1

C Fundamentals

chapter objectives

1.1 Understand the components of a C program

1.2 Create and compile a program

1.3 Declare variables and assign values

1.4 Input numbers from the keyboard

1.5 Perform calculations using arithmetic expressions

1.6 Add comments to a program

1.7 Write your own functions

1.8 Use functions to return values

1.9 Use function arguments

1.10 Remember the C keywords

THE individual elements of a computer language such as C do not stand alone, but rather in conjunction with one another. Therefore, it is necessary to understand several key aspects of C before examining each element of the language in detail. To this end, this chapter presents a quick overview of the C language. Its goal is to give you sufficient working knowledge of C so that you can understand the examples in later chapters.

As you work through this chapter, don't worry if a few points are not entirely clear. The main thing you need to understand is how and why the example programs execute as they do. Keep in mind that most of the topics introduced in this chapter will be discussed in greater detail later in this book. In this chapter, you will learn about the basic structure of a C program; what a C statement is; and what variables, constants, and functions are. You will learn how to display text on the screen and input information from the keyboard.

To use this book to the fullest, you must have a computer, a C compiler, and a text editor. (You may also use a C++ compiler. C++ compilers can also compile C programs.) Your compiler may include its own text editor, in which case you won't need a separate one. For the best results, you should work along with the examples and try the exercises.

UNDERSTAND THE COMPONENTS OF A C PROGRAM

All C programs share certain essential components and traits. All C programs consist of one or more *functions*, each of which contains one or more *statements*. In C, a function is a named subroutine that can be called by other parts of the program. Functions are the building blocks of C. A statement specifies an action to be performed by the program. In other words, statements are the parts of your program that actually perform operations.

All C statements end with a semicolon. C does not recognize the end of the line as a terminator. This means there are no constraints on the position of statements within a line. Also, you may place two or more statements on one line.

The general form of a C function is shown here:

```
ret-type function-name(param-list)
{
 statement sequence
}
```

Here, *ret-type* specifies the type of data returned by the function. As you will see, it is possible for a function to return a value. The *function-name* is the name of the function. Information can be passed to a function through its parameters, which are specified in the function's parameter list, *param-list*. The *statement sequence* may be one or more statements. (Technically, a function can contain no statements, but since this means the function performs no action, it is a degenerative case.) If return types and parameters are new concepts, don't worry, they will be explained later in this chapter.

With few exceptions, you can call a function by any name you like. It must be composed of only the upper- and lowercase letters of the alphabet, the digits 0-9, and the underscore. A digit cannot start a function name, however. C is *case-sensitive*, which means that C recognizes the difference between upper- and lowercase letters. Thus, as far as C is concerned, **Myfunc** and **myfunc** are entirely different names.

Although a C program may contain several functions, the only function that it *must* have is **main( )**. The **main( )** function is where execution of your program begins. That is, when your program begins running, it starts executing the statements inside **main( )**, beginning with the first statement after the opening curly brace. Your program ends when **main( )**'s closing curly brace is reached. Of course, the curly brace does not actually exist in the compiled version of your program, but it is helpful to think of it in this way.

Throughout this book, when a function is referred to in text, it will be printed in bold and followed by parentheses. This way, you can see immediately that the name refers to a function, not some other part of the program.

Another important component of all C programs is *library functions*. The ANSI C standard specifies a set of library functions to be supplied by all C compilers, which your program may use. This collection of functions is usually referred to as the *C standard library*. The standard library contains functions to perform disk I/O (input/ output), string manipulations, mathematical computations, and much more. When your program is compiled, the code for each library function used by your program is automatically included. This differs from the way some other computer languages work. For example, in BASIC or Pascal, operations such as writing to a file or computing a cosine are performed using keywords that are built into the language. The advantage C gains by having them as library functions is increased flexibility. Library functions can be enhanced and expanded as needed to accommodate changing circumstances. The C language itself does not need to change. As you will see, virtually all C programs you create will use functions from the C standard library.

One of the most common library functions is called **printf( )**. This is C's general-purpose output function. The **printf( )** function is quite versatile, allowing many variations. Its simplest form is shown here:

```
printf("string-to-output");
```

The **printf( )** function outputs the characters that are contained between the beginning and ending double quotes to the screen. (The double quotes are not displayed on the screen.) In C, one or more characters enclosed between double quotes is called a *string*. The quoted string between **printf( )**'s parentheses is said to be an *argument* to **printf( )**. In general, information passed to a function is called an argument. In C, calling a library function is a statement; therefore, it must end with a semicolon.

To call a function, you specify its name followed by a parenthesized list of arguments that you will be passing to it. If the function does not require any arguments, no arguments will be specified—and the parenthesized list will be empty. If there is more than one argument, the arguments must be separated by commas.

Another component common to most C programs is the *header file*. In C, information about the standard library functions is found in various files supplied with your compiler. These files all end with a .H extension. The C compiler uses the information in these files to handle the library functions properly. You add these files to your program using the **#include** *preprocessor directive*. All C compilers use as their first phase of compilation a *preprocessor*, which performs various manipulations on your source file before it is compiled.

Preprocessor directives are not actually part of the C language, but rather instructions from you to the compiler. The **#include** directive tells the preprocessor to read in another file and include it with your program. You will learn more about the preprocessor later in this book.

The most commonly required header file is called STDIO.H. Here is the directive that includes this file:

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

You can specify the file name in either upper- or lowercase, but lowercase is the traditional method. The STDIO.H header file contains, among other things, information related to the **printf( )** library function. Notice that the **#include** directive does not end with a semicolon. The reason for this is that **#include** is not a C keyword that can define a statement. Instead, it is an instruction to the C compiler itself.

One last point: With few exceptions, C ignores spaces. That is, it doesn't care where on a line a statement, curly brace, or function name occurs. If you like, you can even put two or more of these items on the same line. The examples you will see in this book reflect the way C code is normally written; it is a form you should follow. The actual positioning of statements, functions, and braces is a stylistic, not a programming, decision.

EXAMPLES

1. Since all C programs share certain common traits, understanding one program will help you understand many others. One of the simplest C programs is shown here:

```
#include <stdio.h>

int main(void)
{
  printf("This is a short C program.");

  return 0;
}
```

When compiled and executed, this program displays the message **This is a short C program.** on the screen of your computer.

Even though this program is only six lines long, it illustrates those aspects common to all C programs. Let's examine it line by line.

The first line of the program is

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

It causes the file STDIO.H to be read by the C compiler and to be included with the program. This file contains information related to **printf( )**.

The second line,

```
int main(void)
```

begins the **main( )** function. As stated earlier, all C programs must have a **main( )** function. This is where program execution begins. The **int** specifies that **main( )** returns an integer value. The **void** tells the compiler that **main( )** does not have any parameters.

After **main( )** is an opening curly brace. This marks the beginning of statements that make up the function.
The next line in the program is

```
printf("This is a short C program.");
```

This is a C statement. It calls the standard library function, **printf( )**, which causes the string to be displayed.

The following line causes **main( )** to return the value zero. In this case, the value is returned to the calling process, which is usually the operating system.

```
return 0;
```

By convention, a return value of zero from **main( )** indicates normal program termination. Any other value represents an error. The operating system can test this value to determine whether the program ran successfully or experienced an error. **return** is one of C's keywords and is examined more closely later in this chapter.

Finally, the program is formally concluded when **main( )**'s closing curly brace is encountered.

2. Here is another simple C program:

```
#include <stdio.h>

int main(void)
{
  printf("This is ");
  printf("another C ");
  printf("program.");

  return 0;
}
```

This program displays **This is another C program.** on the screen. The key point to this program is that statements are executed sequentially, beginning with the opening curly brace and ending with the closing curly brace.

CREATE AND COMPILE A PROGRAM

How you will create and compile a program is determined to a very large extent by the compiler you are using and the operating system under which it is running. If you are using a PC or compatible, you have your choice of a number of excellent compilers, such as those by Borland and Microsoft, that contain integrated program-development environments. If you are using such an environment, you can edit, compile, and run your programs directly inside this environment. This is an excellent option for beginners—just follow the instructions supplied with your compiler.

If you are using a traditional command-line compiler, then you need to follow these steps to create and compile a program:

1. Create your program using an editor.
2. Compile the program.
3. Execute your program.

The exact method to accomplish these steps will be explained in the user's manual for your compiler.

Nearly all modern C compilers are also C++ compilers. As you may know, C++ is the object-oriented extension to C. Most likely you will be using a C++ compiler to compile your C code. Don't worry. This is perfectly acceptable because all C++ compilers are capable of compiling C programs. For example, if you are using Borland C++ or Microsoft Visual C++, then everything will work just fine. However, there is one thing about which you must be very careful: the extension you give your files.

When naming your program's file, you must give it a .C—not .CPP—extension. This is important. If you are using a C++ compiler, then it will automatically assume that a file using the .C extension contains a C program and will compile it as a C program. But, if the file uses a .CPP extension, then the compiler will assume that the program is written in C++ and compile it as such. The problem is that while C is the foundation for C++, not all C programs are valid C++ programs. Trying to compile a C program *as if it were* a C++ program will cause errors in a few cases. Since the programs in this book are C programs, they should be compiled as C programs. Using the .C extension ensures this.

Remember *Your program must be compiled as a C program—not a C++ program. To ensure this, make sure that your programs use the .C, not the .CPP, extension.*

The file that contains the C program that you create is called the *source file*. The file that contains the compiled form of your program that the computer executes is called the *object file*, or, sometimes, the *executable file*.

If you enter something into your program incorrectly, the compiler will report syntax error messages when it attempts to compile it. Most C compilers attempt to make sense out of your source code no matter what you have written. For this reason, the error that gets reported may not always reflect the actual cause of the error. For example, accidentally forgetting the opening curly brace to the **main( )** function in the preceding sample programs will cause some compilers to report the **printf( )** statement as an incorrect identifier. So, when you receive a syntax error message, be prepared to look at the last few lines of code in your program before the point at which the error is reported to find its cause.

Many compilers report not only actual errors but also warning errors. The C language was designed to be very forgiving and to allow virtually anything that is syntactically correct to be compiled. However, some things, even though syntactically correct, are highly suspicious. When the compiler encounters one of these situations it prints a warning. You, as the programmer, then decide whether its suspicions are justified. Frankly, some compilers are a bit too helpful and flag warnings on perfectly correct C statements. More important, some compilers allow you to enable various options that simply report information about your program that you might like to know. Sometimes this type of information is reported in the form of a warning message, even though there is nothing to be "warned" about. The programs in this book are in compliance with the ANSI standard for C and will not generate any warning messages about which you need be concerned.

EXAMPLES

1. If you are using Borland C++, you can create and compile your program using the integrated environment. Online instructions are provided. If you are using the command-line version of Borland C++, you will use a command line such as this (assuming that the name of your program is called TEST.C) to compile the program once you have used a text editor to create it.

   ```
   BCC TEST.C
   ```

2. If you are using Microsoft Visual C++, you can use the integrated environment to create and compile your program. Online instructions are provided. When using the command line compiler, this command line will compile your program after using a text file to create it. (Again, assume that the program is called TEST.C.)

   ```
   CL TEST.C
   ```

3. If you are using another brand of compiler, refer to your user's manual for details on compiling your programs.

EXERCISE

1. Enter into your computer the example programs from Section 1.1. Compile them and run them.

DECLARE VARIABLES AND ASSIGN VALUES

A *variable* is a named memory location that can hold various values. Only the most trivial C programs do not include variables. In C, unlike some computer languages, all variables must be declared before they can be used. A variable's declaration serves one important purpose: It tells the compiler *what type of variable* is being used. C supports five different basic data types, as shown in Table 1-1 along with the C keywords that represent them. Don't be confused by **void**. This is a special-purpose data type that we will later examine closely.

A variable of type **char** is 8 bits long and is most commonly used to hold a single character. Because C is very flexible, a variable of type **char** can also be used as a "little integer" if desired.

Integer variables (**int**) may hold signed whole numbers (numbers with no fractional part). For 16-bit environments, such as DOS or Windows 3.1, integers are usually 16 bits long and may hold values in the range -32,768 to 32,767. In 32-bit environments, such as Windows NT or Windows 95, integers are typically 32 bits in length. In this case, they may store values in the range -2,147,483,648 to 2,147,483,647.

Type	Keyword
character data	char
signed whole numbers	int
floating-point numbers	float
double-precision floating-point numbers	double
valueless	void

TABLE 1-1 *C's Five Basic Data Types*

Variables of types **float** and **double** hold signed floating-point values, which may have fractional components. One difference between **float** and **double** is that **double** provides about twice the precision (number of significant digits) as does **float**. Also, for most uses of C, a variable of type **double** is capable of storing values with absolute magnitudes larger than those stored by variables of type **float**. Of course, in all cases, variables of types **float** and **double** can hold very large values.

To declare a variable, use this general form:

type var-name;

where *type* is a C data type and *var-name* is the name of the variable. For example, this declares **counter** to be of type **int**:

```
int counter;
```

In C, a variable declaration is a statement and it must end in a semicolon.

There are two places where variables are declared: inside a function or outside all functions. Variables declared outside all functions are called *global variables* and they may be accessed by any function in your program. Global variables exist the entire time your program is executing.

Variables declared inside a function are called *local variables*. A local variable is known to—and may be accessed by—only the function in which it is declared. It is common practice to declare all local variables used by a function at the start of the function, after the opening curly brace. There are two important points you need to know about local variables at this time. First, the local variables in one function have no relationship to the local variables in another function. That is, if a variable called **count** is declared in one function, another variable called **count** may also be declared in a second function—the two variables are completely separate from and unrelated to each other. The second thing you need to know is that local variables are created when a function is called, and they are destroyed when the function is exited. Therefore, local variables do not maintain their values between function calls. The examples in this and the next few chapters will use only local variables. Chapter 4 discusses more thoroughly the issues and implications of global and local variables.

You can declare more than one variable of the same type by using a comma-separated list. For example, this declares three floating-point variables, **x**, **y**, and **z**:

```
float x, y, z;
```

Like function names, variable names in C can consist of the letters of the alphabet, the digits 0 through 9, and the underscore. (But a digit may not start a variable's name.) Remember, C is case-sensitive; **count** and **COUNT** are two completely different variable names.

To assign a value to a variable, put its name to the left of an equal sign. Put the value you want to give the variable to the right of the equal sign. In C, an assignment operation is a statement; therefore, it must be terminated by a semicolon. The general form of an assignment statement is:

variable-name = value;

For example, to assign an integer variable named **num** the value 100, you can use this statement:

```
num = 100;
```

In the preceding assignment, 100 is a constant. Just as there are different types of variables, there are different types of constants. A *constant* is a fixed value used in your program. Constants are often used to initialize variables at the beginning of a program's execution.

A character constant is specified by placing the character between single quotes. For example, to specify the letter "A," you would use **'A'**. Integers are specified as whole numbers. Floating-point values must include a decimal point. For example, to specify 100.1, you would use **100.1**. If the floating-point value you wish to specify does not have any digits to the right of the decimal point, then you must use 0. For example, to tell the compiler that 100 is a floating-point number, use **100.0**.

You can use **printf( )** to display values of characters, integers, and floating-point values. To do so, however, requires that you know more about the **printf( )** function. Let's look first at an example. This statement:

```
printf("This prints the number %d", 99);
```

displays **This prints the number 99** on the screen. As you can see, this call to **printf( )** contains not one, but two arguments. The first is the quoted string and the other is the constant 99. Notice that the arguments are separated from each other by a comma. In general, when there is more than one argument to a function, the arguments are separated from each other by commas. The operation of the **printf( )** function is as follows. The first argument is a quoted string that may contain either normal characters or format specifiers that begin with the percent sign. Normal characters are simply displayed as-is on the screen in the order in which they are encountered in the string (reading left to right). A format specifier, also called a format code, informs **printf( )** that a different type item is to be displayed. In this case, the **%d** means that an integer is to be output in decimal format. The value to be displayed is found in the second argument. This value is then output to the screen at the point where the format specifier is found in the string. To understand the relationship between the normal characters and the format codes, examine this statement:

```
printf("This displays %d, too", 99);
```

Now the call to **printf( )** displays **This displays 99, too**. The key point is that the value associated with a format code is displayed at the point where that format code is encountered in the string.

If you want to specify a character value, the format specifier is **%c**. To specify a floating-point value, use **%f**. The **%f** works for both **float** and **double**. As you will see, **printf( )** has many more capabilities.

Keep in mind that the values matched with the format specifier need not be constants; they may be variables, too.

EXAMPLES

1. The program shown here illustrates the three new concepts introduced in this section. First, it declares a variable named **num**. Second, it assigns this variable the value 100. Finally, it uses **printf( )** to display **The value is 100** on the screen. Examine this program closely:

```
#include <stdio.h>

int main(void)
{
  int num;

  num = 100;
  printf("The value is %d", num);

  return 0;
}
```

The statement

```
int num;
```

declares **num** to be an integer variable.

To display the value of **num**, the program uses this statement:

```
printf("The value is %d", num);
```

2. This program creates variables of types **char**, **float**, and **double**; assigns each a value; and outputs these values to the screen.

```
#include <stdio.h>

int main(void)
{
  char ch;
  float f;
  double d;

  ch = 'X';
  f = 100.123;
  d = 123.009;

  printf("ch is %c, ", ch);
  printf("f is %f, ", f);
  printf("d is %f", d);

  return 0;
}
```

EXERCISES

1. Enter, compile, and run the example programs in this section.
2. Write a program that declares one integer variable called **num**. Give this variable the value 1000 and then, using one **printf( )** statement, display the value on the screen like this:

```
1000 is the value of num
```

INPUT NUMBERS FROM THE KEYBOARD

Although there are actually several ways to input numeric values from the keyboard, one of the easiest is to use another of C's standard library functions called **scanf( )**. Although it possesses considerable versatility, we will use it in this chapter to read only integers and floating-point numbers entered from the keyboard.

To use **scanf( )** to read an integer value from the keyboard, call it using the general form

scanf("%d", &*int-var-name*);

where *int-var-name* is the name of the integer variable you wish to receive the value. The first argument to **scanf( )** is a string that determines how the second argument will be treated. In this case, the **%d** specifies that the second argument will be receiving an integer value entered in decimal format. This fragment, for example, reads an integer entered from the keyboard.

```
int num;
scanf("%d", &num);
```

The **&** preceding the variable name is essential to the operation of **scanf( )**. Although a detailed explanation will have to wait until later, loosely, the **&** allows a function to place a value into one of its arguments.

It is important to understand one key point: When you enter a number at the keyboard, you are simply typing a string of digits. The **scanf( )** function waits until you have pressed ENTER before it converts the string into the internal binary format used by the computer.

To read a floating-point number from the keyboard, call **scanf( )** using the general form

scanf("%f", &*float-var-name*);

where *float-var-name* is the name of a variable that is declared as being of type **float**. If you want to input to a **double** variable, use the **%lf** specifier.

Notice that the format specifiers for **scanf( )** are similar to those used for **printf( )** for the corresponding data types except that **%lf** is used to read a **double**. This is no coincidence—**printf( )** and **scanf( )** are complementary functions.

EXAMPLE

1. This program asks you to input an integer and a floating-point number. It then displays the values you enter.

```
#include <stdio.h>

int main(void)
{
  int num;
  float f;

  printf("Enter an integer: ");
  scanf("%d", &num);

  printf("Enter a floating point number: ");
  scanf("%f", &f);

  printf("%d ", num);
  printf("%f", f);

  return 0;
}
```

EXERCISES

1. Enter, compile, and run the example program.
2. Write a program that inputs two floating-point numbers (use type **float**) and then displays their sum.

PERFORM CALCULATIONS USING ARITHMETIC EXPRESSIONS

In C, the expression plays a much more important role than it does in most other programming languages. Part of the reason for this is that C defines many more operators than do most other languages. An *expression* is a combination of operators and operands. C expressions follow the rules of algebra, so, for the most part, they will be familiar. In this section we will look only at arithmetic expressions.

C defines these five arithmetic operators:

Operator	Meaning
+	addition
–	subtraction
*	multiplication
/	division
%	modulus

The +, –, /, and * operators may be used with any of the basic data types. However, the **%** may be used with integer types only. The modulus operator produces the remainder of an integer division. This has no meaning when applied to floating-point types.

The – has two meanings. First, it is the subtraction operator. Second, it can be used as a unary minus to reverse the sign of a number. A unary operator uses only one operand.

An expression may appear on the right side of an assignment statement. For example, this program fragment assigns the integer variable **answer** the value of 100*31.

```
int answer;
answer = 100 * 31;
```

The *, /, and **%** are higher in precedence than the + and the –. However, you can use parentheses to alter the order of evaluation. For example, this expression produces the value zero,

```
10 - 2 * 5
```

but this one produces the value 40.

```
(10 - 2) * 5
```

A C expression may contain variables, constants, or both. For example, assuming that **answer** and **count** are variables, this expression is perfectly valid:

```
answer = count - 100;
```

Finally, you may use spaces liberally within an expression.

EXAMPLES

1. As stated earlier, the modulus operator returns the remainder of an integer division. The remainder of 10 % 3 equals 1, for example. This program shows the outcome of some integer divisions and their remainders:

```
#include <stdio.h>

int main(void)
{
  printf("%d", 5/2);
  printf(" %d", 5%2);
  printf(" %d", 4/2);
  printf(" %d", 4%2);

  return 0;
}
```

This program displays **2 1 2 0** on the screen.

2. In long expressions, the use of parentheses and spaces can add clarity, even if they are not necessary. For example, examine this expression:

```
count *num+88/val-19%count
```

This expression produces the same result, but is much easier to read:

```
(count * num) + (88 / val) - (19 % count)
```

3. This program computes the area of a rectangle, given its dimensions. It first prompts the user for the length and width of the rectangle and then displays the area.

```
#include <stdio.h>

int main(void)
{
  int len, width;

  printf("Enter length: ");
  scanf("%d", &len);
  printf("Enter width: ");
  scanf("%d", &width);

  printf("Area is %d", len * width);

  return 0;
}
```

4. As stated earlier, the - can be used as a unary operator to reverse the sign of its operand. To see how this works, try this program:

```
#include <stdio.h>

int main(void)
{
  int i;

  i = 10;
  i = -i;
  printf("This is i: %d", i);
```

```
    return 0;
}
```

EXERCISES

1. Write a program that computes the volume of a cube. Have the program prompt the user for each dimension.
2. Write a program that computes the number of seconds in a year.

ADD COMMENTS TO A PROGRAM

A *comment* is a note to yourself (or others) that you put into your source code. All comments are ignored by the compiler. They exist solely for your benefit. Comments are used primarily to document the meaning and purpose of your source code, so that you can remember later how it functions and how to use it.

In C, the start of a comment is signaled by the /* character pair. A comment is ended by */. For example, this is a syntactically correct C comment:

```
/* This is a comment. */
```

Comments can extend over several lines. For example, this is completely valid in C:

```
/*
  This is a longer comment
  that extends over
  five lines.
*/
```

In C, a comment can go anywhere except in the middle of any C keyword, function name, or variable name.

You can use a comment to temporarily remove a line of code. Simply surround the line with the comment symbols.

Although not currently defined by ANSI C standard, you may see another style of comment, called a *single-line comment*. It begins with a // and stops at the end of the line. The single-line comment was created by C++. Its use in a C program is technically invalid, but most compilers will accept it. As such, many programmers have begun to use it in their C programs. Since the single-line comment is not defined by the current ANSI C standard, this book will not use it. However, don't be surprised if you see it in commercially written Cprograms.

One final point: In C, you can't have one comment within another comment. That is, comments may not be nested. For example, C will not accept this:

```
/* this is a comment /* this is another comment
  nested inside the first - which will cause
  a syntax error */ with a nested comment
*/
```

EXAMPLES

1. A year on Jupiter (the time it takes for Jupiter to make one full circuit around the Sun) takes about 12 Earth years. The following program allows you to convert Earth days to Jovian years. Simply specify the number of Earth days, and it computes the equivalent number of Jovian years. Notice the use of comments throughout the program.

```
/* This program converts Earth days into Jovian years. */
#include <stdio.h>

int main(void)
{
  float e_days; /* number of Earth days */
  float j_years; /* equivalent number of Jovian years */

  /* get number of Earth days */
```

```
    printf("Enter number of Earth days: ");
    scanf("%f", &e_days);

    /* now, compute Jovian years */
    j_years = e_days / (365.0 * 12.0);

    /* display the answer */
    printf("Equivalent Jovian years: %f", j_years);

    return 0;
  }
```

Notice that comments can appear on the same line as other C program statements.

Comments are often used to help describe what the program is doing. Although this program is easy to understand even without the comments, many programs are very difficult to understand even with the liberal use of comments. For more complex programs, the general approach is the same as used here: simply describe the actions of the program. Also, notice the comment at the start of the program. In general, it is a good idea to identify the purpose of a program at the top of its source file.

2. You cannot place a comment inside the name of a function or variable name. For example, this is an incorrect statement:

```
pri/* wrong */ntf("this won't work");
```

EXERCISES

1. Go back and add comments to the programs developed in previous sections.
2. Is this comment correct?

```
/**/
```

3. Is this comment correct?

```
/* printf("this is a test"); */
```

WRITE YOUR OWN FUNCTIONS

Functions are the building blocks of C. So far, the programs you have seen included only one function: **main( )**. Most real-world programs, however, will contain many functions. In this section you will begin to learn how to write programs that contain multiple functions.

The general form of a C program that has multiple functions is shown here:

```
/* include header files here */

/* function prototypes here */

int main(void)
{
 /* ... */
}

ret-type f1(param-list)
{
 /* ... */
}

ret-type f2(param-list)
{
 /* ... */
}
.
.
.
ret-type fN(param-list)
{
 /* ... */
}
```

Of course, you can call your functions by different names. Here, *ret-type* specifies the type of data returned by the function. If a function does not return a value, then its return type should be **void**. If a function does not use parameters, then its *param-list* should contain the keyword **void**.

Notice the comment about prototypes. A *function prototype* declares a function before it is used and prior to its definition. A prototype consists of a function's name, its return type, and its parameter list. It is terminated by a semicolon. The compiler needs to know this information in order for it to properly execute a call to the function. For example, given this simple function:

```
void myfunc(void)
{
  printf("This is a test.");
}
```

Its prototype is

```
void myfunc(void);
```

The only function that does not need a prototype is **main( )** since it is predefined by the C language.

Prototypes are an important part of C programming, but you will need to learn more about C before you can fully understand their purpose and value. For the next few chapters we will be using prototypes without any further explanation. They will be included as needed in all of the example programs shown in this book. You should also include them in programs that you write. A full explanation of prototypes is found in Chapter 7.

When a function is called, execution transfers to that function. When the end of that function is reached, execution returns to a point immediately after the place at which the function was called. Put differently, when a function ends, execution resumes at the point in your program immediately following the call to the function. Any function inside a program may call any other function within the same program. Traditionally, **main( )** is not called by any other function, but there is no technical restriction to this effect.

In the examples that follow, you will learn to create the simplest type of C functions: those that that do not return values and do not use parameters. The skeletal form of such a function is shown here:

```
void FuncName(void) {
  /* body of  unction here */
}
```

Of course, the name of the function will vary. Because the function does not return a value, its return type is **void**. Because the function does not have parameters, its parameter list is **void**.

EXAMPLES

1. The following program contains two functions: **main( )** and **func1( )**. Try to determine what it displays on the screen before reading the description that follows it.

```
/* A program with two functions */

#include <stdio.h>

void func1(void); /* prototype for func1() */

int main(void)
{
  printf("I ");
  func1();
  printf("C.");

  return 0;
}

void func1(void)
{
  printf("like ");
}
```

This program displays **I like C.** on the screen. Here is how it works. In **main( )**, the first call to **printf( )** executes, printing the **I**. Next, **func1( )** is called. This causes the **printf( )** inside **func1( )** to execute, displaying **like**. Since this is the only statement inside **func1( )**, the function returns. This causes execution to resume inside **main( )** and the **C.** is printed. Notice that the statement that calls **func1( )** ends with a semicolon. (Remember a function call is a statement.)

A key point to understand about writing your own functions is that when the closing curly brace is reached the function will return, and execution resumes one line after the point at which the function was called.

Notice the prototype for **func1( )**. As you can see, it consists of its name, return type, and parameters list, but no body. It is terminated by a semicolon.

2. This program prints **1 2 3** on the screen:

```
/* This program has three functions. */

#include <stdio.h>

void func1(void); /* prototypes */
void func2(void);

int main(void)
{
  func2();
  printf("3");

  return 0;
}

void func2(void)
{
  func1();
  printf("2 ");
}

void func1(void)
{
  printf("1 ");
}
```

In this program, **main( )** first calls **func2( )**, which then calls **func1( )**. Next, **func1( )** displays **1** and then returns to **func2( )**, which prints **2** and then returns to **main( )**, which prints **3**.

EXERCISES

1. Enter, compile, and run the two example programs in this section.
2. Write a program that contains at least two functions and prints the message **The summer soldier, the sunshine patriot**.
3. Remove the prototype from the first example program and then compile it. What happens?

USE FUNCTIONS TO RETURN VALUES

In C, a function may return a value to the calling routine. For example, another of C's standard library functions is **sqrt( )**, which returns the square root of its argument. For your program to obtain the return value, you must put the function on the right side of an assignment statement. For example, this program prints the square root of 10:

```
#include <stdio.h>
#include <math.h> /* needed by sqrt() */

int main(void)
{
  double answer;

  answer = sqrt(10.0);
  printf("%f", answer);

  return 0;
}
```

This program calls **sqrt( )** and assigns its return value to **answer**. Notice that **sqrt( )** uses the MATH.H header file.

Actually, the assignment statement in the preceding program is not technically necessary because **sqrt( )** could simply be used as an argument to **printf( )**, as shown here:

```
#include <stdio.h>
#include <math.h> /* needed by sqrt() */

int main(void)
{
  printf("%f", sqrt(10.0));

  return 0;
}
```

The reason this works is that C will automatically call **sqrt( )** and obtain its return value before calling **printf( )**. The return value then becomes the second argument to **printf( )**. If this seems strange, don't worry; you will understand this sort of situation better as you learn more about C.

The **sqrt( )** function requires a floating-point value for its argument, and the value it returns is of type **double**. You must match the type of value a function returns with the variable that the value will be assigned to. As you learn more about C, you will see why this is important. It is also important that you match the types of a function's arguments to the types it requires.

When writing your own functions, you can return a value to the calling routine using the **return** statement. The **return** statement takes the general form

return *value*;

where *value* is the value to be returned. For example, this program prints **10** on the screen:

```
#include <stdio.h>

int func(void); /* prototype */

int main(void)
{
  int num;

  num = func();
  printf("%d", num);

  return 0;
}

int func(void)
{
  return 10;
}
```

In this example, **func( )** returns an integer value and its return type is specified as **int**. Although you can create functions that return any type of data, functions that return values of type **int** are quite common. Later in this book, you will see many examples of functions that return other types. Functions that are declared as **void** may not return values.

If a function does not explicitly specify a return type, it is assumed to return an integer by default. For example, **func( )** could have been coded like this:

```
func(void)
{
  return 10;
}
```

In this case, the **int** is implied. The use of the "default to **int**" rule is very common in older C code. However, recently there has been a move away from using the integer default. Whether this trend will continue is unknown. In any event, to avoid misunderstandings, this book will always explicitly specify **int**.

One important point: When the **return** statement is encountered, the function returns immediately. No statements after it will be executed. Thus, a **return** statement causes a function to return before its closing curly brace is reached.

The value associated with the **return** statement need not be a constant. It can be any valid C expression.

A **return** statement can also be used by itself, without a return value. This form of **return** looks like this:

```
return ;
```

It is used mostly by **void** functions (i.e., functions that have a **void** return type) to cause the function to return immediately, before the function's closing curly brace is reached. While not recommended, you can also use this form of **return** in functions that are supposed to return values. However, doing so makes the returned value undefined.

There can be more than one **return** in a function. You will see examples of this later in this book.

Even though a function returns a value, you don't necessarily have to assign that value to anything. If the return value of a function is not used, it is lost, but no harm is done.

EXAMPLES

1. This program displays the square of a number entered from the keyboard. The square is computed using the **get_sqr( )** function. Its operation should be clear.

```
#include <stdio.h>

int get_sqr(void);

int main(void)
{
  int sqr;

  sqr = get_sqr();
  printf("Square: %d", sqr);

  return 0;
}

int get_sqr(void)
{
  int num;

  printf("Enter a number: ");
  scanf("%d", &num);
  return num*num; /* square the number */
}
```

2. As mentioned earlier, you can use **return** without specifying a value. This allows a function to return before its closing curly brace is reached. For example, in the following program, the

```
#include <stdio.h>

void func1(void);

int main(void)
{
  func1();

  return 0;
}

void func1(void)
{
  printf("This is printed.");
  return; /* return with no value */
  printf("This is never printed.");
}
```

EXERCISES

1. Enter, compile, and run the example programs in this section.
2. Write a program that uses a function called **convert( )**, which prompts the user for an amount in dollars and returns this value converted into pounds. (Use an exchange rate of $2.00 per pound.) Display the conversion.
3. What is wrong with this program?

```
#include <stdio.h>

int f1(void);

int main(void)
{
  double answer;

  answer = f1();
  printf("%f", answer);

  return 0;
}

int f1(void)
{
  return 100;
}
```

4. What is wrong with this function?

```
void func(void)
{
  int i;

  printf("Enter a number: ");
  scanf("%d", &i);

  return i;
}
```

USE FUNCTION ARGUMENTS

As stated earlier, a function's argument is a value that is passed to the function when the function is called. A function in C can have from zero to several arguments. (The upper limit is determined by the compiler you are using, but the ANSI C standard specifies that a function must be able to take at least 31 arguments.) For a function to be able to take arguments, special variables to receive argument values must be declared. These are called the *formal parameters* of the function. The parameters are declared between the parentheses that follow the function's name. For example, the function listed below prints the sum of the two integer arguments used to call it.

```
void sum(int x, int y)
{
  printf("%d ", x + y);
}
```

Each time **sum( )** is called, it will sum the value passed to **x** with the value passed to **y**. Remember, however, that **x** and **y** are simply the function's operational variables, which receive the values you use when calling the function. Consider the following short program, which illustrates how to call **sum( )**.

```
/* A simple program that demonstrates sum(). */

#include <stdio.h>

void sum(int x, int y);

int main(void)
{
  sum(1, 20);
  sum(9, 6);
  sum(81, 9);

  return 0;
}

void sum(int x, int y)
{
  printf("%d ", x + y);
}
```

This program will print **21**, **15**, and **90** on the screen. When **sum( )** is called, the value of each argument is copied into its matching parameter. That is, in the first call to **sum( )**, 1 is copied into **x** and 20 is copied into **y**. In the second call, 9 is copied into **x** and 6 into **y**. In the third call, 81 is copied into **x** and 9 into **y**.

If you have never worked with a language that allows parameterized functions, the preceding process may seem strange. Don't worry—as you see more examples of C programs, the concept of arguments, parameters, and functions will become clear.

It is important to keep two terms straight. First, *argument* refers to the value that is passed to a function. The variable that receives the value of the argument inside the function is the *formal parameter* of the function. Functions that take arguments are called *parameterized* *functions*. Remember, if a variable is used as an argument to a function, it has nothing to do with the formal parameter that receives its value.

In C functions, arguments are always separated by commas. In this book, the term *argument list* will refer to comma-separated arguments.

All function parameters are declared in a fashion similar to that used by **sum( )**. You must specify the type and name of each parameter and, if there is more than one parameter, you must use a comma to separate them. Functions that do not have parameters should use the keyword **void** in their parameter list.

EXAMPLES

1. An argument to a function can consist of an expression. For example, it is perfectly valid to call **sum( )** as shown here:

```
sum(10-2, 9*7);
```

2. This program uses the **outchar( )** function to output characters to the screen. The program prints **ABC**.

```
#include <stdio.h>

void outchar(char ch);

int main(void)
{
  outchar('A');
  outchar('B');
  outchar('C');

  return 0;
}

void outchar(char ch)
{
  printf("%c", ch);
}
```

EXERCISES

1. Write a program that uses a function called **outnum( )** that takes one integer argument and displays it on the screen.
2. What is wrong with this program?

```
#include <stdio.h>

void sqr_it(int num);

int main(void)
{
  sqr_it(10.0);

  return 0;
}

void sqr_it(int num)
{
  printf("%d", num * num);
}
```

1.10 REMEMBER THE C KEYWORDS

Before concluding this chapter, you should familiarize yourself with the keywords that make up the C language. ANSI C standard has 32 *keywords* that may not be used as variable or function names. These words, combined with the formal C syntax, form the C programming language. They are listed in Table 1-2.

Many C compilers have added several additional keywords that are used to take better advantage of the environment in which the compiler is used, and that give support for interlanguage programming, interrupts, and memory organization. Some commonly used extended keywords are shown in Table 1-3.

The lowercase lettering of the keywords is significant. C requires that all keywords be in lowercase form. For example, **RETURN** will *not* be recognized as the keyword **return**. Also, no keyword may be used as a variable or function name.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

TABLE 1-2 *The 32 Keywords as Defined by the ANSI C Standard* ▼

asm	_cs	_ds	_es
_ss	cdecl	far	huge
interrupt	near	pascal	

TABLE 1-3 *Some Common C Extended Keywords* ▼

Mastery Skills Check

1. The moon's gravity is about 17 percent of Earth's. Write a program that allows you to enter your weight and computes your effective weight on the moon.
2. What is wrong with this program fragment?

```
/* this inputs a number
scanf("%d", &num);
```

3. There are 8 ounces in a cup. Write a program that converts ounces to cups. Use a function called **o_to_c( )** to perform the conversion. Call it with the number of ounces and have it return the number of cups.
4. What are the five basic data types in C?

5. What is wrong with each of these variable names?

   a) short-fall

   b) $balancc

   c) last + name

   d) 9times

2

Introducing C's Program Control Statements

chapter objectives

2.1 Become familiar with the **if**

2.2 Add the **else**

2.3 Create blocks of code

2.4 Use the **for** loop

2.5 Substitute C's increment and decrement operators

2.6 Expand **printf( )**'s capabilities

2.7 Program with C's relational and logical operators

IN this chapter you will learn about two of C's most important program control statements: **if** and **for**. In general, program control statements determine your program's flow of execution. As such, they form the backbone of your programs. In addition to these, you will also learn about blocks of code, the relational and logical operators, and more about the **printf( )** function.

Review Skills Check

Before proceeding, you should be able to correctly answer these questions and do these exercises:

1. All C programs are composed of one or more functions. What is the name of the function that all programs must have? Further, what special purpose does it perform?
2. The **printf( )** function is used to output information to the screen. Write a program that displays **This is the number 100.** (Output the **100** as a number, not as a string.)
3. Header files contain information used by the standard library functions. How do you tell the compiler to include one in your program? Give an example.
4. C supports five basic types of data. Name them.
5. Which of these variable names are invalid in C?
   a. _count
   b. 123count
   c. $test
   d. This_is_a_long_name
   e. new-word
6. What is **scanf( )** used for?
7. Write a program that inputs an integer from the keyboard and displays its square.
8. How are comments entered into a C program? Give an example.
9. How does a function return a value to the routine that called it?

10. A function called **Myfunc( )** has these three parameters: an **int** called **count**, a **float** called **balance**, and a **char** called **ch**. The function does not return a value. Show how this function is prototyped.

2.1 BECOME FAMILIAR WITH THE if

The **if** statement is one of C's *selection statements* (sometimes called *conditional statements*). Its operation is governed by the outcome of a conditional test that evaluates to either true or false. Simply put, selection statements make decisions based upon the outcome of some condition.

In its simplest form, the **if** statement allows your program to conditionally execute a statement. This form of the **if** is shown here:

if(*expression*) *statement*;

The expression may be any valid C expression. If the expression evaluates as true, the statement will be executed. If it does not, the statement is bypassed, and the line of code following the **if** is executed. In C, an expression is true if it evaluates to any nonzero value. If it evaluates to zero, it is false. The statement that follows an **if** is usually referred to as the *target* of the **if** statement.

Commonly, the expression inside the **if** compares one value with another using a *relational operator*. Although you will learn about all the relational operators later in this chapter, three are introduced here so that we can create some example programs. A relational operator tests how one value relates to another. For example, to see if one value is greater than another, C uses the **>** relational operator. The outcome of this comparison is either true or false. For example, **10 > 9** is true, but **9 > 10** is false. Therefore, the following **if** will cause the message **true** to be displayed.

```
if(10 > 9) printf("true");
```

However, because the expression in the following statement is false, the **if** does not execute its target statement.

```
if(5 > 9) printf("this will not print");
```

C uses **<** as its *less than operator*. For example, **10 < 11** is true. To test for equality, C provides the == operator. (There can be no space between the two equal signs.) Therefore, **10 == 10** is true, but **10 == 11** is not.

Of course, the expression inside the **if** may involve variables. For example, the following program tells whether an integer entered from the keyboard is negative or non-negative.

```
#include <stdio.h>

int main(void)
{
  int num;

  printf("Enter an integer: ");
  scanf("%d", &num);

  if(num < 0) printf("Number is negative.");
  if(num > -1) printf("Number is non-negative.");

  return 0;
}
```

Remember, in C, true is any nonzero value and false is zero. Therefore, it is perfectly valid to have an **if** statement such as the one shown here:

```
if(count+1) printf("Not Zero");
```

EXAMPLES

1. This program forms the basis for an addition drill. It displays two numbers and asks the user what the answer is. The program then tells the user if the answer is right or wrong.

   ```
   #include <stdio.h>

   int main(void)
   {
     int answer;
   ```

```
  printf("What is 10 + 14? ");
  scanf("%d", &answer);
  if(answer == 10+14) printf("Right!");

  return 0;
}
```

2. This program converts either feet to meters or meters to feet, depending upon what the user requests.

```
#include <stdio.h>

int main(void)
{
  float num;
  int choice;

  printf("Enter value: ");
  scanf("%f", &num);

  printf("1: Feet to Meters, 2: Meters to Feet. ");
  printf("Enter choice: ");
  scanf("%d", &choice);

  if(choice == 1) printf("%f", num / 3.28);
  if(choice == 2) printf("%f", num * 3.28);

  return 0;
}
```

EXERCISES

1. Which of these expressions are true?

   a. 0

   b. 1

   c. 10 * 9 < 90

   d. 1==1

   e. -1

2. Write a program that asks the user for an integer and then tells the user if that number is even or odd. (Hint, use C's modulus operator %.)

2.2 ADD THE else

B

You can add an **else** statement to the **if**. When this is done, the **if** statement looks like this:

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

If the expression is true, then the target of the **if** will execute, and the **else** portion will be skipped. However, if the expression is false, then the target of the **if** is bypassed, and the target of the **else** will execute. Under no circumstances will both statements execute. Thus, the addition of the **else** provides a two-way decision path.

EXAMPLES

1. You can use the **else** to create more efficient code in some cases. For example, here the **else** is used in place of a second **if** in the program from the preceding section, which determines whether a number is negative or non-negative.

```
#include <stdio.h>

int main(void)
{
  int num;

  printf("Enter an integer: ");
  scanf("%d", &num);

  if(num < 0) printf("Number is negative.");
  else printf("Number is non-negative.");
  return 0;
}
```

Recall that the original version of this program explicitly tested for non-negative numbers by comparing **num** to -1 using a second **if** statement. But since there are only two possibilities—**num** is either negative or non-negative—there is no reason for this second test. Because of the way a C compiler generates code, the **else** requires far fewer machine instructions than an additional **if** and is, therefore, more efficient.

2. This program prompts the user for two numbers, divides the first by the second, and displays the result. However, division by zero is undefined, so the program uses an **if** and an **else** statement to prevent division by zero from occurring.

```
#include <stdio.h>

int main(void)
{
  int num1, num2;

  printf("Enter first number: ");
  scanf("%d", &num1);

  printf("Enter second number: ");
  scanf("%d", &num2);

  if(num2 == 0) printf("Cannot divide by zero.");
  else printf("Answer is: %d.", num1 / num2);

  return 0;
}
```

EXERCISES

1. Write a program that requests two numbers and then displays either their sum or product, depending on what the user selects.

2. Rewrite Exercise 2 from Section 2.1 so that it uses an **else** statement.

2.3 CREATE BLOCKS OF CODE

In C, you can link two or more statements together. This is called a *block of code* or a *code block*. To create a block of code, you surround the statements in the block with opening and closing curly braces. Once this is done, the statements form one logical unit, which may be used anywhere that a single statement may.

For example, the general form of the **if** using blocks of code is

```
if(expression) {
  statement1;
  statement2;
  .
  .
  .
  statement N;
}
else {
  statement1;
  statement2;
  .
  .
  .
  statement N;
}
```

If the expression evaluates to true, then all the statements in the block of code associated with the **if** will be executed. If the expression is false, then all the statements in the **else** block will be executed. (Remember, the **else** is optional and need not be present.) For example, this fragment prints the message **This is an example of a code block.** if the user enters any positive number.

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

if(num > 0) {
  printf("This is ");
  printf("an example of ");
  printf("a code block.");
}
```

Keep in mind that a block of code represents one indivisible logical unit. This means that under no circumstances could one of the **printf( )** statements in this fragment execute without the others also executing.

In the example shown, the statements that appear within the block of code are indented. Although C does not care where a statement appears on a line, it is common practice to indent one level at the start of a block. Indenting makes the structure of a program easier to understand. Also, the placement of the curly braces is arbitrary. However, the way they are shown in the example is a common method and will be used by the examples in this book.

In C, as you will see, you can use a block of code anywhere you can use a single statement.

EXAMPLES

1. This program is an improved version of the feet-to-meters, meters-to-feet conversion program. Notice how the use of code blocks allows the program to prompt specifically for each unit.

```
#include <stdio.h>

int main(void)
{
  float num;
  int choice;

  printf("1: Feet to Meters, 2: Meters to Feet. ");
  printf("Enter choice: ");
  scanf("%d", &choice);

  if(choice == 1) {
    printf("Enter number of feet: ");
    scanf("%f", &num);
    printf("Meters: %f", num / 3.28);
  }
  else {
    printf("Enter number of meters: ");
    scanf("%f", &num);
    printf("Feet: %f", num * 3.28);
  }

  return 0;
}
```

2. Using code blocks, we can improve the addition drill program so that it also prints the correct answer when the user makes a mistake.

```
#include <stdio.h>

int main(void)
{
  int answer;

  printf("What is 10 + 14? ");
  scanf("%d", &answer);

  if(answer == 10+14) printf("Right!");
  else {
    printf("Sorry, you're wrong. ");
    printf("The answer is 24.");
  }

  return 0;
}
```

This example illustrates an important point: it is not necessary for targets of both the **if** and the **else** statements to be blocks of code. In this case, the target of **if** is a single statement, while the target of **else** is a block. Remember, you are free to use either a single statement or a code block at either place.

EXERCISES

1. Write a program that either adds or subtracts two integers. First, prompt the user to choose an operation; then prompt for the two numbers and display the result.
2. Is this fragment correct?

```
if(count < 100)
  printf("Number is less than 100.");
  printf("Its square is %d.", count * count);
}
```

USE THE for LOOP

The **for** loop is one of C's three loop statements. It allows one or more statements to be repeated. If you have programmed in any other computer language, such as BASIC or Pascal, you will be pleased to learn that the **for** behaves much like its equivalent in other languages.

The **for** loop is considered by many C programmers to be its most flexible loop. Although the **for** loop allows a large number of variations, we will examine only its most common form in this section.

The **for** loop is used to repeat a statement or block of statements a specified number of times. Its general form for repeating a single statement is shown here.

for(*initialization* ; *conditional-test* ; *increment*) *statement* ;

The *initialization* section is used to give an initial value to the variable that controls the loop. This variable is usually referred to as the *loop-control variable*. The initialization section is executed only once, before the loop begins. The *conditional-test* portion of the loop tests the loop-control variable against a target value. If the conditional test evaluates true, the loop repeats. If it is false, the loop stops, and program execution picks up with the next line of code that follows the loop. The conditional test is performed at the start or *top* of the loop each time the loop is repeated. The *increment* portion of the **for** is executed at the bottom of the loop. That is, the increment portion is executed after the statement or block that forms its *body* has been executed. The purpose of the increment portion is to increase (or decrease) the loop-control value by a certain amount.

As a simple first example, this program uses a **for** loop to print the numbers **1** through **10** on the screen.

```
#include <stdio.h>

int main(void)
{
  int num;

  for(num=1; num<11; num=num+1) printf("%d ", num);
  printf("terminating");

  return 0;
}
```

This program produces the following output:

```
1 2 3 4 5 6 7 8 9 10 terminating
```

The program works like this: First, the loop control variable **num** is initialized to 1. Next, the expression **num < 11** is evaluated. Since it is true, the **for** loop begins running. After the number is printed, **num** is incremented by one and the conditional test is evaluated again. This process continues until **num** equals 11. When this happens, the **for** loop stops, and **terminating** is displayed. Keep in mind that the initialization portion of the **for** loop is only executed once, when the loop is first entered.

As stated earlier, the conditional test is performed at the start of each iteration. This means that if the test is false to begin with, the loop will not execute even once. For example, this program only displays **terminating** because **num** is initialized to 11, causing the conditional test to fail.

```
#include <stdio.h>

int main(void)
{
  int num;

  /* this loop will not execute */
  for(num=11; num<11; num=num+1) printf("%d ", num);

  printf("terminating");

  return 0;
}
```

To repeat several statements, use a block of code as the target of the **for** loop. For example, this program computes the product and sum of the numbers from 1 to 5:

```
#include <stdio.h>

int main(void)
{
  int num, sum, prod;

  sum = 0;
  prod = 1;

  for(num=1; num<6; num=num+1) {
    sum = sum + num;
    prod = prod * num;
  }
  printf("product and sum: %d %d", prod, sum);

  return 0;
}
```

A **for** loop can run negatively. For example, this fragment decrements the loop-control variable.

```
for(num=20; num>0; num=num-1)...
```

Further, the loop-control variable may be incremented or decremented by more than one. For example, this program counts to 100 by fives:

```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=0; i<101; i=i+5) printf("%d ", i);

  return 0;
}
```

EXAMPLES

1. The addition-drill program created earlier can be enhanced using a **for** loop. The version shown here asks for the sums of the numbers between 1 and 10. That is, it asks for 1 + 1, then 2 + 2, and so on. This program would be useful to a first grader who is learning to add.

```
#include <stdio.h>

int main(void)
{
  int answer, count;

  for(count=1; count<11; count=count+1) {
    printf("What is %d + %d? ", count, count);
    scanf("%d", &answer);

    if(answer == count+count) printf("Right! ");
    else {
      printf("Sorry, you're wrong. ");
      printf("The answer is %d. ", count+count);
    }
  }

  return 0;
}
```

Notice that this program has an **if** statement as part of the **for** block. Notice further that the target of **else** is a block of code. This is perfectly valid. In C, a code block may contain statements that create other code blocks. Notice how the indentation adds clarity to the structure of the program.

2. We can use a **for** loop to create a program that determines if a number is prime. The following program asks the user to enter a number and then checks to see if it has any factors.

```
/* Prime number tester. */
#include <stdio.h>

int main(void)
{
  int num, i, is_prime;

  printf("Enter the number to test: ");
  scanf("%d", &num);

  /* now test for factors */
  is_prime = 1;
  for(i=2; i<=num/2; i=i+1)
    if((num%i)==0) is_prime = 0;

  if(is_prime==1) printf("The number is prime.");
  else printf("The number is not prime.");

  return 0;
}
```

EXERCISES

1. Create a program that prints the numbers from **1** to **100**.
2. Write a program that prints the numbers between **17** and **100** that can be evenly divided by 17.
3. Write a program similar to the prime-number tester, except that it displays all the factors of a number entered by the user. For example, if the user entered **8**, it would respond with **2** and **4**.

SUBSTITUTE C'S INCREMENT AND DECREMENT OPERATORS

When you learned about the **for** in the preceding section, the increment portion of the loop looked more or less like the one shown here:

```
for(num=0; num<some_value; num=num+1)...
```

Although not incorrect, you will almost never see a statement like **num = num + 1** in professionally written C programs because C provides a special operator that increments a variable by one. The *increment operator* is ++(two pluses with no intervening space). Using the increment operator, you can change this line of code:

```
i = i + 1;
```

into this:

```
i++;
```

Therefore, the **for** shown earlier will normally be written like this:

```
for(num=0; num<some_value; num++)...
```

In a similar fashion, to decrease a variable by one, you can use C's *decrement operator*: – –. (There must be no space between the two minus signs.) Therefore,

```
count = count - 1;
```

can be rewritten as

```
count--;
```

Aside from saving you a little typing effort, the reason you will want to use the increment and decrement operators is that, for most C compilers, they will be faster than the equivalent assignment statements. The reason for this difference is that the C compiler can often avoid separate load-and-store machine-language instructions and substitute a single increment or decrement instruction in the executable version of a program.

The increment and decrement operators do not need to follow the variable; they can precede it. Although the effect on the variable is the same, the position of the operator does affect when the operation is performed. To see how, examine this program:

```
#include <stdio.h>

int main(void)
{
  int i, j;

  i = 10;
  j = i++;

  /* this will print 11 10 */
  printf("i and j: %d %d", i, j);

  return 0;
}
```

Don't let the **j = i++** statement trouble you. The increment operator may be used as part of any valid C expression. This statement works like this. First, the current value of **i** is assigned to **j**. Then **i** is incremented. This is why **j** has the value 10, not 11. When the increment or decrement operator follows the variable, the operation is performed *after* its value has been obtained for use in the expression. Therefore, assuming that **max** has the value 1, an expression such as this:

```
count = 10 * max++;
```

assigns the value 10 to **count** and increases **max** by one.

If the variable is preceded by the increment or decrement operator, the operation is performed first, and then the value of the variable is obtained for use in the expression. For example, rewriting the previous program as follows causes **j** to be 11.

```
#include <stdio.h>

int main(void)
{
  int i, j;

  i = 10;
  j = ++i;
  /* this will print 11 11 */
  printf("i and j: %d %d", i, j);

  return 0;
}
```

If you are simply using the increment or decrement operators to replace equivalent assignment statements, it doesn't matter if the operator precedes or follows the variable. This is a matter of your own personal style.

EXAMPLES

1. Here is the addition drill program developed in Section 2. It has been rewritten using the increment operator.

```
#include <stdio.h>

int main(void)
{
  int answer, count;

  for(count=1; count<11; count++) {
    printf("What is %d + %d? ", count, count);
    scanf("%d", &answer);

    if(answer == count+count) printf("Right! ");
    else {
      printf("Sorry, you're wrong. ");
      printf("The answer is %d. ", count+count);
    }
  }

  return 0;
}
```

2. This program illustrates the use of the increment and decrement operators:

```
#include <stdio.h>

int main(void)
```

```
{
  int i;

  i = 0;

  i++;
  printf("%d ", i); /* prints 1 */
  i--;
  printf("%d ", i); /* prints 0 */

  return 0;
}
```

EXERCISES

1. Rewrite the answer to the **for** loop exercises in the previous section so that they use the increment or decrement operators.
2. Change all appropriate assignment statements in this program to increment or decrement statements.

```
#include <stdio.h>

int main(void)
{
  int a, b;

  a = 1;

  a = a + 1;

  b = a;

  b = b - 1;

  printf("%d %d", a, b);

  return 0;
}
```

EXPAND printf( )'S CAPABILITIES

So far, we have used **printf( )** to output strings and numbers. However, you might have been wondering how to tell **printf( )** that you want the output to advance to the next line. The way to accomplish this and other actions is to use C's *backslash-character constants*. The C language defines several special character codes, shown in Table 2-1, which represent characters that cannot be entered from the keyboard, are non-printing characters, may not be found in all character sets, or that serve other unique needs. You can use the backslash codes anywhere you can use a normal character. The backslash constants are also referred to as *escape sequences*.

Code	Meaning
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\"	Double quote
\'	Single quote
\0	Null
\\	Backslash
\v	Vertical tab
\a	Alert
\?	Question mark
\N	Octal constant (where *N* is an octal value)
\x*N*	Hexadecimal constant (where *N* is a hexidecimal value)

TABLE 2-1 *C's Backslash Codes* ▼

Perhaps the single most important backslash code is **\n**, which is often referred to as the *newline character*. When the C compiler encounters **\n**, it translates it into a carriage return/linefeed combination. For example, this program:

```
#include <stdio.h>

int main(void)
{
  printf("This is line one.\n");
  printf("This is line two.\n");
  printf("This is line three.");

  return 0;
}
```

displays

```
This is line one.
This is line two.
This is line three.
```

on the screen.

Remember, the backslash codes are character constants. Therefore, to assign one to a character variable, you must enclose the backslash code within single quotes, as shown in this fragment:

```
char ch;

ch = '\t'; /* assign ch the tab character */
```

EXAMPLES

1. This program sounds the bell:

```
#include <stdio.h>

int main(void)
{
  printf("\a");
  return 0;
}
```

2. You can enter any special character by specifying it as an octal or hexadecimal value following the backslash. The octal number system is based on 8 and uses the digits 0 through 7. In octal, the number 10 is the same as 8 in decimal. The hexadecimal number system is based on 16 and uses the digits 0 through 9 plus the letters 'A' through 'F', which stand for 10, 11, 12, 13, 14, and 15. For example, the hexadecimal number 10 is 16 in decimal. When specifying a character in hexadecimal, you must follow the backslash with an 'x', followed by the number. The ASCII character set is defined from 0 to 127. However, many computers, including most PCs, use the values 128 to 255 for special and graphics characters. If your computer supports these extra characters, the following program will display a few of them on the screen.

```
#include <stdio.h>

int main(void)
{
  printf("\xA0 \xA1 \xA2 \xA3");

  return 0;
}
```

3. The **\n** newline character does not have to go at the end of the string that is being output by **printf( )**; it can go anywhere in the string. Further, there can be as many newline characters in a string as you desire. The point is that there is no connection between a newline and the end of a string. For example, this program:

```
#include <stdio.h>

int main(void)
{
  printf("one\ntwo\nthree\nfour");

  return 0;
}
```

displays

```
one
two
three
four
```

on the screen.

EXERCISES

1. Write a program that outputs a table of numbers. Each line in the table contains three entries: the number, its square, and its cube. Begin with **1** and end with **10**. Also, use a **for** loop to generate the numbers.
2. Write a program that prompts the user for an integer value. Next, using a **for** loop, make it count down from this value to 0, displaying each number on its own line. When it reaches 0, have it sound the bell.
3. Experiment on your own with the backslash codes.

PROGRAM WITH C'S RELATIONAL AND LOGICAL OPERATORS

The C language contains a rich set of operators. In this section you will learn about C's relational and logical operators. As you saw earlier, the relational operators compare two values and return a true or false result based upon that comparison. The logical operators connect together true/false results. These operators are shown in Table 2-2 and Table 2-3.

Operator	Action
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Equal
!=	Not equal

TABLE 2-2 *Relational Operators* ▼

The logical operators are used to support the basic logical operations of AND, OR, and NOT according to this truth table. The table uses 1 for true and 0 for false.

p	q	p&&q	p||q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

The relational and logical operators are both lower in precedence than the arithmetic operators. This means that an expression like

```
10 + count > a + 12
```

Operator	Action
&&	AND
||	OR
!	NOT

TABLE 2-3 *Logical Operators* ▼

is evaluated as if it were written

```
(10 + count) > (a + 12)
```

You may link any number of relational operations together using logical operators. For example, this expression joins three relational operations.

```
var > max || !(max==100) && 0 <= item
```

The table below shows the relative precedence of the relational and logical operators.

Highest	!
	> >= < <=
	== !=
	&&
Lowest	||

There is one important fact to remember about the values produced by the relational and logical operators: the result is either 0 or 1. Even though C defines true as any nonzero value, the relational and logical operators always produce the value 1 for true. Your programs may make use of this fact.

You can use the relational and logical operators in both the **if** and **for** statements. For example, the following statement reports when both **a** and **b** are positive:

```
if(a>0 && b>0) printf("Both are positive.");
```

EXAMPLES

1. In professionally written C code, it is uncommon to find a statement like this:

   ```
   if(count != 0)...
   ```

   The reason is that in C, true is any nonzero value and false is zero. Therefore, the preceding statement is generally written as this:

   ```
   if(count)...
   ```

Further, statements like this:

```
if(count == 0)...
```

are generally written as:

```
if(!count)...
```

The expression **!count** is true only if **count** is zero.

2. It is important to remember that the outcome of a relational or logical operation is 0 when false and 1 when true. For example, the following program requests two integers, then displays the outcome of each relational and logical operation when applied to them. In all cases, the result will be a 0 or a 1.

```
#include <stdio.h>

int main(void)
{
  int i, j;

  printf("Enter first number: ");
  scanf("%d", &i);
  printf("Enter second number: ");
  scanf("%d", &j);

  /* relational operations */
  printf("i < j %d\n", i < j);
  printf("i <= j %d\n", i <= j);
  printf("i == j %d\n", i == j);
  printf("i > j %d\n", i > j);
  printf("i >= j %d\n", i >= j);

  /* logical operations */
  printf("i && j %d\n", i && j);
  printf("i || j %d\n", i || j);
  printf("!i !j %d %d\n", !i, !j);

  return 0;
}
```

3. C does not define an exclusive-OR (XOR) logical operator. However, it is easy to create a function that performs the operation. The XOR operation uses this truth table:

p	**q**	**XOR**
0	0	0
0	1	1
1	0	1
1	1	0

That is, the XOR operation produces a true result when one and only one operand is true. The following function uses the ***&&*** and **||** operators to construct an XOR operation. It compares the values of its two arguments and returns the outcome of an XOR operation.

```
int xor(int a, int b)
{
  return (a || b) && !(a && b);
}
```

The following program uses this function. It displays the result of an AND, OR, and XOR on the values you enter.

```
/* This program demonstrates the xor() function. */
#include <stdio.h>

int xor(int a, int b);

int main(void)
{
  int p, q;

  printf("enter P (0 or 1): ");
  scanf("%d", &p);
  printf("enter Q (0 or 1): ");
  scanf("%d", &q);
  printf("P AND Q: %d\n", p && q);
  printf("P OR Q: %d\n", p || q);
  printf("P XOR Q: %d\n", xor(p, q));

  return 0;
}

int xor(int a, int b)
```

```
{
  return (a || b) && !(a && b);
}
```

EXERCISES

1. What does this loop do?

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

2. Is this expression true?

   ```
   !(10==9)
   ```

3. Do these two expressions evaluate to the same outcome?

   a. 0 && 1 || 1

   b. 0 && (1 || 1)

4. On your own, experiment with the relational and logical operators.

Mastery Skills Check

1. Write a program that plays a computerized form of the "guess the magic number" game. It works like this: The player has ten tries to guess the magic number. If the number entered is the value you have selected for your magic number, have the program print the message "RIGHT!" and then terminate. Otherwise, have the program report whether the guess was high or low and then let the player enter another number. This process goes on until the player guesses the number or the ten tries have been used up. For fun, you might want to report the number of tries it takes to guess the number.

2. Write a program that computes the square footage of a house given the dimensions of each room. Have the program ask the user how many rooms are in the house and then request the dimensions of each room. Display the resulting total square footage.
3. What are the increment and decrement operators and what do they do?
4. Create an improved addition-drill program that keeps track of the number of right and wrong answers and displays them when the program ends.
5. Write a program that prints the numbers **1** to **100** using 5 columns. Have each number separated from the next by a tab.

3

More C Program Control Statements

chapter objectives

3.1 Input characters

3.2 Nest **if** statements

3.3 Examine **for** loop variations

3.4 Understand C's **while** loop

3.5 Use the **do** loop

3.6 Create nested loops

3.7 Use **break** to exit a loop

3.8 Know when to use the **continue** statement

3.9 Select among alternatives with the **switch** statement

3.10 Understand the **goto** statement

THIS chapter continues the discussion of C's program control statements. Before doing so, however, the chapter begins by explaining how to read characters from the keyboard. Although you know how to input numbers, it is now time for you to know how to input individual characters because several examples in this chapter will make use of them. Next, the chapter finishes the discussion of the **if** and **for** statements. Then it presents C's two other loop statements, the **while** and **do**. Next you will learn about nested loops and two more of C's control statements, the **break** and **continue**. This chapter also covers C's other selection statement, the **switch**. It ends with a short discussion of C's unconditional jump statement, **goto**.

Review Skills Check

Before proceeding, you should be able to answer these questions and perform these exercises:

1. What are C's relational and logical operators?
2. What is a block of code? How do you make one?
3. How do you output a newline using **printf( )**?
4. Write a program that prints the numbers **-100** to **100**.
5. Write a program that prints 5 different proverbs. The program prompts the user for the number of the proverb to print and then displays it. (Use any proverbs you like.)
6. How can this statement be rewritten?

```
count = count + 1;
```

7. What values are true in C? What values are false?

3.1 INPUT CHARACTERS

Although numbers are important, your programs will also need to read characters from the keyboard. In C you can do this in a variety of ways. Unfortunately, this conceptually simple task is complicated by some baggage left over from the origins of C. However, let's begin with the traditional way characters are read from the keyboard. Later you will learn an alternative.

C defines a function called **getchar( )**, which returns a single character typed on the keyboard. When called, the function waits for a key to be pressed. Then **getchar( )** echoes the keystroke to the screen and returns the value of the key to the caller. The **getchar( )** function is defined by the ANSI C standard and requires the header file STDIO.H. This program illustrates its use by reading a character and then telling you what it received. (Remember, to display a character, use the **%c** **printf( )** format specifier.)

```
#include <stdio.h>

int main(void)
{
  char ch;

  ch = getchar(); /* read a char */
  printf(" you typed: %c", ch);

  return 0;
}
```

If you try this program, it may behave differently than you expected. The trouble is this: in many C compilers, **getchar( )** is implemented in such a way that it *line buffers* input. That is, it does not immediately return as soon as you have pressed a key, but waits until you have entered an entire line, which may include several other characters. This means that even though it will read and return only one character, **getchar( )** waits until you enter a carriage return (i.e., press ENTER) before doing so. When **getchar( )** returns, it will return the first character you typed. However, any other characters that you entered, including the carriage return, will still be in the input buffer. These characters will be consumed by subsequent input requests, such as through calls to **scanf( )**. In some circumstances, this can lead to trouble. This situation is examined more closely in Chapter 8. For now, just be aware that **getchar( )** may behave differently than your intuition would suggest. Of course, the programs shown in this book behave properly.

The reason that **getchar( )** works the way it does is that the version of UNIX for which C was developed line-buffered input. When C compilers were created for other interactive environments, developers had to decide how to make **getchar( )** behave. Many C compiler developers have decided, for the sake of compatibility, to keep **getchar( )** line-buffered, even though there is no technical reason for it. (In fact, the ANSI C standard states that **getchar( )** need not be line-buffered.) When **getchar( )** is implemented in a line-buffered fashion in a modern interactive environment, its use is severely limited.

Because many compilers have implemented line-buffered versions of **getchar( )**, most C compilers supply another function to perform interactive console input. Although it is not defined by the ANSI C standard, most compilers call this function **getche( )**. You use it just like **getchar( )**, except that it will return its value immediately after a key is pressed; it does not line-buffer input. For most compilers, this function requires a header file called CONIO.H, but it might be called something different in your compiler. Thus, if you want to achieve interactive character input, you will usually need to use the **getche( )** function rather than **getchar( )**.

Since all readers will have access to the **getchar( )** function, it will be used by most of the examples in this book that require character input. However, some examples will use the **getche( )** function. If your compiler does not include this function, substitute **getchar( )**. You should feel free to experiment with **getche( )** on your own.

Note

*At the time of this writing, when using Microsoft's Visual C++ compiler, **getche()** is not compatible with C's standard input functions, such as **scanf()**. Instead, you must use special console versions of these of these functions, such as **cscanf()**. This and other non-standard I/O functions are described in Chapter 8. The examples in this book that use **getche()** work correctly with Visual C++ because they avoid the use of the standard input functions.*

Virtually all computers use the ASCII character codes when representing characters. Therefore, characters returned by either **getchar( )** or **getche( )** will be represented by their ASCII codes. This is useful because the ASCII character codes are an ordered sequence; each letter's code is one greater than the previous letter; each digit's code is one greater than the previous digit. This means that 'a' is less than 'b', '2' is less than '3', and so on. You may compare characters just like you compare numbers. For example,

```
ch = getchar();
if(ch < 'f' printf("character is less than f");
```

is a perfectly valid fragment that will display its message if the user enters any character that comes before **f**.

EXAMPLES

1. This program reads a character and displays its ASCII code. This illustrates an important feature of C: You can use a character as if it were a "little integer." The program also demonstrates the use of the **getche( )** function.

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

int main(void)
{
  char ch;

  printf("Enter a character: ");
  ch = getche();
  printf("\nIts ASCII code is %d", ch);

  return 0;
}
```

   Because this program uses **getche( )**, it responds as soon as you press a key. Before continuing, try substituting **getchar( )** for **getche( )** in this program and observe the results. As you will see, **getchar( )** does not return a character to your program until you press ENTER.

2. One of the most common uses of character input is to obtain a menu selection. For example, this program allows the user to add, subtract, multiply, or divide two numbers.

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

```
int main(void)
{
  int a, b;
  char ch;

  printf("Do you want to:\n");
  printf("Add, Subtract, Multiply, or Divide?\n");
  printf("Enter first letter: ");
  ch = getchar();
  printf("\n");

  printf("Enter first number: ");
  scanf("%d", &a);
  printf("Enter second number: ");
  scanf("%d", &b);

  if(ch=='A') printf("%d", a+b);
  if(ch=='S') printf("%d", a-b);
  if(ch=='M') printf("%d", a*b);
  if(ch=='D' && b!=0) printf("%d", a/b);

  return 0;
}
```

One point to keep in mind is that C makes a distinction between upper- and lowercase letters. So, if the user enters an **s**, the program will not recognize it as a request to subtract. (Later, you will learn how to convert the case of a character.)

3. Another common reason that your program will need to read a character from the keyboard is to obtain a yes/no response from the user. For example, this fragment determines if the user wants to proceed.

```
printf("Do you wish to continue? (Y/N : ");
ch = getche();
if(ch=='Y') {
  /* continue with something */
  .
  .
  .
}
```

EXERCISES

1. Write a program that reads ten letters. After the letters have been read, display the one that comes earliest in the alphabet. (Hint: The one with the smallest value comes first.)
2. Write a program that displays the ASCII codes for the characters A through Z and a through z. How do the codes differ between the upper- and lowercase characters?

3.2 NEST if STATEMENTS

When an **if** statement is the target of another **if** or **else**, it is said to be *nested* within the outer **if**. Here is a simple example of a nested **if**:

```
if(count > max) /* outer if */
  if(error) printf("Error, try again."); /* nested if */
```

Here, the **printf( )** statement will only execute if **count** is greater than **max** and if **error** is nonzero. Notice how the nested **if** is indented. This is common practice. It enables anyone reading your program to know quickly that the **if** is nested and what actions are nested. A nested **if** may also appear inside a block of statements that are the target of the outer **if**.

An ANSI-standard compiler will allow you to nest **if**s at least 15 levels deep. (However, it would be rare to find such a deep nesting.)

One confusing aspect of nested **if**s is illustrated by the following fragment:

```
if(p)
  if(q) printf("a and b are true");
else printf("To which statement does this else apply?");
```

The question suggested by the second **printf( )** is: which **if** is associated with the **else**? Fortunately, the answer is quite easy: An **else** always associates with the nearest **if** in the same block that does not already have an **else** associated with it. In this example, the **else** is associated with the second **if**.

EXAMPLES

1. It is possible to string together several **if**s and **else**s into what is sometimes called an *if-else-if ladder* or *if-else-if staircase* because of its visual appearance. In this situation a nested **if** has as its target another **if**. The general form of the if-else-if ladder is shown here:

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



          else statement;
```

The expressions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed, and the rest of the ladder is bypassed. If none of the expressions are true, the final **else** will be executed. That is, if all other conditional tests fail, the last **else** statement is performed. If the final **else** is not present, no action will take place if all expressions are false.

Although the indentation of the general form of the if-else-if ladder just shown is technically correct, it can lead to overly deep indentation. Because of this, the if-else-if ladder is generally written like this:

```
if(expression) statement;
else if(expression) statement;
else if(expression) statement;
.
.
.
else statement;
```

We can improve the arithmetic program developed in Section 3.1 by using an if-else-if ladder, as shown here:

```
#include <stdio.h>

int main(void)
{
  int a, b;
  char ch;

  printf("Do you want to:\n");
  printf("Add, Subtract, Multiply, or Divide?\n");
  printf("Enter first letter: ");
  ch = getchar();
  printf("\n");

  printf("Enter first number: ");
  scanf("%d", &a);
  printf("Enter second number: ");
  scanf("%d", &b);

  if(ch=='A') printf("%d", a+b);
  else if(ch=='S') printf("%d", a-b);
  else if(ch=='M') printf("%d", a*b);
  else if(ch=='D' && b!=0) printf("%d", a/b);

  return 0;
}
```

This is an improvement over the original version because once a match is found, any remaining **if** statements are skipped. This means that the program isn't wasting time on needless operations. While this is not too important in this example, you will encounter situations where it will be.

2. Nested **if** statements are very common in programming. For example, here is a further improvement to the addition drill program developed in the preceding chapter. It lets the user have a second try at getting the right answer.

```
#include <stdio.h>

int main(void)
{
  int answer, count;
  int again;

  for(count=1; count<11; count++) {
    printf("What is %d + %d? ", count, count);
    scanf("%d", &answer);

    if(answer == count+count) printf("Right!\n");
    else {
      printf("Sorry, you're wrong\n");
      printf("Try again.\n ");

      printf("\nWhat is %d + %d? ", count, count);
      scanf("%d", &answer);

      /* nested if */
      if(answer == count+count) printf("Right!\n");
      else
        printf("Wrong, the answer is %d\n",
                count+count);
    }
  }
  return 0;
}
```

Here, the second **if** is nested within the outer **if**'s **else** block.

EXERCISES

1. To which **if** does the **else** relate to in this example?

```
if(ch=='S') { /* first if */
  printf("Enter a number: ");
  scanf("%d", &y);

  /* second if */
  if(y) printf("Its square is %d.", y*y);
}
else printf("Make next selection.");
```

2. Write a program that computes the area of either a circle, rectangle, or triangle. Use an if-else-if ladder.

3.3 EXAMINE for LOOP VARIATIONS

The **for** loop in C is significantly more powerful and flexible than in most other computer languages. When you were introduced to the **for** loop in Chapter 2, you were only shown the form similar to that used by other languages. However, you will see that **for** is much more flexible.

The reason that **for** is so flexible is that the expressions we called the *initialization*, *conditional-test*, and *increment* portions of the loop are not limited to these narrow roles. The **for** loop places no limits on the types of expressions that occur inside it. For example, you do not have to use the initialization section to initialize a loop-control variable. Further, there does not need to be any loop-control variable because the conditional expression may use some other means of stopping the loop. Finally, the increment portion is technically just an expression that is evaluated each time the loop iterates. It does not have to increment or decrement a variable.

Another important reason that the **for** is so flexible is that one or more of the expressions inside it may be empty. For example, if the loop-control variable has already been initialized outside the **for**, there is no need for an initialization expression.

EXAMPLES

1. This program continues to loop until a **q** is entered at the keyboard. Instead of testing a loop-control variable, the conditional test in this **for** checks the value of a character entered by the user.

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

int main(void)
{
  int i;
  char ch;

  ch = 'a'; /* give ch an initial value */

  for(i=0; ch != 'q'; i++) {
    printf("pass: %d\n", i);
    ch = getche();
  }

  return 0;
}
```

Here, the condition that controls the loop has nothing to do with the loop-control variable. The reason **ch** is given an initial value is to prevent it from accidentally containing a **q** when the program begins.

2. As stated earlier, it is possible to leave an expression in a loop empty. For example, this program asks the user for a value and then counts down to zero from this number. Here, the loop-control variable is initialized by the user outside the loop, so the initialization portion of the loop is empty.

```
#include <stdio.h>

int main(void)
{
  int i;

  printf("Enter an integer: ");
  scanf("%d", &i);

  for(; i; i--) printf("%d ", i);

  return 0;
}
```

3. Another variation to **for** is that its target may be empty. For example, this program simply keeps inputting characters until the user types **q**.

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

int main(void)
{
  char ch;

  for(ch=getche(); ch!='q'; ch=getche());
  printf("Found the q.");
  return 0;
}
```

Notice that the statements assigning **ch** a value have been moved into the loop. This means that when the loop starts, **getche( )** is called. Then, the value of **ch** is tested against **q**. Next, conceptually, the nonexistent target of the **for** is executed, and the call to **getche( )** in the increment portion of the loop is executed. This process repeats until the user enters a **q**.

The reason the target of the **for** can be empty is because C allows null statements.

4. Using the **for**, it is possible to create a loop that never stops. This type of loop is usually called an *infinite loop*. Although accidentally creating an infinite loop is a bug, you will sometimes want to create one on purpose. (Later in this chapter, you will see that there are ways to exit even an infinite loop!) To create an infinite loop, use a **for** construct like this:

```
for( ; ; ) {
  .
  .
  .
}
```

As you can see, there are no expressions in the **for**. When there is no expression in the conditional portion, the compiler assumes that it is true. Therefore, the loop continues to run.

5. In C, unlike most other computer languages, it is perfectly valid for the loop-control variable to be altered outside the increment section. For example, the following program manually increments **i** at the bottom of the loop.

```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=0; i<10; ) {
    printf("%d ", i);
    i++;
  }
```

```
    return 0;
}
```

EXERCISES

1. Write a program that computes driving time when given the distance and the average speed. Let the user specify the number of drive time computations he or she wants to perform.
2. To create time-delay loops, **for** loops with empty targets are often used. Create a program that asks the user for a number and then iterates until zero is reached. Once the countdown is done, sound the bell, but don't display anything on the screen.
3. Even if a **for** loop uses a loop-control variable, it need not be incremented or decremented by a fixed amount. Instead, the amount added or subtracted may vary. Write a program that begins at 1 and runs to 1000. Have the program add the loop-control variable to itself inside the increment expression. This is an easy way to produce the arithmetic progression 1 2 4 8 16, and so on.

UNDERSTAND C'S while LOOP

Another of C's loops is **while**. It has this general form:

while(*expression*) *statement*;

Of course, the target of **while** may also be a block of code. The **while** loop works by repeating its target as long as the expression is true. When it becomes false, the loop stops. The value of the expression is checked at the top of the loop. This means that if the expression is false to begin with, the loop will not execute even once.

EXAMPLES

1. Even though the **for** is flexible enough to allow itself to be controlled by factors not related to its traditional use, you should generally select the loop that best fits the needs of the situation. For example, a better way to wait for the letter **q** to be typed is shown here using **while**. If you compare it to Example 3 in Section 3.3, you will see how much clearer this version is. (However, you will soon see that a better loop for this job exists.)

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

int main(void)
{
  char ch;

  ch = getche();

  while(ch!='q') ch = getche();
  printf("Found the q.");

  return 0;
}
```

2. The following program is a simple code machine. It translates the characters you type into a coded form by adding 1 to each letter. That is, 'A' becomes 'B,' and so forth. The program stops when you press ENTER. (The **getche( )** function returns **\r** when ENTER is pressed.)

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

int main(void)
{
  char ch;

  printf("Enter your message.\n");

  ch = getche();
  while(ch != '\r') {
      printf("%c", ch+1);
      ch = getche();
    }

    return 0;
  }
```

EXERCISES

1. In Exercise 1 of Section 3.3, you created a program that computed driving time, given distance and average speed. You used a **for** loop to let the user compute several drive times. Rework that program so that it uses a **while** loop.
2. Write a program that will decode messages that have been encoded using the code machine program in the second example in this section.

USE THE do LOOP

C's final loop is **do**, which has this general form:

```
do {
  statements
} while(expression);
```

If only one statement is being repeated, the curly braces are not necessary. Most programmers include them, however, so that they cai easily recognize that the **while** that ends the **do** is part of a **do** loop, not the beginning of a **while** loop.

The **do** loop repeats the statement or statements while the expression is true. It stops when the expression becomes false. The **do** loop is unique because it will always execute the code within the loop at least once, since the expression controlling the loop is tested at the bottom of the loop.

EXAMPLES

1. The fact that **do** will always execute the body of its loop at least once makes it perfect for checking menu input. For example, this version of the arithmetic program reprompts the user until a valid response is entered.

```
#include <stdio.h>

int main(void)
{
  int a, b;
  char ch;

  printf("Do you want to:\n");
  printf("Add, Subtract, Multiply, or Divide?\n");

  /* force user to enter a valid response */
  do {
    printf("Enter first letter: ");
    ch = getchar();
  } while(ch!='A' && ch!='S' && ch!='M' && ch!='D');
  printf("\n");

  printf("Enter first number: ");
  scanf("%d", &a);
  printf("Enter second number: ");
  scanf("%d", &b);

  if(ch=='A') printf("%d", a+b);
  else if(ch=='S') printf("%d", a-b);
  else if(ch=='M') printf("%d", a*b);
  else if(ch=='D' && b!=0) printf("%d", a/b);

  return 0;
}
```

2. The **do** loop is especially useful when your program is waiting for some event to occur. For example, this program waits for the user to type a **q**. Notice that it contains one less call to **getche( )** than the equivalent program described in the section on the **while** loop.

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

```
int main(void)
{
  char ch;

  do {
    ch = getche();
  } while(ch!='q');

  printf("Found the q.");

  return 0;
}
```

Since the loop condition is tested at the bottom, it is not necessary to initialize **ch** prior to entering the loop.

EXERCISES

1. Write a program that converts gallons to liters. Using a **do** loop, allow the user to repeat the conversion. (One gallon is approximately 3.7854 liters.)
2. Write a program that displays the menu below and uses a **do** loop to check for valid responses. (Your program does not need to implement the actual functions shown in the menu.)

   Mailing list menu:

   1. Enter addresses
   2. Delete address
   3. Search the list
   4. Print the list
   5. Quit

   Enter the number of your choice (1-5).

CREATE NESTED LOOPS

When the body of one loop contains another, the second is said to be nested inside the first. Any of C's loops may be nested within any other loop. The ANSI C standard specifies that loops may be nested at least 15 levels deep. However, most compilers allow nesting to virtually any level. As a simple example of nested **for**s, this fragment prints the numbers **1** to **10** on the screen ten times.

```
for(i=0; i<10; i++) {
  for(j=1; j<11; j++) printf("%d ", j); /* nested loop */
  printf("\n");
}
```

EXAMPLES

1. You can use a nested **for** to make another improvement to the arithmetic drill. In the version shown below, the program will give the user three chances to get the right answer. Notice the use of the variable **right** to stop the loop early if the correct answer is given.

```
#include <stdio.h>

int main(void)
{
  int answer, count, chances, right;

  for(count=1; count<11; count++) {
    printf("What is %d + %d?", count, count);
    scanf("%d", &answer);

    if(answer == count+count) printf("Right!\n");
    else {
      printf("Sorry, you're wrong.\n");
      printf("Try again.\n");

      right = 0;
```

```
        /* nested for */
        for(chances=0; chances<3 && !right; chances++) {
          printf("What is %d + %d? ", count, count);
          scanf("%d", &answer);

          if(answer == count+count) {
            printf("Right!\n");
            right = 1;
          }
        }

        /* if answer still wrong, tell user */
        if(!right)
          printf("The answer is %d.\n", count+count);
      }
    }

    return 0;
  }
```

2. This program uses three **for** loops to print the alphabet three times, each time printing each letter twice:

```
#include <stdio.h>

int main(void)
{
  int i, j, k;
  for(i=0; i<3; i++)
    for(j=0; j<26; j++)
      for(k=0; k<2; k++) printf("%c", 'A'+j);

  return 0;
}
```

The statement

```
printf("%c", 'A'+j);
```

works because ASCII codes for the letters of the alphabet are strictly ascending—each one is greater than the letter that precedes it.

EXERCISES

1. Write a program that finds all the prime numbers between **2** and **1000**.
2. Write a program that reads ten characters from the keyboard. Each time a character is read, use its ASCII code value to output a string of periods equal in number to this code. For example, given the letter 'A', whose code is 65, your program would output 65 periods.

USE **break** *TO EXIT A LOOP*

The **break** statement allows you to exit a loop from any point within its body, bypassing its normal termination expression. When the **break** statement is encountered inside a loop, the loop is immediately stopped, and program control resumes at the next statement following the loop. For example, this loop prints only the numbers **1** to **10**:

```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=1; i<100; i++) {
    printf("%d ", i);
    if(i==10) break; /* exit the loop */
  }

  return 0;
}
```

The **break** statement can be used with all three of C's loops.

You can have as many **break** statements within a loop as you desire. However, since too many exit points from a loop tend to destructure your code, it is generally best to use the **break** for special purposes, not as your normal loop exit.

EXAMPLES

1. The **break** statement is commonly used in loops in which a special condition can cause immediate termination. Here is an example of such a situation. In this case, a keypress can stop the execution of the program.

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

int main(void)
{
  int i;
  char ch;

  /* display all numbers that are multiples of 6 */
  for(i=1; i<10000; i++) {
    if(!(i%6)) {
      printf("%d, more? (Y/N)", i);
      ch = getche();
      if(ch=='N') break; /* stop the loop */
      printf("\n");
    }
  }

  return 0;
}
```

2. A **break** will cause an exit from only the innermost loop. For example, this program prints the numbers **0** to **5** five times:

```
#include <stdio.h>

int main(void)
{
  int i, j;

  for(i=0; i<5; i++) {
    for(j=0; j<100; j++) {
      printf("%d", j);
      if(j==5) break;
    }
    printf("\n");
  }

  return 0;
}
```

3. The reason C includes the **break** statement is to allow your programs to be more efficient. For example, examine this fragment:

```
do {
  printf("Load, Save, Edit, Quit?\n");
  do {
    printf("Enter your selection: ");
    ch = getchar();
  } while(ch!='L' && ch!='S' && ch!='E' && ch!='Q');

  if(ch != 'Q') {
    /* do something */
  }

  if(ch != 'Q') {
    /* do something else*/
  }
  /* etc. */
} while(ch != 'Q')
```

In this situation, several additional tests are performed on **ch** to see if it is equal to 'Q' to avoid executing certain sections of code when the **Quit** option is selected. Most C programmers would write the preceding loop as shown here:

```
for( ; ; ) { /* infinite for loop */
  printf("Load, Save, Edit, Quit?\n");
  do {
    printf ("Enter your selection: ");
    ch = getchar();
  } while(ch!='L' && ch!='S' && ch!='E' && ch!='Q');

  if(ch == 'Q') break;

  /* do something */
  /* do something else */
  /* etc. */
}
```

In this version, **ch** need only be tested once to see if it contains a 'Q'. As you can see, this implementation is more efficient because only one **if** statement is required.

EXERCISES

1. On your own, write several short programs that use **break** to exit a loop. Be sure to try all three loop statements.
2. Write a program that prints a table showing the proper amount of tip to leave. Start the table at $1 and stop at $100, using increments of $1. Compute three tip percentages: 10%, 15%, and 20%. After each line, ask the user if he or she wants to continue. If not, use **break** to stop the loop and end the program

KNOW WHEN TO USE THE continue STATEMENT

The **continue** statement is somewhat the opposite of the **break** statement. It forces the next iteration of the loop to take place, skipping any code in between itself and the test condition of the loop. For example, this program never displays any output:

```
#include <stdio.h>

int main(void)
{
  int x;

  for(x=0; x<100; x++) {
    continue;
    printf("%d ", x); /* this is never executed */
  }
  return 0;
}
```

Each time the **continue** statement is reached, it causes the loop to repeat, skipping the **printf( )** statement.

In **while** and **do-while** loops, a **continue** statement will cause control to go directly to the test condition and then continue the looping process. In the case of **for**, the increment part of the loop is performed, the conditional test is executed, and the loop continues.

Frankly, **continue** is seldom used, not because it is poor practice to use it, but simply because good applications for it are not common.

EXAMPLE

1. One good use for **continue** is to restart a statement sequence when an error occurs. For example, this program computes a running total of numbers entered by the user. Before adding a value to the running total, it verifies that the number was correctly entered by having the user enter it a second time. If the two numbers don't match, the program uses **continue** to restart the loop.

```
#include <stdio.h>

int main(void)
{
  int total, i, j;

  total = 0;
  do {
    printf("Enter next number (0 to stop): ");
    scanf("%d", &i);
    printf("Enter number again: ");
    scanf("%d", &j);
    if(i != j) {
      printf("Mismatch\n");
      continue;
    }
    total = total + i;
    } while(i);

    printf("Total is %d\n", total);

    return 0;
  }
```

EXERCISE

1. Write a program that prints only the odd numbers between **1** and **100**. Use a **for** loop that looks like this:

   ```
   for(i=1; i<101; i++) . . .
   ```

   Use a **continue** statement to avoid printing even numbers.

SELECT AMONG ALTERNATIVES WITH THE switch STATEMENT

While **if** is good for choosing between two alternatives, it quickly becomes cumbersome when several alternatives are needed. C's solution to this problem is the **switch** statement. The **switch** statement is C's multiple selection statement. It is used to select one of several alternative paths in program execution and works as follows. A value is successively tested against a list of integer or character constants. When a match is found, the statement sequence associated with that match is executed. The general form of the **switch** statement is this:

```
switch(value) {
  case constant1:
    statement sequence
    break;
```

```
  case constant2:
    statement sequence
    break;
  case constant3:
    statement sequence
    break;
  .
  .
  .
  default:
    statement sequence
    break;
}
```

The **default** statement sequence is performed if no matches are found. The **default** is optional. If all matches fail and **default** is absent, no action takes place. When a match is found, the statements associated with that **case** are executed until **break** is encountered or, in the case of **default** or the last **case**, the end of the **switch** is reached.

As a very simple example, this program recognizes the numbers 1, 2, 3, and 4 and prints the name of the one you enter. That is, if you enter **2**, the program displays **two**.

```
#include <stdio.h>

int main(void)
{
  int i;

  printf("Enter a number between 1 and 4: ");
  scanf("%d", &i);

  switch(i) {
    case 1:
      printf("one");
      break;
    case 2:
      printf("two");
      break;
    case 3:
      printf("three");
      break;
    case 4:
      printf("four");
      break;
    default:
      printf("Unrecognized Number");
  }

  return 0;
}
```

The **switch** statement differs from **if** in that **switch** can only test for equality, whereas the **if** conditional expression can be of any type. Also, **switch** will work with only **int** or **char** types. You cannot, for example, use floating-point numbers.

The statement sequences associated with each **case** are *not* blocks; they are not enclosed by curly braces.

The ANSI C standard states that at least 257 **case** statements will be allowed. In practice, you should usually limit the amount of **case** statements to a much smaller number for efficiency reasons. Also, no two **case** constants in the same **switch** can have identical values.

It is possible to have a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested switch*. If the **case** constants of the inner and outer **switch** contain common values, no conflicts will arise. For example, the following code fragment is perfectly acceptable:

```
switch(a) {
  case 1:
    switch(b) {
      case 0: printf("b is false");
              break;
      case 1: printf("b is true");
    }
    break;
  case 2:
    .
    .
    .
```

An ANSI-standard compiler will allow at least 15 levels of nesting for **switch** statements.

EXAMPLES

1. The **switch** statement is often used to process menu commands. For example, the arithmetic program can be recoded as shown here. This version reflects the way professional C code is written.

```
#include <stdio.h>

int main(void)
{
  int a, b;
  char ch;

  printf("Do you want to:\n");
  printf("Add, Subtract, Multiply, or Divide?\n");
  /* force user to enter a valid response */
  do {
    printf("Enter first letter: ");
    ch = getchar();
  } while(ch!='A' && ch!='S' && ch!='M' && ch!='D');
  printf("\n");

  printf("Enter first number: ");
  scanf("%d", &a);
  printf("Enter second number: ");
  scanf("%d", &b);

  switch(ch) {
    case 'A': printf("%d", a+b);
      break;
    case 'S': printf("%d", a-b);
      break;
    case 'M': printf("%d", a*b);
      break;
    case 'D': if(b!=0) printf("%d", a/b);
  }

  return 0;
}
```

2. Technically, the **break** statement is optional. The **break** statement, when encountered within a **switch**, causes the program flow to exit from the entire **switch** statement and continue on to the next statement outside the **switch**. This is much the way it works when breaking out of a loop. However, if a **break** statement is omitted, execution continues into the following **case** or **default** statement (if either exists). That is, when a **break** statement is missing, execution "falls through" into the next **case** and stops only when a **break** statement or the end of the **switch** is encountered. For example, study this program carefully:

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

int main(void)
{
  char ch;

  do {
    printf("\nEnter a character, q to quit: ");
    ch = getche();
    printf("\n");

    switch(ch) {
      case 'a':
        printf("Now is ");
      case 'b':
        printf("the time ");
      case 'c':
        printf("for all good men");
        break;
      case 'd':
        printf("The summer ");
      case 'e':
        printf("soldier ");
    }
  } while(ch != 'q');

  return 0;
}
```

If the user types **a**, the entire phrase **Now is the time for all good men** is displayed. Typing **b** displays **the time for all**

good men. As you can see, once execution begins inside a **case**, it continues until a **break** statement or the end of the **switch** is encountered.

3. The statement sequence associated with a **case** may be empty. This allows two or more **cases** to share a common statement sequence without duplication of code. For example, here is a program that categorizes letters into vowels and consonants:

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

int main(void)
{
  char ch;

  printf("Enter the letter: ");
  ch = getche();

  switch(ch) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
    case 'y':
      printf(" is a vowel\n");
      break;
    default:
      printf(" is a consonant");
  }

  return 0;
}
```

EXERCISES

1. What is wrong with this fragment?

```
float f;

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

```
switch(f) {
  case 10.05:
.
.
.
```

2. Write a program that counts the numbers of letters, digits, and common punctuation symbols entered by the user. Stop inputting when the user presses ENTER. Use a **switch** statement to categorize the characters into punctuation, digits, and letters. When the program ends, report the number of characters in each category. (If you like, simply assume that, if a character is not a digit or punctuation, it is a letter. Also, just use the most common punctuation symbols.)

3.10 UNDERSTAND THE goto STATEMENT

C supports a non-conditional jump statement, called the **goto**. Because C is a replacement for assembly code, the inclusion of **goto** is necessary because it can be used to create very fast routines. However, most programmers do not use **goto** because it destructures a program and, if frequently used, can render the program virtually impossible to understand later. Also, there is no routine that requires a **goto**. For these reasons, it is not used in this book outside of this section.

The **goto** statement can perform a jump within a function. It cannot jump between functions. It works with a label. In C, a *label* is a valid identifier name followed by a colon. For example, the following **goto** jumps around the **printf( )** statement:

```
goto mylabel;
printf("This will not print.");
mylabel: printf("This will print.");
```

About the only good use for **goto** is to jump out of a deeply nested routine when a catastrophic error occurs.

EXAMPLE

1. This program uses **goto** to create the equivalent of a **for** loop running from **1** to **10**. (This is just an example of **goto**. In actual practice, you should use a real **for** loop when one is needed.)

```
#include <stdio.h>

int main(void)
{
  int i;

  i = 1;
  again:
    printf("%d ", i);
    i++;
    if(i<10) goto again;

  return 0;
}
```

EXERCISES

1. Write a program that uses **goto** to emulate a **while** loop that counts from **1** to **10**.

Mastery Skills Check

At this point, you should be able to answer these questions and perform these exercises:

1. As illustrated by Exercise 2 in Section 3.1, the ASCII codes for the lowercase letters are separated from the uppercase letters by a difference of 32. Therefore, to convert a lowercase letter to

an uppercase one, simply subtract 32 from it. Write a program that reads characters from the keyboard and displays lowercase letters as uppercase ones. Stop when ENTER is pressed.

2. Using a nested **if** statement, write a program that prompts the user for a number and then reports if the number is positive, zero, or negative.

3. Is this a valid **for** loop?

```
char ch;

ch = 'x';
for( ; ch != ' ' ; ) ch = getche();
```

4. Show the traditional way to create an infinite loop in C.

5. Using the three loop statements, show three different ways to count from 1 to 10.

6. What does the **break** statement do when used in a loop?

7. Is this **switch** statement correct?

```
switch(i) {
  case 1: printf("nickel");
    break;
  case 2: printf("dime");
    break;
  case 3: printf("quarter");
}
```

8. Is this **goto** fragment correct?

```
goto alldone;
.
.
.
alldone
```

Cumulative Skills Check

This section checks how well you have integrated the material in this chapter with that from earlier chapters.

1. Using a **switch** statement, write a program that reads characters from the keyboard and watches for tabs, newlines, and backspaces. When one is received, display what it is in words. For example, when the user presses the TAB key, print **tab**. Have the user enter a **q** to stop the program.
2. While this program is not incorrect, show how it would look if written by an experienced C programmer.

```
#include <stdio.h>

int main(void)
{
  int i, j, k;

  for(k=0; k<10; k=k+1) {
    printf("Enter first number: ");
    scanf("%d", &i);

    printf("Enter second number: ");
    scanf("%d", &j);

    if(j != 0) printf("%d\n", i/j);
    if(j == 0) printf("cannot divide by zero\n");
  }

  return 0;
}
```

4

A Closer Look at Data Types, Variables, and Expressions

chapter objectives

4.1 Use C's data-type modifiers

4.2 Learn where variables are declared

4.3 Take a closer look at constants

4.4 Initialize variables

4.5 Understand type conversions in expressions

4.6 Understand type conversions in assignments

4.7 Program with type casts

THIS chapter more fully examines several concepts presented in Chapter 1. It covers C's data-type modifiers, global and local variables, and constants. It also discusses how C handles various type conversions.

Review Skills Chec

Before proceeding, you should be able to answer these questions and perform these exercises:

1. Using C's three loop statements, show three ways to write a loop that counts from **1** to **10**.
2. Convert this series of **if**s into an equivalent **switch**.

   ```
   if(ch=='L') load();
   else if(ch=='S') save();
   else if(ch=='E') enter();
   else if(ch=='D') display();
   else if(ch=='Q') quit();
   ```

3. Write a program that inputs characters until the user strikes the ENTER key.
4. What does **break** do?
5. What does **continue** do?
6. Write a program that displays this menu, performs the selected operation, and then repeats until the user selects **Quit**.

   ```
   Convert
     1. feet to meters
     2. meters to feet
     3. ounces to pounds
     4. pounds to ounces
     5. Quit
   Enter the number of your choice:
   ```

4.1 USE C'S DATA-TYPE MODIFIERS

In Chapter 1 you learned that C has five basic data types: **void**, **char**, **int**, **float**, and **double**. These basic types, except type **void**, can be modified using C's *type modifiers* to more precisely fit your specific need. The type modifiers are

long
short
signed
unsigned

The type modifier precedes the type name. For example, this declares a **long** integer:

```
long int i;
```

The effect of each modifier is examined next.

The **long** and **short** modifiers may be applied to **int.** As a general rule, **short ints** are often smaller than **ints** and **long ints** are often larger than **ints**. For example, in most 16-bit environments, an **int** is 16 bits long and a **long int** is 32 bits in length. However, the precise meaning of **long** and **short** is implementation dependent. When the ANSI C standard was created, it specified *minimum ranges* for integers, short integers, and long integers. It did not set fixed sizes for these items. (See Table 4-1.) For example, using the minimum ranges set forth in the ANSI C standard, the smallest acceptable size for an **int** is 16 bits and the smallest acceptable size for a **short int** is also 16 bits. Thus, it is permissible for integers and short integers to be the same size! In fact, in most 16-bit environments, there is no difference between an **int** and a **short int**. Further, in many 32-bit environments, you will find that integers and long integers are the same size. Since the exact effect of **long** and **short** on integers is determined by the environment in which you are working and by the compiler you are using, you will need to check your compiler's documentation for their precise effects.

The **long** modifier may also be applied to **double**. Doing so roughly doubles the precision of a floating point variable.

The **signed** modifier is used to specify a signed integer value. (A signed number means that it can be positive or negative.) However, the use of **signed** on integers is redundant because the default integer declaration automatically creates a signed variable. The main use of the **signed** modifier is with **char**. Whether **char** is signed or unsigned by itself is implementation dependent. In some implementations **char** is unsigned by default; in others, it is signed. To ensure a signed character variable in all environments, you must declare it as **signed char**. Since most compilers implement **char** as signed, this book simply assumes that characters are **signed** and will not use the **signed** modifier.

The **unsigned** modifier can be applied to **char** and **int**. It may also be used in combination with **long** or **short**. It is used to create an unsigned integer. The difference between signed and unsigned integers is in the way the high-order bit of the integer is interpreted. If a signed integer is specified, then the C compiler will generate code that assumes the high-order bit is used as a sign flag. If the sign flag is 0, the number is positive; if it is 1, the number is negative. Negative numbers are generally represented using the *two's complement* approach. In this method, all bits in the number (except the sign flag) are reversed, and 1 is added to this number. Finally, the sign flag is set to 1. (The reason for this method of representation is that it makes it easier for the CPU to perform arithmetic operations on negative values.)

Signed integers are important for a great many algorithms, but they only have half the absolute magnitude of their unsigned relatives. For example, here is 32,767 shown in binary:

```
0 1 1 1 1 1 1 1   1 1 1 1 1 1 1 1
```

If this is a signed value and the high-order bit is set to 1, the number would then be interpreted as –1 (assuming two's complement format). However, if this is an unsigned value, then when the high-order bit is set to 1, the number becomes 65,535.

Table 4-1 shows all allowed combinations of the basic types and the type modifiers. The table also shows the most common size and minimum range for each type as specified by the ANSI C standard.

It is important to understand that the ranges shown in Table 4-1 are just the minimums that all compilers must provide. The compiler is free to exceed them, and most compilers do for at least some data types. As mentioned, an **int** in a 32-bit environment will usually have a range larger than the minimum. Also, in environments that use

Type	Typical Size in Bits	Minimal Range
char	8	–127 to 127
unsigned char	8	0 to 255
signed char	8	–127 to 127
int	16 or 32	–32,767 to 32,767
unsigned int	16 or 32	0 to 65,535
signed int	16 or 32	same as int
short int	16	same as int
unsigned short int	16	0 to 65,535
signed short int	16	same as short int
long int	32	–2,147,483,647 to 2,147,483,647
signed long int	32	same as long int
unsigned long int	32	0 to 4,294,967,295
float	32	Six digits of precision
double	64	Ten digits of precision
long double	80	Ten digits of precision

TABLE 4-1 *All Data Types Defined by the ANSI C Standard* ▼

two's complement arithmetic (which is the case for the vast majority of computers), the lower bound for signed characters and integers is one greater than the minimums shown. For instance, in most environments, a **signed char** has a range of –128 to 127 and a **short int** is typically –32,768 to 32,767. You will need to check your compiler's documentation for the specific ranges of the data types as they apply to your compiler.

C allows a shorthand notation for declaring **unsigned**, **short**, or **long** integers. You may simply use the word **unsigned**, **short**, or **long** without the **int**. The **int** is implied. For example,

```
unsigned count;
unsigned int num;
```

both declare **unsigned int** variables.

It is important to remember that variables of type **char** may be used to hold values other than just the ASCII character set. C makes little distinction between a character and an integer, except for the magnitudes of the values each may hold. Therefore, as mentioned earlier, a signed **char** variable can also be used as a "small" integer when the situation does not require larger numbers.

When outputting integers modified by **short**, **long**, or **unsigned** using **printf( )**, you cannot simply use the **%d** specifier. The reason is that **printf( )** needs to know precisely what type of data it is receiving. To use **printf( )** to output a **short**, use **%hd**. To output a **long**, use **%ld**. When outputting an **unsigned** value, use **%u**. To output an **unsigned long int**, use **%lu**. Also, to output a **long double** use **%Lf**.

The **scanf( )** function operates in a fashion similar to **printf( )**. When reading a **short int** using **scanf( )**, use **%hd**. When reading a **long int**, use **%ld**. To read an **unsigned long int**, use **%lu**. To read a **double**, use **%1f**. To read a **long double**, use **%Lf**.

EXAMPLES

1. This program shows how to input and output **short**, **long**, and **unsigned** values.

```
#include <stdio.h>

int main(void)
{
  unsigned u;
  long l;
  short s;

  printf("Enter an unsigned: ");
  scanf("%u", &u);
  printf("Enter a long: ");
  scanf("%ld", &l);
  printf("Enter a short: ");
  scanf("%hd", &s);

  printf("%u %ld %hd\n", u, l, s);

  return 0;
}
```

2. To understand the difference between the way that signed and unsigned integers are interpreted by C, run the following short program. (This program assumes that short integers are 16 bits wide.)

```
#include <stdio.h>

int main(void)
{
  short int i;  /* a signed short integer */
  unsigned short int u; /* an unsigned short integer */

  u = 33000;
  i = u;
  printf("%hd %hu", i, u);

  return 0;
}
```

When this program is run, the output is **-32536 33000**. The reason for this is that the bit pattern that 33000 represents as an **unsigned short int** is interpreted as -32536 as a **signed short int**.

3. In C, you may use a **char** variable any place you would use an **int** variable (assuming the differences in their ranges is not a factor). For example, the following program uses a **char** variable to control the loop that is summing the numbers between 1 and 100. In some cases it takes the computer less time to access a single byte (one character) than it does to access two bytes. Therefore, many professional programmers use a character variable rather than an integer one when the range permits.

```
#include <stdio.h>

int main(void)
{
  int i;
  char j;

  i = 0;
  for(j=1; j<101; j++) i = j + i;

  printf("Total is: %d", i);

  return 0;
}
```

EXERCISES

1. Show how to declare an **unsigned short int** called **loc_counter**.
2. Write a program that prompts the user for a distance and computes how long it takes light to travel that distance. Use an **unsigned long int** to hold the distance. (Light travels at approximately 186,000 miles per second.)
3. Write this statement another way:

   ```
   short int i;
   ```

4.2 LEARN WHERE VARIABLES ARE DECLARED

As you learned in Chapter 1, there are two basic places where a variable will be declared: inside a function and outside all functions. These variables are called *local* variables and *global* variables, respectively. It is now time to take a closer look at these two types of variables and the *scope rules* that govern them.

Local variables (declared *inside* a function) may be referenced only by statements that are inside that function. They are not known outside their own function. One of the most important things to understand about local variables is that they exist only while the function in which they are declared is executing. That is, a local variable is created upon entry into its function and destroyed upon exit.

Since local variables are not known outside their own function, it is perfectly acceptable for local variables in different functions to have the same name. Consider the following program:

```
#include <stdio.h>

void f1(void), f2(void);

int main(void)
{
  f1();

  return 0;
}
```

```
void f1(void)
{
  int count;

  for(count=0; count<10; count++) f2();
}

void f2(void)
{
  int count;

  for(count=0; count<10; count++) printf("%d ", count);
}
```

This program prints the numbers **0** through **9** on the screen ten times. The fact that both functions use a variable called **count** has no effect upon the operation of the code. Therefore, what happens to **count** inside **f2( )** has no effect on **count** in **f1( )**.

The C language contains the keyword **auto**, which can be used to declare local variables. However, since all local variables are, by default, assumed to be **auto**, it is virtually never used. Hence, you will not see it in any of the examples in this book.

Within a function, local variables can be declared at the start of any block. They do not need to be declared only at the start of the block that defines the function. For example, the following program is perfectly valid:

```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=0; i<10; i++) {
    if(i==5) {
      int j; /* declare j within the if block */

      j = i * 10;
      printf ("%d", j);
    }
  }

  return 0;
}
```

A variable declared within a block is known only to other code within that block. Thus, **j** may not be used outside of its block. Frankly, most C programmers declare all variables used by a function at the start of the function's block because it is simply more convenient to do so. This is the approach that will be used in this book.

Remember one important point: You must declare all local variables at the start of the block in which they are defined, prior to any program statements. For example, the following is incorrect:

```
#include <stdio.h>

int main(void)
{
  printf("This program won't compile.");
  int i; /* this should come first */
  i = 10;
  printf("%d", i);

  return 0;
}
```

When a function is called, its local variables are created, and upon its return, they are destroyed. This means that local variables cannot retain their values between calls.

The formal parameters to a function are also local variables. Even though these variables perform the special task of receiving the value of the arguments passed to the function, they can be used like any other local variable within that function.

Unlike local variables, global variables are known throughout the entire program and may be used by any piece of code in the program Also, they will hold their value during the entire execution of the program. Global variables are created by declaring them outside any function. For example, consider this program:

```
#include <stdio.h>

void f1(void);

int max; /* this is a global variable */

int main(void)
{
  max = 10;
  f1();

  return 0;
}

void f1(void)
{
  int i;

  for(i=0; i<max; i++) printf("%d ", i);
}
```

Here, both **main( )** and **f1( )** use the global variable **max**. The **main( )** function sets the value of **max** to 10, and **f1( )** uses this value to control its **for** loop.

EXAMPLES

1. In C, a local variable and a global variable may have the same name. For example, this is a valid program:

```
#include <stdio.h>

void f1(void);

int count; /* global count */

int main(void)
{
  count = 10;
  f1();
  printf("count in main(): %d\n", count);

  return 0;
}

void f1(void)
{
  int count; /* local count */

  count = 100;
  printf("count in f1() : %d\n", count);
}
```

The program displays this output:

```
count in f1() : 100
count in main() : 10
```

In **main( )**, the reference to **count** is to the global variable. Inside **f1( )**, a local variable called **count** is also defined. When the assignment statement inside **f1( )** is encountered, the compiler first looks to see if there is a local variable called **count**. Since there is, the local variable is used, not the global one with the same name. That is, when local and global variables share the same name, the compiler will always use the local variable.

2. Global variables are very helpful when the same data is used by many functions in your program. However, you should always use local variables where you can because the excessive use of global variables has some negative consequences. First, global variables use memory the entire time your program is executing, not just when they are needed. In situations where memory is in short supply, this could be a problem. Second, using a global where a local variable will do makes a function less general, because it relies on something that must be defined outside itself. For example, here is a case where global variables are being used for no reason:

```
#include <stdio.h>

int power(void);

int m, e;

int main(void)
{
  m = 2;
  e = 3;

  printf("%d raised to the %d power is %d", m, e, power());

  return 0;
}

/* Non-general version of power. */
int power(void)
```

```
  int temp, temp2;

  temp = 1;
  temp2 = e;
  for( ; temp2> 0; temp2--) temp = temp * m;

  return temp;
}
```

Here, the function **power( )** is created to compute the value of **m** raised to the **e**th power. Since **m** and **e** are global, the function cannot be used to compute the power of other values. It can only operate on those contained within **m** and **e**. However, if the program is rewritten as follows, **power( )** can be used with any two values.

```
#include <stdio.h>

int power(int m, int e);

int main(void)
{
  int m, e;
  m = 2;
  e = 3;

  printf("%d to the %d is %d\n", m, e, power(m, e));
  printf("4 to the 5th is %d\n", power(4, 5));
  printf("3 to the 3rd is %d\n", power(3, 3));

  return 0;
}

/* Parameterized version of power. */
int power(int m, int e)
{
  int temp;

  temp = 1;
  for( ; e> 0; e--) temp = temp * m;

  return temp;
}
```

By parameterizing **power( )**, you can use it to return the result of any value raised to some power, as the program now shows.

The important point is that in the non-generalized version, any program that uses **power( )** must always declare **m** and **e** as global variables and then load them with the desired values each time **power( )** is used. In the parameterized form, the function is complete within itself—no extra baggage need be carried about when it is used.

Finally, using a large number of global variables can lead to program errors because of unknown and unwanted side effects. A major problem in developing large programs is the accidental modification of a variable's value because it was used elsewhere in the program. This can happen in C if you use too many global variables in your programs.

3. Remember, local variables do not maintain their values between functions calls. For example, the following program will not work correctly:

```
#include <stdio.h>

int series(void);

int main(void)
{
  int i;

  for(i=0; i<10; i++) printf("%d ", series());

  return 0;
}

/* This is incorrect. */
int series(void)
{
  int total;

  total = (total + 1423) % 1422;
  return total;
}
```

This program attempts to use **series( )** to generate a number series in which each number is based upon the value of the preceding one. However, the value **total** will not be maintained between function calls, and the function fails to carry out its intended task.

EXERCISES

1. What are key differences between local and global variables?
2. Write a program that contains a function called **soundspeed( )**, which computes the number of seconds it will take sound to travel a specified distance. Write the program two ways: first, with **soundspeed( )** as a non-general function and second, with **soundspeed( )** parameterized. (For the speed of sound, use 1129 feet per second).

TAKE A CLOSER LOOK AT CONSTANTS

Constants refer to fixed values that may not be altered by the program. For example, the number 100 is a constant. We have been using constants in the preceding sample programs without much fanfare because, in most cases, their use is intuitive. However, the time has come to cover them formally.

Integer constants are specified as numbers without fractional components. For example, 10 and –100 are integer constants. Floating-point constants require the use of the decimal point followed by the number's fractional component. For example, 11.123 is a floating-point constant. C also allows you to use scientific notation for floating-point numbers. Constants using scientific notation must follow this general form:

number E *sign exponent*

The *sign* is optional. Although the general form is shown with spaces between the component parts for clarity, there may be no spaces between the parts in an actual number. For example, the following defines the value 1234.56 using scientific notation:

```
123.456E1
```

Character constants are enclosed between single quotes. For example 'a' and '%' are both character constants. As some of the examples have shown, this means that if you wish to assign a character to a variable of type **char**, you will use a statement similar to

```
ch = 'Z';
```

However, there is nothing in C that prevents you from assigning a character variable a value using a numeric constant. For example, the ASCII code for 'A' is 65. Therefore, these two assignment statements are equivalent.

```
char ch;

ch = 'A';
ch = 65;
```

When you enter numeric constants into your program, the compiler must decide what type of constant they are. For example, is 1000 an **int**, an **unsigned**, or a **short**? The reason we haven't worried about this earlier is that C automatically converts the type of the right side of an assignment statement to that of the variable on the left. (We will examine this process more fully later in this chapter.) So, for many situations it doesn't matter what the compiler thinks 1000 is. However, this can be important when you use a constant as an argument to a function, such as in a call to **printf( )**.

By default, the C compiler fits a numeric constant into the smallest compatible data type that will hold it. Assuming 16-bit integers, 10 is an **int** by default and 100003 is a **long**. Even though the value 10 could be fit into a **char**, the compiler will not do this because it means crossing type boundaries. The only exceptions to the smallest-type rule are floating-point constants, which are assumed to be **doubles**. For virtually all programs you will write as a beginner, the compiler defaults are perfectly adequate. However, as you will see later in this book, there will come a point when you will need to specify precisely the type of constant you want.

In cases where the assumption that C makes about a numeric constant is not what you want, C allows you to specify the exact type by using a suffix. For floating-point types, if you follow the number with an 'F', the number is treated as a **float**. If you follow it with an 'L', the number becomes a **long double**. For integer types, the 'U' suffix stands for **unsigned** and the 'L' stands for **long**.

As you may know, in programming it is sometimes easier to use a number system based on 8 or 16 instead of 10. As you learned in Chapter 2, the number system based on 8 is called *octal* and it uses the digits 0 through 7. The base-16 number system is called *hexadecimal* and uses the digits 0 through 9 plus the letters 'A' through 'F', which stand for 10 through 15. C allows you to specify integer constants as hexadecimal or octal instead of decimal if you prefer. A hexadecimal constant must begin with '0x' (a zero followed by an x) then the constant in hexadecimal form. An octal constant begins with a zero. For example, **0xAB** is a hexadecimal constant, and **024** is an octal constant. You may use either upper- or lowercase letters when entering hexadecimal constants.

C supports one other type of constant in addition to those of the predefined data types: the string. A *string* is a set of characters enclosed by double quotes. You have been working with strings since Chapter 1 because both the **printf( )** and **scanf( )** functions use them. Keep in mind one important fact: although C allows you to define string constants, it does not formally have a string data type. Instead, as you will see a little later in this book, strings are supported in C as character arrays. (Arrays are discussed in Chapter 5.)

To display a string using **printf( )** you can either make it part of the control string or pass it as a separate argument and display it using the **%s** format code. For example, this program prints **Once upon a time** on the screen:

```
#include <stdio.h>

int main(void)
{
  printf("%s %s %s", "Once", "upon", "a time");

  return 0;
}
```

Here, each string is passed to **printf( )** as an argument and displayed using the **%s** specifier.

EXAMPLES

1. To see why it is important to use the correct type specifier with **printf( )**, try this program. (It assumes that short integers are 16 bits.) Instead of printing the number **42340**, it displays **-23196**, because it thinks that it is receiving a signed short integer. The problem is that 42,340 is outside the range of a **short int**. To make it work properly, you must use the **%hu** specifier.

   ```
   #include <stdio.h>

   int main(void)
   {
     printf("%hd", 42340); /* this won't work right */

     return 0;
   }
   ```

2. To see why you may need to explicitly tell the compiler what type of constant you are using, try this program. For most compilers, it will not produce the desired output. (If it does work, it is only by chance.)

   ```
   #include <stdio.h>

   int main(void)
   {
     printf("%f", 2309);

     return 0;
   }
   ```

   This program is telling **printf( )** to expect a floating point value, but the compiler assumes that 2309 is simply an **int**. Hence, it does not output the correct value. To fix it, you must specify 2309 as 2309.0. Adding the decimal point forces the compiler to treat the value as a **double**.

EXERCISES

1. How do you tell the C compiler that a floating-point constant should be represented as a **float** instead of a **double**?
2. Write a program that reads and writes a **long int** value.
3. Write a program that outputs **I like C** using three separate strings.

INITIALIZE VARIABLES

A variable may be given an initial value when it is declared. This is called *variable initialization*. The general form of variable initialization is shown here:

```
type var-name = constant;
```

For example, this statement declares **count** as an **int** and gives it an initial value of 100.

```
int count = 100;
```

The main advantage of using an initialization rather than a separate assignment statement is that the compiler may be able to produce faster code. Also, this saves some typing effort on your part.

Global variables may be initialized using only constants. Local variables can be initialized using constants, variables, or function calls as long as each is valid at the time of the initialization. However, most often both global and local variables are initialized using constants.

Global variables are initialized only once, at the start of program execution. Local variables are initialized each time a function is entered.

Global variables that are not explicitly initialized are automatically set to zero. Local variables that are not initialized should be assumed to contain unknown values. Although some C compilers automatically initialize un-initialized local variables to 0, you should not count on this.

EXAMPLES

1. This program gives **i** the initial value of −1 and then displays its value.

```
#include <stdio.h>

int main(void)
{
  int i = -1;

  printf("i is initialized to %d", i);

  return 0;
}
```

2. When you declare a list of variables, you may initialize one or more of them. For example, this fragment initializes **min** to 0 and **max** to 100. It does not initialize **count**.

```
int min=0, count, max=100;
```

3. As stated earlier, local variables are initialized each time the function is entered. For this reason, this program prints **10** three times.

```
#include <stdio.h>

void f(void);

int main(void)
{
  f();
  f();
  f();

  return 0;
}

void f(void)
{
  int i = 10;

  printf("%d ", i);
}
```

4. A local variable can be initialized by any expression valid at the time the variable is declared. For example, consider this program:

```
#include <stdio.h>

int x = 10; /* initialize global variable */

int myfunc(int i);

int main(void)
{
  /* initialize a local variable using
     a global variable */
  int y = x;

  /* initialize a local variable using another
     local variable and a function call */
  int z = myfunc(y);

  printf("%d %d", y, z);

  return 0;
}

int myfunc(int i)
{
  return i/2;
}
```

The local variable **y** is initialized using the value of the global variable **x**. Since **x** is initialized before **main( )** is called, it is valid to use its value to initialize a local variable. The value of **z** is initialized by calling **myfunc( )** using **y** as an argument. Since **y** has already been initialized, it is entirely proper to use it as an argument to **myfunc( )** at this point.

EXERCISES

1. Write a program that gives an integer variable called **i** an initial value of **100** and then uses **i** to control a **for** loop that displays the numbers **100** down to **1**.

2. Assume that this line of code declares global variables. Is it correct?

   ```
   int a=1, b=2, c=a;
   ```

3. If the preceding declaration was for local variables, would it be correct?

UNDERSTAND TYPE CONVERSIONS IN EXPRESSIONS

Unlike many other computer languages, C lets you mix different typ of data together in one expression. For example, this is perfectly vali C code:

```
char ch;
int i;
float f;
double outcome;

ch = '0';
i = 10;
f = 10.2;

outcome = ch * i / f;
```

C allows the mixing of types within an expression because it has a strict set of conversion rules that dictate how type differences are resolved. Let's look closely at them in this section.

One portion of C's conversion rules is called *integral promotion*. In C, whenever a **char** or a **short int** is used in an expression, its value is automatically elevated to **int** during the evaluation of that expression. This is why you can use **char** variables as "little integers" anywhere you can use an **int** variable. Keep in mind that the integral promotion is only in effect during the evaluation of an expression. The variable does not become physically larger. (In essence, the compiler just uses a temporary copy of its value.)

After the automatic integral promotions have been applied, the C compiler will convert all operands "up" to the type of the largest operand. This is called *type promotion* and is done on an operation-by-operation basis, as described in the following type-conversion algorithm.

```
IF an operand is a long double
THEN the second is converted to long double
ELSE IF an operand is a double
THEN the second is converted to double
ELSE IF an operand is a float
THEN the second is converted to float
ELSE IF an operand is an unsigned long
THEN the second is converted to unsigned long
ELSE IF an operand is long
THEN the second is converted to long
ELSE IF an operand is unsigned
THEN the second is converted to unsigned
```

There is one additional special case: If one operand is **long** and the other is **unsigned int**, and if the value of the **unsigned int** cannot be represented by a **long**, both operands are converted to **unsigned long**.

Once these conversion rules have been applied, each pair of operands will be of the same type and the result of each operation will be the same as the type of both operands.

EXAMPLES

1. In this program, **i** is elevated to a **float** during the evaluation of the expression **i*f**. Thus, the program prints **232.5**.

```
#include <stdio.h>

int main(void)
{
  int i;
  float f;

  i = 10;
  f = 23.25;

  printf("%f", i*f);

  return 0;
}
```

2. This program illustrates how **short ints** are automatically promoted to **ints**. The **printf( )** statement works correctly eve though the **%d** modifier is used because **i** is automatically elevated to **int** when **printf( )** is called.

```
#include <stdio.h>

int main(void)
{
  short int i;

  i = -10;
  printf("%d", i);

  return 0;
}
```

3. Even though the final outcome of an expression will be of the largest type, the type conversion rules are applied on an operation-by-operation basis. For example, in this expression

```
100.0/(10/3)
```

the division of 10 by 3 produces an integer result, since both ar integers. Then this value is elevated to 3.0 to divide 100.0.

EXERCISES

1. Given these variables,

```
char ch;
short i;
unsigned long ul;
float f;
```

what is the overall type of this expression:

```
f/ch - (i*ul)
```

2. What is the type of the subexpression **i*ul**, above?

4.6 UNDERSTAND TYPE CONVERSIONS IN ASSIGNMENTS

In an assignment statement in which the type of the right side differs from that of the left, the type of the right side is converted into that of the left. When the type of the left side is larger than the type of the right side, this process causes no problems. However, when the type of the left side is smaller than the type of the right, data loss may occur. For example, this program displays **–24**:

```
#include <stdio.h>

int main(void)
{
  char ch;
  int i;

  i = 1000;
  ch = i;

  printf("%d", ch);

  return 0;
}
```

The reason for this is that only the low-order eight bits of **i** are copied into **ch**. Since this sort of assignment type conversion is not an error in C, you will receive no error message. Remember, one reason C was created was to replace assembly language, so it must allow all sorts of type conversions. For example, in some instances you may only want the low-order eight bits of **i**, and this sort of assignment is an easy way to obtain them.

When there is an integer-to-character or a longer-integer to shorter-integer type conversion across an assignment, the basic rule is that the appropriate number of high-order bits will be removed. For example, in many environments, this means 8 bits will be lost when going from an **int** to a **char**, and 16 bits will be lost when going from a **long** to an **int**.

When converting from a **long double** to a **double** or from a **double** to a **float**, precision is lost. When converting from a floating-point value to an integer value, the fractional part is lost, and if the number is too large to fit in the target type, a garbage value will result.

Remember two important points: First, the conversion of an **int** to a **float** or a **float** to **double**, and so on, will not add any precision or accuracy. These kinds of conversions will only change the form in which the value is represented. Second, some C compilers will always treat a **char** variable as an **unsigned** value. Others will treat it as a **signed** value. Thus, what will happen when a character variable holds a value greater than 127 is implementation-dependent. If this is important in a program that you write, it is best to declare the variable explicitly as either **signed** or **unsigned**.

EXAMPLES

1. As stated, when converting from a floating-point value to an integer value, the fractional portion of the number is lost. The following program illustrates this fact. It prints **1234.0098 1234**.

```
#include <stdio.h>

int main(void)
{
  int i;
  float f;

  f = 1234.0098;
  i = f; /* convert to int */
  printf("%f %d", f, i);

  return 0;
}
```

2. When converting from a larger integer type to a smaller one, it is possible to generate a garbage value, as this program illustrates. (This program assumes that short integers are 16 bits long and that long integers are 32 bits long.)

```
#include <stdio.h>

int main(void)
{
  short int si;
  long int li;
  li = 100000;
  si = li; /* convert to short int */

  printf("%hd", si);

  return 0;
}
```

Since the largest value that a short integer can hold is 32,767, it cannot hold 100,000. What the compiler does, however, is copy the lower-order 16 bits of **li** into **si**. This produces the meaningless value of **–31072** on the screen.

EXERCISES

1. What will this program display?

```
#include <stdio.h>

int main(void)
{
  int i;
  long double ld;

  ld = 10.0;
  i = ld;

  printf("%d", i);
}
```

2. What does this program display?

```
#include <stdio.h>

int main(void)
{
  float f;

  f = 10 / 3;
  printf("%f", f);

  return 0;
}
```

4.7 PROGRAM WITH TYPE CASTS

Sometimes you may want to transform the type of a variable temporarily. For example, you may want to use a floating-point value for one computation, but wish to apply the modulus operator to it elsewhere. Since the modulus operator can only be used on integer values, you have a problem. One solution is to create an integer variable for use in the modulus operation and assign the value of the floating-point variable to it when the time comes. This is a somewhat inelegant solution, however. The other way around this problem is to use a *type cast*, which causes a temporary type change.

A type cast takes this general form:

(*type*) *value*

where *type* is the name of a valid C data type. For example,

```
float f;

f = 100.2;

/* print f as an integer */
printf("%d", (int) f);
```

Here, the type cast causes the value of **f** to be converted to an **int**.

EXAMPLES

1. As you learned in Chapter 1, **sqrt( )**, one of C's library functions, returns the square root of its argument. It uses the MATH.H header file. Its single argument must be of type **double**. It also returns a **double** value. The following program prints the square roots of the numbers between **1** and **100** using a **for** loop. It also prints the whole number portion and the fractional part of each result separately. To do so, it uses a type cast to convert **sqrt( )**'s return value into an **int**.

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

int main(void)
{
  double i;

  for(i=1.0; i<101.0; i++) {
    printf("The square root of %lf is %lf\n", i, sqrt(i));
    printf("Whole number part: %d ", (int)sqrt(i));
    printf("Fractional part: %lf\n", sqrt(i)-(int)sqrt(i));
    printf("\n");
  }

  return 0;
}
```

2. You cannot cast a variable that is on the left side of an assignment statement. For example, this is an invalid statement in C:

```
int num;

(float) num = 123.23; /* this is incorrect */
```

EXERCISES

1. Write a program that uses **for** to print the numbers **1** to **10** by tenths. Use a floating-point variable to control the loop. However, use a type cast so that the conditional expression is evaluated as an integer expression in the interest of speed.
2. Since a floating point value cannot be used with the % operator, how can you fix this statement?

```
x = 123.23 % 3; /* fix this statement */
```

Mastery Skills Check

At this point you should be able to answer these questions and perform these exercises:

1. What are C's data-type modifiers and what function do they perform?
2. How do you explicitly define an **unsigned** constant, a **long** constant, and a **long double** constant?
3. Show how to give a **float** variable called **balance** an initial value of 0.0.
4. What are C's automatic integral promotions?
5. What is the difference between a **signed** and an **unsigned** integer?
6. Give one reason why you might want to use a global variable in your program.
7. Write a program that contains a function called **series( )**. Have this function generate a series of numbers, based upon this formula:

   next-number = (*previous-number* * 1468) % 467

   Give the number an initial value of 21. Use a global variable to hold the last value between function calls. In **main( )** demonstrate that the function works by calling it ten times and displaying the result.
8. What is a type cast? Give an example.

Cumulative Skills Check

This section checks how well you have integrated the material in this chapter with that from earlier chapters.

1. As you know from Chapter 3, no two **cases** with the same **switch** may use the same value. Therefore, is this **switch** valid or invalid? Why? (Hint: the ASCII code for 'A' is 65.)

```
switch(x) {
  case 'A' : printf("is an A");
    break;
  case 65 : printf("is the number 65");
    break;
}
```

2. Technically, for traditional reasons the **getchar( )** and **getche( )** functions are declared as returning integers, not character values. However, the character read from the keyboard is contained in the low-order byte. Can you explain why this value can be assigned to **char** variables?

3. In this fragment, will the loop ever terminate? Why? (Assume integers are 16 bits long.)

```
int i
for(i=0; i<33000; i++);
```