A function 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 contains a user-defined function.

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

```
1:  /* Demonstrates a simple function */
2:  #include <stdio.h>
3:
4:  long cube(long x);
5:
6:  long input, answer;
7:
8:  main()
9:  {
10:    printf("Enter an integer value: ");
11:    scanf("%d", &input);
12:    answer = cube(input);
13:    /* Note: %ld is the conversion specifier for */
14:    /* a long integer */
15:    printf("\nThe cube of %ld is %ld.\n", input, answer);
16:
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.

NOTE: 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 on Day 12, "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.

Figure 5.1 shows a program with three functions, each of which is called once. Each time a function is called, execution passes to that function. When the function is finished, execution passes back to the place from which the function

was called. A function can be called as many times as needed, and functions can be called in any order.

Figure 5.1. *When a program calls a function, execution passes to the function and then back to the calling program.*

You now know what a function is and the importance of functions. Lessons on how to create and use your own functions follow.

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 smaller, 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.This method of programming results in a *hierarchical*, or layered, program structure. Figure 5.2 illustrates hierarchical programming for the address list program.

Figure 5.2. *A structured program is organized hierarchically.*

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*. You saw this illustrated in Figure 5.2, 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.

This is a good approach to program design. Day 13, "Advanced Program Control," shows how you can use the switch statement to create a versatile menu-driven system.

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. They are shown in Figure 5.3 and explained in the following sections.

Figure 5.3. *The three components of a function header.*

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 (given in Day 3, "Storing Data: Variables and Constants"). 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)
```

NOTE: 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 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
```

Figure 5.4 shows the relationship between arguments and parameters.

Figure 5.4. *Each time a function is called, the arguments are passed to the function's parameters.*

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 are called *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. Note 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.

```
1:   /* Demonstrates local variables. */
2:
3:   #include <stdio.h>
4:
5:   int x = 1, y = 2;
6:
7:   void demo(void);
8:
9:   main()
10:  {
11:    printf("\nBefore calling demo(), x = %d and y = %d.", x, y);
12:    demo();
13:    printf("\nAfter calling demo(), x = %d and y = %d\n.", x, y);
14:
15:    return 0;
16:  }
17:
18:  void demo(void)
19:  {
20:      /* Declare and initialize two local variables. */
21:
22:      int x = 88, y = 99;
23:
24:      /* 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.

As you can see, local variables x and y in the function are totally independent from the global variables x and y declared outside the function. 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, as shown in Figure 5.5.

Figure 5.5. *Multiple arguments are assigned to function parameters in order.*

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, I 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.

```
1:  /* Demonstrates function recursion. Calculates the */
2:  /* factorial of a number. */
3:
4:  #include <stdio.h>
5:
6:  unsigned int f, x;
7:  unsigned int factorial(unsigned int a);
```

```
8:
9:   main()
10:  {
11:     puts("Enter an integer value between 1 and 8: ");
12:     scanf("%d", &x);
13:
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(). Figure 5.6 shows the basic structure of a program that uses functions.

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. This technique is discussed on Day 21, "Advanced Compiler Use."

Figure 5.6. *Place your function prototypes before main() and your function definitions after main().*

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.

Q&A

Q What if I need to return more than one value from a function?

A Many times you will need to return more than one value from a function, or, more commonly, you will want to change a value you send to the function and keep the change after the function ends. This process is covered on Day 18, "Getting More from Functions."

Q How do I know what a good function name is?

A A good function name describes as specifically as possible what the function does.

Q When variables are declared at the top of the listing, before main(), they can be used anywhere, but local variables can be used only in the specific function. Why not just declare everything before main()?

A Variable scope is discussed in more detail on Day 12.

Q What other ways are there to use recursion?

A The factorial function is a prime example of using recursion. The factorial number is needed in many statistical calculations. Recursion is just a loop; however, it has one difference from other loops. With recursion, each time a recursive function is called, a new set of variables is created. This is not true of the other loops that you will learn about in the next chapter.

Q Does main() have to be the first function in a program?

A No. It is a standard in C that the main() function is the first function to execute; however, it can be placed anywhere in your source file. Most people place it first so that it's easy to locate.

Q What are member functions?

A Member functions are special functions used in C++ and Java. They are part of a class--which is a special type of structure used in C++ and Java.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. Will you use structured programming when writing your C programs?
2. How does structured programming work?
3. How do C functions fit into structured programming?
4. What must be the first line of a function definition, and what information does it contain?
5. How many values can a function return?
6. If a function doesn't return a value, what type should it be declared?
7. What's the difference between a function definition and a function prototype?
8. What is a local variable?
9. How are local variables special?
10. Where should the main() function be placed?

Exercises

1. Write a header for a function named do_it() that takes three type char arguments and returns a type float to the calling program.
2. Write a header for a function named print_a_number() that takes a single type int argument and doesn't return anything to the calling program.
3. What type value do the following functions return?
a. int print_error(float err_nbr);
b. long read_record(int rec_nbr, int size);
4. BUG BUSTER: What's wrong with the following listing?

```
#include <stdio.h>
void print_msg( void );
main()
{
   print_msg( "This is a message to print" );
   return 0;
}
void print_msg( void )
{
   puts( "This is a message to print" );
   return 0;
}
```

5. BUG BUSTER: What's wrong with the following function definition?

```
int twice(int y);
{
   return (2 * y);
}
```

6. Rewrite Listing 5.4 so that it needs only one return statement in the larger_of() function.
7. Write a function that receives two numbers as arguments and returns the value of their product.

8. Write a function that receives two numbers as arguments. The function should divide the first number by the second. Don't divide by the second number if it's zero. (Hint: Use an if statement.)
9. Write a function that calls the functions in exercises 7 and 8.
10. Write a program that uses a function to find the average of five type float values entered by the user.
11. Write a recursive function to take the value 3 to the power of another number. For example, if 4 is passed, the function will return 81.

- 6 -

Basic Program Control

Today you will learn

- How to use simple arrays
- How to use for, while, and do...while loops to execute statements multiple times
- How you can nest program control statements

This chapter is not intended to be a complete treatment of these topics, but I hope to provide enough information for you to be able to start writing real programs. These topics are covered in greater detail on Day 13, "Advanced Program Control."

Arrays: The Basics

Before we cover the for statement, let's take a short detour and learn the basics of arrays. (See Day 8, "Using Numeric Arrays," for a complete treatment 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], 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.

Controlling Program Execution

The default order of execution in a C program is top-down. Execution starts at the beginning of the main() function and progresses, statement by statement, until the end of main() is reached. However, this order is rarely encountered in real C programs. The C language includes a variety of program control statements that let you control the order of program execution. You have already learned how to use C's fundamental decision operator, the if statement, so let's explore three additional control statements you will find useful.

The for Statement

The for statement is a C programming construct that executes a block of one or more statements a certain number of times. It is sometimes called the for *loop*because program execution typically loops through the statement more than once. You've seen a few for statements used in programming examples earlier in this book. Now you're ready to see how the for statement works.

A for statement has the following structure:

for (*initial*; *condition*; *increment*)
  statement;

initial, *condition*, and *increment* are all C expressions, and *statement* is a single or compound C statement. When a for statement is encountered during program execution, the following events occur:

1. The expression *initial* is evaluated. *initial* is usually an assignment statement that sets a variable to a particular value.
2. The expression *condition* is evaluated. *condition* is typically a relational expression.
3. If *condition* evaluates to false (that is, as zero), the for statement terminates, and execution passes to the first statement following *statement*.
4. If *condition* evaluates to true (that is, as nonzero), the C statement(s) in *statement* are executed.
5. The expression *increment* is evaluated, and execution returns to step 2.

Figure 6.1 shows the operation of a for statement. Note that *statement* never executes if *condition* is false the first time it's evaluated.

Figure 6.1. *A schematic representation of a for statement.*

Here is a simple example. Listing 6.1 uses a for statement to print the numbers 1 through 20. You can see that the resulting code is much more compact than it would be if a separate printf() statement were used for each of the 20 values.

Listing 6.1. A simple for statement.

```
1:  /* Demonstrates a simple for statement */
2:
3:  #include <stdio.h>
4:
5:  int count;
6:
7:  main()
8:  {
9:     /* Print the numbers 1 through 20 */
10:
11:    for (count = 1; count <= 20; count++)
12:       printf("%d\n", count);
13:
14:    return 0;
15: }
```

ANALYSIS: Line 3 includes the standard input/output header file. Line 5 declares a type int variable, named count, that will be used in the for loop. Lines 11 and 12 are the for loop. When the for statement is reached, the initial statement is executed first. In this listing, the initial statement is count = 1. This initializes count so that it can be used by the rest of the loop. The second step in executing this for statement is the evaluation of the condition count <= 20. Because count was just initialized to 1, you know that it is less than 20, so the statement in the for command, the printf(), is executed. After executing the printing function, the increment expression, count++, is evaluated. This adds 1 to count, making it 2. Now the program loops back and checks the condition again. If it is true, the printf() reexecutes, the increment adds to count (making it 3), and the condition is checked. This loop continues until the condition evaluates to false, at which point the program exits the loop and continues to the next line (line 14), which returns 0 before ending the program.

The for statement is frequently used, as in the previous example, to "count up," incrementing a counter from one value to another. You also can use it to "count down," decrementing (rather than incrementing) the counter variable.

```
for (count = 100; count > 0; count--)
```

You can also "count by" a value other than 1, as in this example:

```
for (count = 0; count < 1000; count += 5)
```

The for statement is quite flexible. For example, you can omit the initialization expression if the test variable has been initialized previously in your program. (You still must use the semicolon separator as shown, however.)

```
count = 1;
for ( ; count < 1000; count++)
```

The initialization expression doesn't need to be an actual initialization; it can be any valid C expression. Whatever it is, it is executed once when the for statement is first reached. For example, the following prints the statement Now sorting the array...:

```
count = 1;
for (printf("Now sorting the array...") ; count < 1000; count++)
   /* Sorting statements here */
```

You can also omit the increment expression, performing the updating in the body of the for statement. Again, the semicolon must be included. To print the numbers from 0 to 99, for example, you could write

```
for (count = 0; count < 100; )
   printf("%d", count++);
```

The test expression that terminates the loop can be any C expression. As long as it evaluates to true (nonzero), the for statement continues to execute. You can use C's logical operators to construct complex test expressions. For example, the following for statement prints the elements of an array named array[], stopping when all elements have been printed or an element with a value of 0 is encountered:

```
for (count = 0; count < 1000 && array[count] != 0; count++)
   printf("%d", array[count]);
```

You could simplify this for loop even further by writing it as follows. (If you don't understand the change made to the test expression, you need to review Day 4.)

```
for (count = 0; count < 1000 && array[count]; )
   printf("%d", array[count++]);
```

You can follow the for statement with a null statement, allowing all the work to be done in the for statement itself. Remember, the null statement is a semicolon alone on a line. For example, to initialize all elements of a 1,000-element array to the value 50, you could write

```
for (count = 0; count < 1000; array[count++] = 50)
   ;
```

In this for statement, 50 is assigned to each member of the array by the increment part of the statement.

Day 4 mentioned that C's comma operator is most often used in for statements. You can create an expression by separating two subexpressions with the comma operator. The two subexpressions are evaluated (in left-to-right order), and the entire expression evaluates to the value of the right subexpression. By using the comma operator, you can make each part of a for statement perform multiple duties.

Imagine that you have two 1,000-element arrays, a[] and b[]. You want to copy the contents of a[] to b[] in reverse order so that after the copy operation, b[0] = a[999], b[1] = a[998], and so on. The following for statement does the trick:

```
for (i = 0, j = 999; i < 1000; i++, j--)
   b[j] = a[i];
```

The comma operator is used to initialize two variables, i and j. It is also used to increment part of these two variables with each loop.

The 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.

```
1:   /* Demonstrates nesting two for statements */
2:
3:   #include <stdio.h>
4:
5:   void draw_box( int, int);
6:
7:   main()
8:   {
9:       draw_box( 8, 35 );
10:
11:      return 0;
12:  }
13:
```

```
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 semicolon 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! */
```

The while Statement

The while statement, also called the while *loop*, executes a block of statements as long as a specified condition is true. The while statement has the following form:

```
while (condition)
   statement
```

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

1. The expression *condition* is evaluated.
2. If *condition* evaluates to false (that is, zero), the while statement terminates, and execution passes to the first statement following *statement*.
3. If *condition* evaluates to true (that is, nonzero), the C statement(s) in *statement* are executed.
4. Execution returns to step 1.

The operation of a while statement is shown in Figure 6.3.

Figure 6.3. *The operation of a while statement.*

Listing 6.3 is a simple program that uses a while statement to print the numbers 1 through 20. (This is the same task that is performed by a for statement in Listing 6.1.)

Listing 6.3. A simple while statement.

```
1:   /* Demonstrates a simple while statement */
2:
3:   #include <stdio.h>
```

```
4:
5:   int count;
6:
7:   int main()
8:   {
9:       /* Print the numbers 1 through 20 */
10:
11:      count = 1;
12:
13:      while (count <= 20)
14:      {
15:          printf("%d\n", count);
16:          count++;
17:      }
18:      return 0;
19:  }
```

ANALYSIS: Examine Listing 6.3 and compare it with Listing 6.1, which uses a for statement to perform the same task. In line 11, count is initialized to 1. Because the while statement doesn't contain an initialization section, you must take care of initializing any variables before starting the while. Line 13 is the actual while statement, and it contains the same condition statement from Listing 6.1, count <= 20. In the while loop, line 16 takes care of incrementing count. What do you think would happen if you forgot to put line 16 in this program? Your program wouldn't know when to stop, because count would always be 1, which is always less than 20.

You might have noticed that a while statement is essentially a for statement without the initialization and increment components. Thus,

for (; *condition* ;)

is equivalent to

while (*condition*)

Because of this equality, anything that can be done with a for statement can also be done with a while statement. When you use a while statement, any necessary initialization must first be performed in a separate statement, and the updating must be performed by a statement that is part of the while loop.

When initialization and updating are required, most experienced C programmers prefer to use a for statement rather than a while statement. This preference is based primarily on source code readability. When you use a for statement, the initialization, test, and increment expressions are located together and are easy to find and modify. With a while statement, the initialization and update expressions are located separately and might be less obvious.

The 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.

```
1:   /* Demonstrates nested while statements */
2:
3:   #include <stdio.h>
4:
5:   int array[5];
6:
7:   main()
8:   {
9:      int ctr = 0,
10:         nbr = 0;
11:
12:     printf("This program prompts you to enter 5 numbers\n");
13:     printf("Each number should be from 1 to 10\n");
14:
15:     while ( ctr < 5 )
16:     {
17:        nbr = 0;
18:        while (nbr < 1 || nbr > 10)
19:        {
20:           printf("\nEnter number %d of 5: ", ctr + 1 );
21:           scanf("%d", &nbr );
22:        }
23:
24:        array[ctr] = nbr;
25:        ctr++;
26:     }
27:
28:     for (ctr = 0; ctr < 5; ctr++)
29:        printf("Value %d is %d\n", ctr + 1, array[ctr] );
30:
31:     return 0;
32:  }
```

```
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 operation of a do...while loop is shown in Figure 6.4.

Figure 6.4. *The operation of a do...while loop.*

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.

```
1:   /* Demonstrates a simple do...while statement */
2:
3:   #include <stdio.h>
4:
5:   int get_menu_choice( void );
6:
7:   main()
8:   {
9:      int choice;
10:
11:     choice = get_menu_choice();
12:
13:     printf("You chose Menu Option %d\n", choice );
14:
15:     return 0;
16:  }
17:
18:  int get_menu_choice( void )
19:  {
20:     int selection = 0;
21:
22:     do
23:     {
24:        printf("\n" );
25:        printf("\n1 - Add a Record" );
26:        printf("\n2 - Change a record");
27:        printf("\n3 - Delete a record");
28:        printf("\n4 - Quit");
29:        printf("\n" );
30:        printf("\nEnter a selection: " );
31:
32:        scanf("%d", &selection );
33:
34:     }while ( selection < 1 || selection > 4 );
35:
36:     return selection;
37:  }
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.

NOTE: 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.

Summary

Now you are almost ready to start writing real C programs on your own.

C has three loop statements that control program execution: for, while, and do...while. Each of these constructs lets your program execute a block of statements zero times, one time, or more than one time, based on the condition of certain program variables. Many programming tasks are well-served by the repetitive execution allowed by these loop statements.

Although all three can be used to accomplish the same task, each is different. The for statement lets you initialize, evaluate, and increment all in one command. The while statement operates as long as a condition is true. The do...while statement always executes its statements at least once and continues to execute them until a condition is false.

Nesting is the placing of one command within another. C allows for the nesting of any of its commands. Nesting the if statement was demonstrated on Day 4. In this chapter, the for, while, and do...while statements were nested.

Q&A

Q How do I know which programming control statement to use--the for, the while, or the do...while?

A If you look at the syntax boxes provided, you can see that any of the three can be used to solve a looping problem. Each has a small twist to what it can do, however. The for statement is best when you know that you need to initialize and increment in your loop. If you only have a condition that you want to meet, and you aren't dealing with a specific number of loops, while is a good choice. If you know that a set of statements needs to be executed at least once, a do...while might be best. Because all three can be used for most problems, the best course is to learn them all and then evaluate each programming situation to determine which is best.

Q How deep can I nest my loops?

A You can nest as many loops as you want. If your program requires you to nest more than two loops deep, consider using a function instead. You might find sorting through all those braces difficult, so perhaps a function would be easier to follow in code.

Q Can I nest different loop commands?

A You can nest if, for, while, do...while, or any other command. You will find that many of the programs you try to write will require that you nest at least a few of these.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered, and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the index value of the first element in an array?
2. What is the difference between a for statement and a while statement?
3. What is the difference between a while statement and a do...while statement?
4. Is it true that a while statement can be used and still get the same results as coding a for statement?
5. What must you remember when nesting statements?
6. Can a while statement be nested in a do...while statement?
7. What are the four parts of a for statement?
8. What are the two parts of a while statement?
9. What are the two parts of a do...while statement?

Exercises

1. Write a declaration for an array that will hold 50 type long values.

2. Show a statement that assigns the value of 123.456 to the 50th element in the array from exercise 1.

3. What is the value of x when the following statement is complete?

```
for (x = 0; x < 100, x++) ;
```

4. What is the value of ctr when the following statement is complete?

```
for (ctr = 2; ctr < 10; ctr += 3) ;
```

5. How many Xs does the following print?

```
for (x = 0; x < 10; x++)
   for (y = 5; y > 0; y--)
      puts("X");
```

6. Write a for statement to count from 1 to 100 by 3s.

7. Write a while statement to count from 1 to 100 by 3s.

8. Write a do...while statement to count from 1 to 100 by 3s.

9. BUG BUSTER: What is wrong with the following code fragment?

```
record = 0;
while (record < 100)
{
   printf( "\nRecord %d ", record );
   printf( "\nGetting next number..." );
}
```

10. BUG BUSTER: What is wrong with the following code fragment? (MAXVALUES is not the problem!)

```
for (counter = 1; counter < MAXVALUES; counter++);
   printf("\nCounter = %d", counter );
```

- 7 -

Fundamentals of Input and Output

- Displaying Information On-Screen
  - The printf() Function
  - The printf() Format Strings
  - Displaying Messages with puts()
- Inputting Numeric Data with scanf()
- Summary
- Q&A
- Workshop
  - Quiz
  - Exercises

In most programs you create, you will need to display information on the screen or read information from the keyboard. Many of the programs presented in earlier chapters performed these tasks, but you might not have understood exactly how. Today you will learn

- The basics of C's input and output statements
- How to display information on-screen with the printf() and puts() library functions
- How to format the information that is displayed on-screen
- How to read data from the keyboard with the scanf() library function

This chapter isn't intended to be a complete treatment of these topics, but it provides enough information so that you can start writing real programs. These topics are covered in greater detail later in this book.

Displaying Information On-Screen

You will want most of your programs to display information on-screen. The two most frequently used ways to do this are with C's library functions printf() and puts().

The printf() Function

The printf() function, part of the standard C library, is perhaps the most versatile way for a program to display data on-screen. You've already seen printf() used in many of the examples in this book. Now you will to see how printf() works.

Printing a text message on-screen is simple. Call the printf() function, passing the desired message enclosed in double quotation marks. For example, to display An error has occurred! on-screen, you write

```
printf("An error has occurred!");
```

In addition to text messages, however, you frequently need to display the value of program variables. This is a little more complicated than displaying only a message. For example, suppose you want to display the value of the numeric variable x on-screen, along with some identifying text. Furthermore, you want the information to start at the beginning of a new line. You could use the printf() function as follows:

```
printf("\nThe value of x is %d", x);
```

The resulting screen display, assuming that the value of x is 12, would be

```
The value of x is 12
```

In this example, two arguments are passed to printf(). The first argument is enclosed in double quotation marks and is called the *format string*. The second argument is the name of the variable (x) containing the value to be printed.

The printf() Format Strings

A printf() format string specifies how the output is formatted. Here are the three possible components of a format string:

- *Literal text* is displayed exactly as entered in the format string. In the preceding example, the characters starting with the T (in The) and up to, but not including, the % comprise a literal string.
- An *escape sequence* provides special formatting control. An escape sequence consists of a backslash (\) followed by a single character. In the preceding example, \n is an escape sequence. It is called the *newline character,* and it means "move to the start of the next line." Escape sequences are also used to print certain characters. Escape sequences are listed in Table 7.1.
- A *conversion specifier* consists of the percent sign (%) followed by a single character. In the example, the conversion specifier is %d. A conversion specifier tells printf() how to interpret the variable(s) being printed. The %d tells printf() to interpret the variable x as a signed decimal integer.

Table 7.1. The most frequently used escape sequences.

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

The printf() Escape Sequences

Now let's look at the format string components in more detail. Escape sequences are used to control the location of output by moving the screen cursor. They are also used to print characters that would otherwise have a special meaning to printf(). For example, to print a single backslash character, include a double backslash (\\) in the format string. The first backslash tells printf() that the second backslash is to be interpreted as a literal character, not as the start of an escape sequence. In general, the backslash tells printf() to interpret the next character in a special manner. Here are some examples:

Sequence	*Meaning*

n	The character n
\n	Newline
\"	The double quotation character
"	The start or end of a string

Table 7.1 lists C's most commonly used escape sequences. A full list can be found on Day 15, "Pointers: Beyond the Basics."

Listing 7.1 demonstrates some of the frequently used escape sequences.

Listing 7.1. Using printf() escape sequences.

```
1:   /* Demonstration of frequently used escape sequences */
2:
3:   #include <stdio.h>
4:
5:   #define QUIT  3
6:
7:   int  get_menu_choice( void );
8:   void print_report( void );
9:
10:  main()
11:  {
12:      int choice = 0;
13:
14:      while (choice != QUIT)
15:      {
16:          choice = get_menu_choice();
17:
18:          if (choice == 1)
19:              printf("\nBeeping the computer\a\a\a" );
20:          else
21:          {
22:              if (choice == 2)
23:                  print_report();
24:          }
25:      }
26:      printf("You chose to quit!\n");
27:
28:      return 0;
29:  }
30:
31:  int get_menu_choice( void )
32:  {
33:      int selection = 0;
34:
35:      do
```

```
36:     {
37:        printf( "\n" );
38:        printf( "\n1 - Beep Computer" );
39:        printf( "\n2 - Display Report");
40:        printf( "\n3 - Quit");
41:        printf( "\n" );
42:        printf( "\nEnter a selection:" );
43:
44:        scanf( "%d", &selection );
45:
46:     }while ( selection < 1 || selection > 3 );
47:
48:     return selection;
49: }
50:
51: void print_report( void )
52: {
53:     printf( "\nSAMPLE REPORT" );
54:     printf( "\n\nSequence\tMeaning" );
55:     printf( "\n=========\t=======" );
56:     printf( "\n\\a\t\tbell (alert)" );
57:     printf( "\n\\b\t\tbackspace" );
58:     printf( "\n...\t\t...");
59: }
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:1
Beeping the computer
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:2
SAMPLE REPORT
Sequence        Meaning
=========       =======
\a              bell (alert)
\b              backspace
...             ...
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:3
You chose to quit!
```

ANALYSIS: Listing 7.1 seems long compared with previous examples, but it offers some additions that are worth noting. The STDIO.H header was included in line 3 because printf() is used in this listing. In line 5, a constant named QUIT is defined. From Day 3, "Storing Data: Variables and Constants," you know that #define makes using the constant QUIT equivalent to using the value 3. Lines 7 and 8 are function prototypes. This program has two functions: get_menu_choice() and print_report(). get_menu_choice() is defined in lines 31 through 49. This is similar to the menu function in Listing 6.5. Lines 37 and 41 contain calls to printf() that print the newline escape sequence. Lines 38, 39, 40, and 42 also use the newline escape character, and they print text. Line 37 could have been eliminated by changing line 38 to the following:

```
printf( "\n\n1 - Beep Computer" );
```

However, leaving line 37 makes the program easier to read.

Looking at the main() function, you see the start of a while loop on line 14. The while loop's statements will keep looping as long as choice is not equal to QUIT. Because QUIT is a constant, you could have replaced it with 3; however, the program wouldn't be as clear. Line 16 gets the variable choice, which is then analyzed in lines 18 through 25 in an if statement. If the user chooses 1, line 19 prints the newline character, a message, and then three beeps. If the user selects 2, line 23 calls the function print_report().

print_report() is defined on lines 51 through 59. This simple function shows the ease of using printf() and the escape sequences to print formatted information to the screen. You've already seen the newline character. Lines 54 through 58 also use the tab escape character, \t. It aligns the columns of the report vertically. Lines 56 and 57 might seem confusing at first, but if you start at the left and work to the right, they make sense. Line 56 prints a newline (\n), then a backslash (\), then the letter a, and then two tabs (\t\t). The line ends with some descriptive text, (bell (alert)). Line 57 follows the same format.

This program prints the first two lines of Table 7.1, along with a report title and column headings. In exercise 9 at the end of this chapter, you will complete this program by making it print the rest of the table.

The printf() Conversion Specifiers

The format string must contain one conversion specifier for each printed variable. printf() then displays each variable as directed by its corresponding conversion specifier. You'll learn more about this process on Day 15. For now, be sure to use the conversion specifier that corresponds to the type of variable being printed.

Exactly what does this mean? If you're printing a variable that is a signed decimal integer (types int and long), use the %d conversion specifier. For an unsigned decimal integer (types unsigned int and unsigned long), use %u. For a floating-

point variable (types float and double), use the %f specifier. The conversion specifiers you need most often are listed in Table 7.2.

Table 7.2. The most commonly needed conversion specifiers.

Specifier	Meaning	Types Converted
%c	Single character	char
%d	Signed decimal integer	int, short
%ld	Signed long decimal integer	long
%f	Decimal floating-point number	float, double
%s	Character string	char arrays
%u	Unsigned decimal integer	unsigned int, unsigned short
%lu	Unsigned long decimal integer	unsigned long

The literal text of a format specifier is anything that doesn't qualify as either an escape sequence or a conversion specifier. Literal text is simply printed as is, including all spaces.

What about printing the values of more than one variable? A single printf() statement can print an unlimited number of variables, but the format string must contain one conversion specifier for each variable. The conversion specifiers are paired with variables in left-to-right order. If you write

```
printf("Rate = %f, amount = %d", rate, amount);
```

the variable rate is paired with the %f specifier, and the variable amount is paired with the %d specifier. The positions of the conversion specifiers in the format string determine the position of the output. If there are more variables passed to printf() than there are conversion specifiers, the unmatched variables aren't printed. If there are more specifiers than variables, the unmatched specifiers print "garbage."

You aren't limited to printing the value of variables with printf(). The arguments can be any valid C expression. For example, to print the sum of x and y, you could write

```
z = x + y;
printf("%d", z);
```

You also could write

```
printf("%d", x + y);
```

Any program that uses printf() should include the header file STDIO.H. Listing 7.2 demonstrates the use of printf(). Day 15 gives more details on printf().

Listing 7.2. Using printf() to display numerical values.

```
1:  /* Demonstration using printf() to display numerical values. */
2:
3:  #include <stdio.h>
4:
5:  int a = 2, b = 10, c = 50;
6:  float f = 1.05, g = 25.5, h = -0.1;
7:
8:  main()
9:  {
10:     printf("\nDecimal values without tabs: %d %d %d", a, b, c);
11:     printf("\nDecimal values with tabs: \t%d \t%d \t%d", a, b, c);
12:
13:     printf("\nThree floats on 1 line: \t%f\t%f\t%f", f, g, h);
14:     printf("\nThree floats on 3 lines: \n\t%f\n\t%f\n\t%f", f, g, h);
15:
16:     printf("\nThe rate is %f%%", f);
17:     printf("\nThe result of %f/%f = %f\n", g, f, g / f);
18:
19:     return 0;
20: }
Decimal values without tabs: 2 10 50
Decimal values with tabs:       2       10      50
Three floats on 1 line:         1.050000        25.500000       -0.100000
Three floats on 3 lines:
        1.050000
        25.500000
        -0.100000
The rate is 1.050000%
The result of 25.500000/1.050000 = 24.285715
```

ANALYSIS: Listing 7.2 prints six lines of information. Lines 10 and 11 each print three decimals: a, b, and c. Line 10 prints them without tabs, and line 11 prints them with tabs. Lines 13 and 14 each print three float variables: f, g, and h. Line 13 prints them on one line, and line 14 prints them on three lines. Line 16 prints a float variable, f, followed by a percent sign. Because a percent sign is normally a message to print a variable, you must place two in a row to print a single percent sign. This is exactly like the backslash escape character. Line 17 shows one final concept. When printing values in conversion specifiers, you don't have to use variables. You can also use expressions such as g / f, or even constants.

DON'T try to put multiple lines of text into one printf() statement. In most instances, it's clearer to print multiple lines with multiple print statements than to use just one with several newline (\n) escape characters.

DON'T forget to use the newline escape character when printing multiple lines of information in separate printf() statements.

DON'T misspell stdio.h. Many C programmers accidentally type studio.h; however, there is no u.

The printf() Function

#include <stdio.h>
printf(*format-string*[,*arguments*,...]);

printf() is a function that accepts a series of *arguments*, each applying to a conversion specifier in the given format string. printf() prints the formatted information to the standard output device, usually the display screen. When using printf(), you need to include the standard input/output header file, STDIO.H.

The *format-string* is required; however, arguments are optional. For each argument, there must be a conversion specifier. Table 7.2 lists the most commonly used conversion specifiers.

The *format-string* can also contain escape sequences. Table 7.1 lists the most frequently used escape sequences.

The following are examples of calls to printf() and their output:

Example 1 Input

```
#include <stdio.h>
main()
{
   printf("This is an example of something printed!");
   return 0;
}
```

Example 1 Output

```
This is an example of something printed!
```

Example 2 Input

```
printf("This prints a character, %c\na number, %d\na floating \
point, %f", `z', 123, 456.789 );
```

Example 2 Output

```
This prints a character, z
a number, 123
a floating point, 456.789
```

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. Note 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 Input

puts("This prints on the first line. \nThis prints on the second line.");

```
puts("This prints on the third line.");
puts("If these were printf()s, all four lines would be on two lines!");
```

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 Numeric Data with scanf()

Just as most programs need to output data to the screen, they also need to input data from the keyboard. The most flexible way your program can read numeric data from the keyboard is by using the scanf() library function.

The scanf() function reads data from the keyboard according to a specified format and assigns the input data to one or more program variables. Like printf(), scanf() uses a format string to describe the format of the input. The format string utilizes the same conversion specifiers as the printf() function. For example, the statement

```
scanf("%d", &x);
```

reads a decimal integer from the keyboard and assigns it to the integer variable x. Likewise, the following statement reads a floating-point value from the keyboard and assigns it to the variable rate:

```
scanf("%f", &rate);
```

What is that ampersand (&) before the variable's name? The & symbol is C's *address-of* operator, which is fully explained on Day 9, "Understanding Pointers." For now, all you need to remember is that scanf() requires the & symbol before each numeric variable name in its argument list (unless the variable is a pointer, which is also explained on Day 9).

A single scanf() can input more than one value if you include multiple conversion specifiers in the format string and variable names (again, each preceded by & in the argument list). The following statement inputs an integer value and a floating-point value and assigns them to the variables x and rate, respectively:

```
scanf("%d %f", &x, &rate);
```

When multiple variables are entered, scanf() uses white space to separate input into fields. White space can be spaces, tabs, or new lines. Each conversion specifier in the scanf() format string is matched with an input field; the end of each input field is identified by white space.

This gives you considerable flexibility. In response to the preceding scanf(), you could enter

10 12.45

You also could enter this:

10 12.45

or this:

10
12.45

As long as there's some white space between values, scanf() can assign each value to its variable.

As with the other functions discussed in this chapter, programs that use scanf() must include the STDIO.H header file. Although Listing 7.3 gives an example of using scanf(), a more complete description is presented on Day 15.

Listing 7.3. Using scanf() to obtain numerical values.

```
1:   /* Demonstration of using scanf() */
2:
3:   #include <stdio.h>
4:
5:   #define QUIT 4
6:
7:   int get_menu_choice( void );
8:
9:   main()
10:  {
11:      int   choice    = 0;
12:      int   int_var   = 0;
13:      float float_var = 0.0;
14:      unsigned unsigned_var = 0;
15:
16:      while (choice != QUIT)
17:      {
18:          choice = get_menu_choice();
19:
20:          if (choice == 1)
21:          {
22:              puts("\nEnter a signed decimal integer (i.e. -123)");
23:              scanf("%d", &int_var);
24:          }
25:          if (choice == 2)
26:          {
```

```
27:             puts("\nEnter a decimal floating-point number\
28                  (i.e. 1.23)");
29:             scanf("%f", &float_var);
30:         }
31:         if (choice == 3)
32:         {
33:              puts("\nEnter an unsigned decimal integer \
34                  (i.e. 123)" );
35:              scanf( "%u", &unsigned_var );
36:         }
37:     }
38:     printf("\nYour values are: int: %d  float: %f  unsigned: %u \n",
39:                              int_var, float_var, unsigned_var );
40:
41:    return 0;
42:  }
43:
44:  int get_menu_choice( void )
45:  {
46:     int selection = 0;
47:
48:     do
49:     {
50:        puts( "\n1 - Get a signed decimal integer" );
51:        puts( "2 - Get a decimal floating-point number" );
52:        puts( "3 - Get an unsigned decimal integer" );
53:        puts( "4 - Quit" );
54:        puts( "\nEnter a selection:" );
55:
56:        scanf( "%d", &selection );
57:
58:      }while ( selection < 1 || selection > 4 );
59:
60:      return selection;
61:  }
```

```
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
1
Enter a signed decimal integer (i.e. -123)
-123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
```

Enter a selection:
3
Enter an unsigned decimal integer (i.e. 123)
321
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
2
Enter a decimal floating point number (i.e. 1.23)
1231.123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
4
Your values are: int: -123 float: 1231.123047 unsigned: 321

ANALYSIS: Listing 7.3 uses the same menu concepts that were used in Listing 7.1. The differences in get_menu_choice() (lines 44 through 61) are minor but should be noted. First, puts() is used instead of printf(). Because no variables are printed, there is no need to use printf(). Because puts() is being used, the newline escape characters have been removed from lines 51 through 53. Line 58 was also changed to allow values from 1 to 4 because there are now four menu options. Notice that line 56 has not changed; however, now it should make a little more sense. scanf() gets a decimal value and places it in the variable selection. The function returns selection to the calling program in line 60.

Listings 7.1 and 7.3 use the same main() structure. An if statement evaluates choice, the return value of get_menu_choice(). Based on choice's value, the program prints a message, asks for a number to be entered, and reads the value using scanf(). Notice the difference between lines 23, 29, and 35. Each is set up to get a different type of variable. Lines 12 through 14 declare variables of the appropriate types.

When the user selects Quit, the program prints the last-entered number for all three types. If the user didn't enter a value, 0 is printed, because lines 12, 13, and 14 initialized all three types. One final note on lines 20 through 36: The if statements used here are not structured well. If you're thinking that an if...else structure would have been better, you're correct. Day 14, "Working with the Screen, Printer, and Keyboard," introduces a new control statement, switch. This statement offers an even better option.

DON'T forget to include the address-of operator (&) when using scanf() variables.

DO use printf() or puts() in conjunction with scanf(). Use the printing functions to display a prompting message for the data you want scanf() to get.

The scanf() Function

#include <stdio.h>
scanf(*format-string*[,*arguments*,...]);

scanf() is a function that uses a conversion specifier in a given format-string to place values into variable arguments. The arguments should be the addresses of the variables rather than the actual variables themselves. For numeric variables, you can pass the address by putting the address-of operator (&) at the beginning of the variable name. When using scanf(), you should include the STDIO.H header file.

scanf() reads input fields from the standard input stream, usually the keyboard. It places each of these read fields into an argument. When it places the information, it converts it to the format of the corresponding specifier in the format string. For each argument, there must be a conversion specifier. Table 7.2 lists the most commonly needed conversion specifiers.

Example 1

```
int x, y, z;
scanf( "%d %d %d", &x, &y, &z);
```

Example 2

```
#include <stdio.h>
main()
{
   float y;
   int x;
   puts( "Enter a float, then an int" );
   scanf( "%f %d", &y, &x);
   printf( "\nYou entered %f and %d ", y, x );
   return 0;
}
```

Summary

With the completion of this chapter, you are ready to write your own C programs. By combining the printf(), puts(), and scanf() functions and the programming control statements you learned about in earlier chapters, you have the tools needed to write simple programs.

Screen display is performed with the printf() and puts() functions. The puts() function can display text messages only, whereas printf() can display text

messages and variables. Both functions use escape sequences for special characters and printing controls.

The scanf() function reads one or more numeric values from the keyboard and interprets each one according to a conversion specifier. Each value is assigned to a program variable.

Q&A

Q Why should I use puts() if printf() does everything puts() does and more?

A Because printf() does more, it has additional overhead. When you're trying to write a small, efficient program, or when your programs get big and resources are valuable, you will want to take advantage of the smaller overhead of puts(). In general, you should use the simplest available resource.

Q Why do I need to include STDIO.H when I use printf(), puts(), or scanf()?

A STDIO.H contains the prototypes for the standard input/output functions. printf(), puts(), and scanf() are three of these standard functions. Try running a program without the STDIO.H header and see the errors and warnings you get.

Q What happens if I leave the address-of operator (&) off a scanf() variable?

A This is an easy mistake to make. Unpredictable results can occur if you forget the address-of operator. When you read about pointers on Days 9 and 13, you will understand this better. For now, know that if you omit the address-of operator, scanf() doesn't place the entered information in your variable, but in some other place in memory. This could do anything from apparently having no effect to locking up your computer so that you must reboot.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered, and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the difference between puts() and printf()?
2. What header file should you include when you use printf()?
3. What do the following escape sequences do?
a. \\

b. \b
c. \n
d. \t
e. \a

4. What conversion specifiers should be used to print the following?

a. A character string
b. A signed decimal integer
c. A decimal floating-point number

5. What is the difference between using each of the following in the literal text of puts()?

a. b
b. \b

Exercises

NOTE: Starting with this chapter, some of the exercises ask you to write complete programs that perform a particular task. Because there is always more than one way to do things in C, the answers provided at the back of the book shouldn't be interpreted as the only correct ones. If you can write your own code that performs what's required, great! If you have trouble, refer to the answer for help. The answers are presented with minimal comments because it's good practice for you to figure out how they operate.

---

1. Write both a printf() and a puts() statement to start a new line.

2. Write a scanf() statement that could be used to get a character, an unsigned decimal integer, and another single character.

3. Write the statements to get an integer value and print it.

4. Modify exercise 3 so that it accepts only even values (2, 4, 6, and so on).

5. Modify exercise 4 so that it returns values until the number 99 is entered, or until six even values have been entered. Store the numbers in an array. (Hint: You need a loop.)

6. Turn exercise 5 into an executable program. Add a function that prints the values, separated by tabs, in the array on a single line. (Print only the values that were entered into the array.)

7. BUG BUSTER: Find the error(s) in the following code fragment:

```
printf( "Jack said, "Peter Piper picked a peck of pickled peppers."");
```

8. BUG BUSTER: Find the error(s) in the following program:

```
int get_1_or_2( void )
{
   int answer = 0;
   while (answer < 1 || answer > 2)
   {
      printf(Enter 1 for Yes, 2 for No);
      scanf( "%f", answer );
   }
   return answer;
}
```

9. Using Listing 7.1, complete the print_report() function so that it prints the rest of Table 7.1.

10. Write a program that inputs two floating-point values from the keyboard and then displays their product.

11. Write a program that inputs 10 integer values from the keyboard and then displays their sum.

12. Write a program that inputs integers from the keyboard, storing them in an array. Input should stop when a zero is entered or when the end of the array is reached. Then, find and display the array's largest and smallest values. (Note: This is a tough problem, because arrays haven't been completely covered in this book yet. If you have difficulty, try solving this problem again after reading Day 8, "Using Numeric Arrays.")