

2

Introduction to C Programming

*What's in a name?
That which we call a rose
By any other name would
smell as sweet.*

—William Shakespeare

*“Take some more tea,” the
March Hare said to Alice, very
earnestly. “I’ve had nothing yet,”
Alice replied in an offended
tone: “so I can’t take more.” “You
mean you can’t take less,” said
the Hatter: “it’s very easy to take
more than nothing.”*

—Lewis Carroll

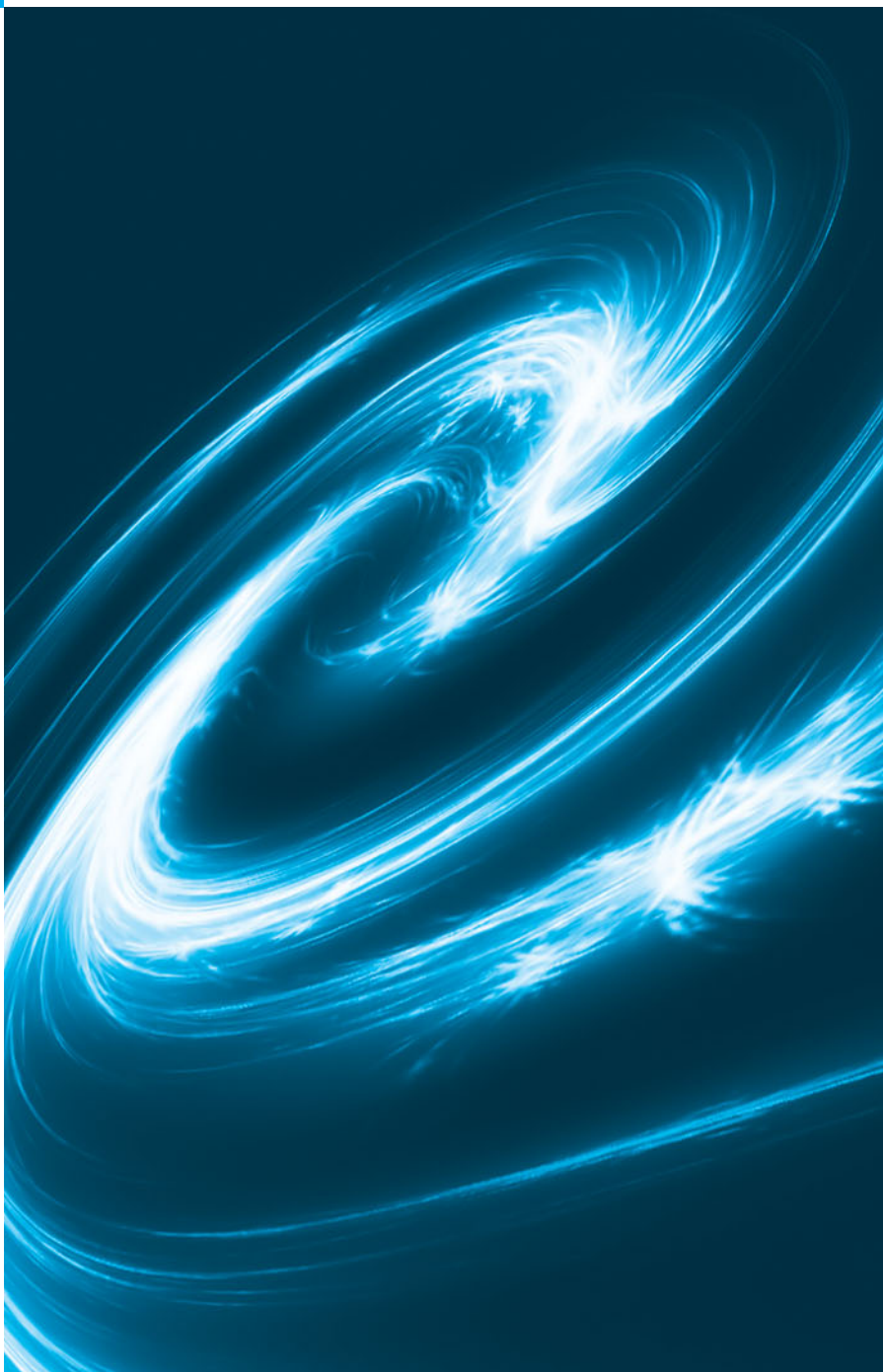
*High thoughts must have high
language.*

—Aristophanes

Objectives

In this chapter, you’ll:

- Write simple computer programs in C.
- Use simple input and output statements.
- Use the fundamental data types.
- Learn computer memory concepts.
- Use arithmetic operators.
- Learn the precedence of arithmetic operators.
- Write simple decision-making statements.



- | | |
|--|---|
| 2.1 Introduction | 2.4 Memory Concepts |
| 2.2 A Simple C Program: Printing a Line of Text | 2.5 Arithmetic in C |
| 2.3 Another Simple C Program: Adding Two Integers | 2.6 Decision Making: Equality and Relational Operators |
| | 2.7 Secure C Programming |

Summary | Terminology | Self-Review Exercises | Answers to Self-Review Exercises | Exercises | Making a Difference

2.1 Introduction

The C language facilitates a structured and disciplined approach to computer-program design. In this chapter we introduce C programming and present several examples that illustrate many important features of C. Each example is analyzed one statement at a time. In Chapters 3 and 4 we present an introduction to structured programming in C. We then use the structured approach throughout the remainder of the C portion of the text.

2.2 A Simple C Program: Printing a Line of Text

C uses some notations that may appear strange to people who have not programmed computers. We begin by considering a simple C program. Our first example prints a line of text. The program and its screen output are shown in Fig. 2.1.

```

1 // Fig. 2.1: fig02_01.c
2 // A first program in C.
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main( void )
7 {
8     printf( "Welcome to C!\n" );
9 } // end function main

```

```
Welcome to C!
```

Fig. 2.1 | A first program in C.

Comments

Even though this program is simple, it illustrates several important features of the C language. Lines 1 and 2

```

// Fig. 2.1: fig02_01.c
// A first program in C

```

begin with `//`, indicating that these two lines are **comments**. You insert comments to **document programs** and improve program readability. Comments do *not* cause the computer to perform any action when the program is run. Comments are *ignored* by the C compiler and do *not* cause any machine-language object code to be generated. The preceding com-

ment simply describes the figure number, file name and purpose of the program. Comments also help other people read and understand your program.

You can also use `/*...*/` **multi-line comments** in which everything from `/*` on the first line to `*/` at the end of the last line is a comment. We prefer `//` comments because they're shorter and they eliminate common programming errors that occur with `/*...*/` comments, especially when the closing `*/` is omitted.

#include Preprocessor Directive

Line 3

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

is a directive to the **C preprocessor**. Lines beginning with `#` are processed by the preprocessor *before* compilation. Line 3 tells the preprocessor to include the contents of the **standard input/output header** (`<stdio.h>`) in the program. This header contains information used by the compiler when compiling calls to standard input/output library functions such as `printf` (line 8). We explain the contents of headers in more detail in Chapter 5.

Blank Lines and White Space

Line 4 is simply a blank line. You use blank lines, space characters and tab characters (i.e., “tabs”) to make programs easier to read. Together, these characters are known as **white space**. White-space characters are normally ignored by the compiler.

The main Function

Line 6

```
int main( void )
```

is a part of every C program. The parentheses after `main` indicate that `main` is a program building block called a **function**. C programs contain one or more functions, one of which *must* be `main`. Every program in C begins executing at the function `main`. Functions can *return* information. The keyword `int` to the left of `main` indicates that `main` “returns” an integer (whole-number) value. We’ll explain what it means for a function to “return a value” when we demonstrate how to create your own functions in Chapter 5. For now, simply include the keyword `int` to the left of `main` in each of your programs. Functions also can *receive* information when they’re called upon to execute. The `void` in parentheses here means that `main` does *not* receive any information. In Chapter 14, we’ll show an example of `main` receiving information.



Good Programming Practice 2.1

Every function should be preceded by a comment describing the purpose of the function.

A left brace, `{`, begins the **body** of every function (line 7). A corresponding **right brace** ends each function (line 9). This pair of braces and the portion of the program between the braces is called a **block**. The block is an important program unit in C.

An Output Statement

Line 8

```
printf( "Welcome to C!\n" );
```

instructs the computer to perform an **action**, namely to print on the screen the **string** of characters marked by the quotation marks. A string is sometimes called a **character string**, a **message** or a **literal**. The entire line, including the `printf` function (the “f” stands for “formatted”), its **argument** within the parentheses and the semicolon (;), is called a **statement**. Every statement must end with a semicolon (also known as the **statement terminator**). When the preceding `printf` statement is executed, it prints the message `Welcome to C!` on the screen. The characters normally print exactly as they appear between the double quotes in the `printf` statement.

Escape Sequences

Notice that the characters `\n` were not printed on the screen. The backslash (`\`) is called an **escape character**. It indicates that `printf` is supposed to do something out of the ordinary. When encountering a backslash in a string, the compiler looks ahead at the next character and combines it with the backslash to form an **escape sequence**. The escape sequence `\n` means **newline**. When a newline appears in the string output by a `printf`, the newline causes the cursor to position to the beginning of the next line on the screen. Some common escape sequences are listed in Fig. 2.2.

Escape sequence	Description
<code>\n</code>	Newline. Position the cursor at the beginning of the next line.
<code>\t</code>	Horizontal tab. Move the cursor to the next tab stop.
<code>\a</code>	Alert. Produces a sound or visible alert without changing the current cursor position.
<code>\\</code>	Backslash. Insert a backslash character in a string.
<code>\"</code>	Double quote. Insert a double-quote character in a string.

Fig. 2.2 | Some common escape sequences .

Because the backslash has special meaning in a string, i.e., the compiler recognizes it as an escape character, we use a double backslash (`\\`) to place a single backslash in a string. Printing a double quote also presents a problem because double quotes mark the boundaries of a string—such quotes are not printed. By using the escape sequence `\"` in a string to be output by `printf`, we indicate that `printf` should display a double quote. The right brace, `}`, (line 9) indicates that the end of `main` has been reached.



Good Programming Practice 2.2

Add a comment to the line containing the right brace, `}`, that closes every function, including `main`.

We said that `printf` causes the computer to perform an **action**. As any program executes, it performs a variety of actions and makes **decisions**. Section 2.6 discusses decision making. Chapter 3 discusses this **action/decision model** of programming in depth.

The Linker and Executables

Standard library functions like `printf` and `scanf` are *not* part of the C programming language. For example, the compiler *cannot* find a spelling error in `printf` or `scanf`. When

the compiler compiles a `printf` statement, it merely provides space in the object program for a “call” to the library function. But the compiler does *not* know where the library functions are—the *linker* does. When the linker runs, it locates the library functions and inserts the proper calls to these library functions in the object program. Now the object program is complete and ready to be executed. For this reason, the linked program is called an **executable**. If the function name is misspelled, the *linker* will spot the error, because it will not be able to match the name in the C program with the name of any known function in the libraries.



Common Programming Error 2.1

Mistyping the name of the output function `printf` as `print` in a program.



Good Programming Practice 2.3

Indent the entire body of each function one level of indentation (we recommend three spaces) within the braces that define the body of the function. This indentation emphasizes the functional structure of programs and helps make programs easier to read.



Good Programming Practice 2.4

Set a convention for the size of indent you prefer and then uniformly apply that convention. The tab key may be used to create indents, but tab stops may vary.

Using Multiple `printf`s

The `printf` function can print `Welcome to C!` several different ways. For example, the program of Fig. 2.3 produces the same output as the program of Fig. 2.1. This works because each `printf` resumes printing where the previous `printf` stopped printing. The first `printf` (line 8) prints `Welcome` followed by a space, and the second `printf` (line 9) begins printing on the *same* line immediately following the space.

```
1 // Fig. 2.3: fig02_03.c
2 // Printing on one line with two printf statements.
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main( void )
7 {
8     printf( "Welcome " );
9     printf( "to C!\n" );
10 } // end function main
```

```
Welcome to C!
```

Fig. 2.3 | Printing on one line with two `printf` statements.

One `printf` can print *several* lines by using additional newline characters as in Fig. 2.4. Each time the `\n` (newline) escape sequence is encountered, output continues at the beginning of the next line.

```
1 // Fig. 2.4: fig02_04.c
2 // Printing multiple lines with a single printf.
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main( void )
7 {
8     printf( "Welcome\nto\nC!\n" );
9 } // end function main
```

```
Welcome
to
C!
```

Fig. 2.4 | Printing multiple lines with a single printf.

2.3 Another Simple C Program: Adding Two Integers

Our next program uses the Standard Library function `scanf` to obtain two integers typed by a user at the keyboard, computes the sum of these values and prints the result using `printf`. The program and sample output are shown in Fig. 2.5. [In the input/output dialog of Fig. 2.5, we emphasize the numbers entered by the user in **bold**.]

```
1 // Fig. 2.5: fig02_05.c
2 // Addition program.
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main( void )
7 {
8     int integer1; // first number to be entered by user
9     int integer2; // second number to be entered by user
10    int sum; // variable in which sum will be stored
11
12    printf( "Enter first integer\n" ); // prompt
13    scanf( "%d", &integer1 ); // read an integer
14
15    printf( "Enter second integer\n" ); // prompt
16    scanf( "%d", &integer2 ); // read an integer
17
18    sum = integer1 + integer2; // assign total to sum
19
20    printf( "Sum is %d\n", sum ); // print sum
21 } // end function main
```

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Fig. 2.5 | Addition program.

The comment in lines 1–2 states the purpose of the program. As we stated earlier, every program begins execution with `main`. The left brace `{` (line 7) marks the beginning of the body of `main`, and the corresponding right brace `}` (line 21) marks the end of `main`.

Variables and Variable Definitions

Lines 8–10

```
int integer1; // first number to be entered by user
int integer2; // second number to be entered by user
int sum; // variable in which sum will be stored
```

are **definitions**. The names `integer1`, `integer2` and `sum` are the names of **variables**—locations in memory where values can be stored for use by a program. These definitions specify that variables `integer1`, `integer2` and `sum` are of type **int**, which means that they'll hold **integer** values, i.e., whole numbers such as 7, -11, 0, 31914 and the like.

All variables must be defined with a name and a data type *before* they can be used in a program. For readers using the Microsoft Visual C++ compiler, note that we're placing our variable definitions immediately after the left brace that begins the body of `main`. The C standard allows you to place each variable definition *anywhere* in `main` before that variable's first use in the code. Some compilers, such as GNU `gcc`, have implemented this capability. We'll address this issue in more depth in later chapters.

The preceding definitions could have been combined into a single definition statement as follows:

```
int integer1, integer2, sum;
```

but that would have made it difficult to describe the variables with corresponding comments as we did in lines 8–10.

Identifiers and Case Sensitivity

A variable name in C is any valid **identifier**. An identifier is a series of characters consisting of letters, digits and underscores (`_`) that does *not* begin with a digit. C is **case sensitive**—uppercase and lowercase letters are *different* in C, so `a1` and `A1` are *different* identifiers.



Common Programming Error 2.2

Using a capital letter where a lowercase letter should be used (for example, typing `Main` instead of `main`).



Error-Prevention Tip 2.1

Avoid starting identifiers with the underscore character (`_`) to prevent conflicts with compiler-generated identifiers and standard library identifiers.



Good Programming Practice 2.5

Choosing meaningful variable names helps make a program self-documenting—that is, fewer comments are needed.



Good Programming Practice 2.6

The first letter of an identifier used as a simple variable name should be a lowercase letter. Later in the text we'll assign special significance to identifiers that begin with a capital letter and to identifiers that use all capital letters.

**Good Programming Practice 2.7**

Multiple-word variable names can help make a program more readable. Separate the words with underscores as in `total_commissions`, or, if you run the words together, begin each word after the first with a capital letter as in `totalCommissions`. The latter style is preferred.

Syntax Errors

We discussed what syntax errors are in Chapter 1. Recall that the Microsoft Visual C++ compiler requires variable definitions to be placed *after* the left brace of a function and *before* any executable statements. Therefore, in the program in Fig. 2.5, inserting the definition of `integer1` *after* the first `printf` would cause a syntax error in Visual C++.

**Common Programming Error 2.3**

Placing variable definitions among executable statements causes syntax errors in the Microsoft Visual C++ Compiler.

Prompting Messages

Line 12

```
printf( "Enter first integer\n" ); // prompt
```

displays the literal "Enter first integer" and positions the cursor to the beginning of the next line. This message is called a **prompt** because it tells the user to take a specific action.

The `scanf` Function and Formatted Inputs

The next statement

```
scanf( "%d", &integer1 ); // read an integer
```

uses **scanf** (the "f" stands for "formatted") to obtain a value from the user. The function reads from the *standard input*, which is usually the keyboard. This `scanf` has two arguments, `"%d"` and `&integer1`. The first, the **format control string**, indicates the *type* of data that should be entered by the user. The **%d conversion specifier** indicates that the data should be an integer (the letter `d` stands for "decimal integer"). The `%` in this context is treated by `scanf` (and `printf` as we'll see) as a special character that begins a conversion specifier. The second argument of `scanf` begins with an ampersand (`&`)—called the **address operator**—followed by the variable name. The `&`, when combined with the variable name, tells `scanf` the location (or address) in memory at which the variable `integer1` is stored. The computer then stores the value that the user enters for `integer1` at that location. The use of ampersand (`&`) is often confusing to novice programmers or to people who have programmed in other languages that do not require this notation. For now, just remember to precede each variable in every call to `scanf` with an ampersand. Some exceptions to this rule are discussed in Chapters 6 and 7. The use of the ampersand will become clear after we study *pointers* in Chapter 7.

**Good Programming Practice 2.8**

Place a space after each comma (,) to make programs more readable.

When the computer executes the preceding `scanf`, it waits for the user to enter a value for variable `integer1`. The user responds by typing an integer, then pressing the **Enter key** to send the number to the computer. The computer then assigns this number, or value, to

the variable `integer1`. Any subsequent references to `integer1` in this program will use this same value. Functions `printf` and `scanf` facilitate interaction between the user and the computer. Because this interaction resembles a dialogue, it's often called **interactive computing**.

Line 15

```
printf( "Enter second integer\n" ); // prompt
```

displays the message Enter second integer on the screen, then positions the cursor to the beginning of the next line. This `printf` also prompts the user to take action.

Line 16

```
scanf( "%d", &integer2 ); // read an integer
```

obtains a value for variable `integer2` from the user.

Assignment Statement

The **assignment statement** in line 18

```
sum = integer1 + integer2; // assign total to sum
```

calculates the total of variables `integer1` and `integer2` and assigns the result to variable `sum` using the assignment operator `=`. The statement is read as, “`sum` gets the value of `integer1 + integer2`.” Most calculations are performed in assignments. The `=` operator and the `+` operator are called *binary* operators because each has **two operands**. The `+` operator's two operands are `integer1` and `integer2`. The `=` operator's two operands are `sum` and the value of the expression `integer1 + integer2`.



Good Programming Practice 2.9

Place spaces on either side of a binary operator. This makes the operator stand out and makes the program more readable.



Common Programming Error 2.4

A calculation in an assignment statement must be on the right side of the `=` operator. It's a compilation error to place a calculation on the left side of an assignment operator.

Printing with a Format Control String

Line 20

```
printf( "Sum is %d\n", sum ); // print sum
```

calls function `printf` to print the literal `Sum is` followed by the numerical value of variable `sum` on the screen. This `printf` has two arguments, `"Sum is %d\n"` and `sum`. The first argument is the format control string. It contains some literal characters to be displayed, and it contains the conversion specifier `%d` indicating that an integer will be printed. The second argument specifies the value to be printed. Notice that the conversion specifier for an integer is the same in both `printf` and `scanf`—this is the case for most C data types.

Calculations in `printf` Statements

Calculations can also be performed inside `printf` statements. We could have combined the previous two statements into the statement

```
printf( "Sum is %d\n", integer1 + integer2 );
```

The right brace, `}`, at line 21 indicates that the end of function `main` has been reached.



Common Programming Error 2.5

Forgetting to precede a variable in a scanf statement with an ampersand when that variable should, in fact, be preceded by an ampersand results in an execution-time error. On many systems, this causes a “segmentation fault” or “access violation.” Such an error occurs when a user’s program attempts to access a part of the computer’s memory to which it does not have access privileges. The precise cause of this error will be explained in Chapter 7.



Common Programming Error 2.6

Preceding a variable included in a printf statement with an ampersand when, in fact, that variable should not be preceded by an ampersand.

2.4 Memory Concepts

Variable names such as `integer1`, `integer2` and `sum` actually correspond to locations in the computer’s memory. Every variable has a name, a **type** and a **value**.

In the addition program of Fig. 2.5, when the statement (line 13)

```
scanf( "%d", &integer1 ); // read an integer
```

is executed, the value entered by the user is placed into a memory location to which the name `integer1` has been assigned. Suppose the user enters the number 45 as the value for `integer1`. The computer will place 45 into location `integer1`, as shown in Fig. 2.6.

integer1	45
----------	----

Fig. 2.6 | Memory location showing the name and value of a variable.

Whenever a value is placed in a memory location, the value *replaces* the previous value in that location; thus, this process is said to be **destructive**.

Returning to our addition program again, when the statement (line 16)

```
scanf( "%d", &integer2 ); // read an integer
```

executes, suppose the user enters the value 72. This value is placed into location `integer2`, and memory appears as in Fig. 2.7. These locations are not necessarily adjacent in memory.

Once the program has obtained values for `integer1` and `integer2`, it adds these values and places the total into variable `sum`. The statement (line 18)

```
sum = integer1 + integer2; // assign total to sum
```

integer1	45
----------	----

integer2	72
----------	----

Fig. 2.7 | Memory locations after both variables are input.

that performs the addition also *replaces* whatever value was stored in `sum`. This occurs when the calculated total of `integer1` and `integer2` is placed into location `sum` (destroying the value already in `sum`). After `sum` is calculated, memory appears as in Fig. 2.8. The values of `integer1` and `integer2` appear exactly as they did *before* they were used in the calculation. They were used, but not destroyed, as the computer performed the calculation. Thus, when a value is *read* from a memory location, the process is said to be **nondestructive**.

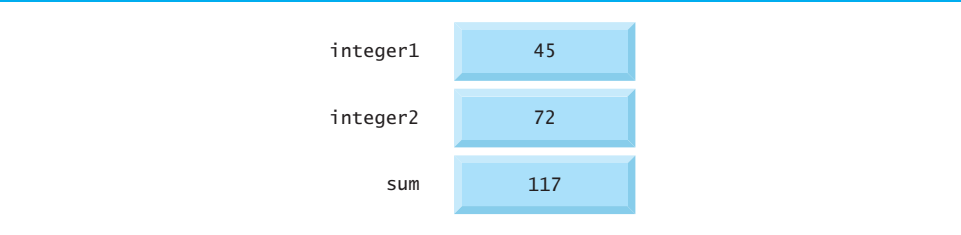


Fig. 2.8 | Memory locations after a calculation.

2.5 Arithmetic in C

Most C programs perform calculations using the C **arithmetic operators** (Fig. 2.9). Note the use of various special symbols not used in algebra. The **asterisk** (`*`) indicates *multiplication* and the **percent sign** (`%`) denotes the *remainder operator*, which is introduced below. In algebra, to multiply *a* times *b*, we simply place these single-letter variable names side by side, as in *ab*. In C, however, if we were to do this, *ab* would be interpreted as a single, two-letter name (or identifier). Therefore, C (and many other programming languages) require that multiplication be explicitly denoted by using the `*` operator, as in *a * b*. The arithmetic operators are all *binary* operators. For example, the expression `3 + 7` contains the binary operator `+` and the operands 3 and 7.

C operation	Arithmetic operator	Algebraic expression	C expression
Addition	<code>+</code>	$f + 7$	<code>f + 7</code>
Subtraction	<code>-</code>	$p - c$	<code>p - c</code>
Multiplication	<code>*</code>	bm	<code>b * m</code>
Division	<code>/</code>	x / y or $\frac{x}{y}$ or $x \div y$	<code>x / y</code>
Remainder	<code>%</code>	$r \bmod s$	<code>r % s</code>

Fig. 2.9 | Arithmetic operators.

Integer Division and the Remainder Operator

Integer division yields an integer result. For example, the expression `7 / 4` evaluates to 1 and the expression `17 / 5` evaluates to 3. C provides the **remainder operator**, `%`, which yields the *remainder* after integer division. The remainder operator is an integer operator that can be used only with integer operands. The expression `x % y` yields the remainder after *x* is divided by *y*. Thus, `7 % 4` yields 3 and `17 % 5` yields 2. We'll discuss many interesting applications of the remainder operator.



Common Programming Error 2.7

An attempt to divide by zero is normally undefined on computer systems and generally results in a fatal error, i.e., an error that causes the program to terminate immediately without having successfully performed its job. Nonfatal errors allow programs to run to completion, often producing incorrect results.

Arithmetic Expressions in Straight-Line Form

Arithmetic expressions in C must be written in **straight-line form** to facilitate entering programs into the computer. Thus, expressions such as “a divided by b” must be written as `a/b` so that all operators and operands appear in a straight line. The algebraic notation

$$\frac{a}{b}$$

is generally not acceptable to compilers, although some special-purpose software packages do support more natural notation for complex mathematical expressions.

Parentheses for Grouping Subexpressions

Parentheses are used in C expressions in the same manner as in algebraic expressions. For example, to multiply a times the quantity `b + c` we write `a * (b + c)`.

Rules of Operator Precedence

C applies the operators in arithmetic expressions in a precise sequence determined by the following **rules of operator precedence**, which are generally the same as those in algebra:

1. Operators in expressions contained within pairs of parentheses are evaluated first. Parentheses are said to be at the “highest level of precedence.” In cases of **nested**, or **embedded**, **parentheses**, such as

$$((a + b) + c)$$

the operators in the *innermost* pair of parentheses are applied first.

2. Multiplication, division and remainder operations are applied next. If an expression contains several multiplication, division and remainder operations, evaluation proceeds from left to right. Multiplication, division and remainder are said to be on the same level of precedence.
3. Addition and subtraction operations are evaluated next. If an expression contains several addition and subtraction operations, evaluation proceeds from left to right. Addition and subtraction also have the same level of precedence, which is lower than the precedence of the multiplication, division and remainder operations.
4. The assignment operator (`=`) is evaluated last.

The rules of operator precedence specify the order C uses to evaluate expressions.¹ When we say evaluation proceeds from left to right, we’re referring to the **associativity** of the operators. We’ll see that some operators associate from right to left. Figure 2.10 summarizes these rules of operator precedence for the operators we’ve seen so far.

1. We use simple examples to explain the order of evaluation of expressions. Subtle issues occur in more complex expressions that you’ll encounter later in the book. We’ll discuss these issues as they arise.

Operator(s)	Operation(s)	Order of evaluation (precedence)
()	Parentheses	Evaluated first. If the parentheses are nested, the expression in the <i>innermost</i> pair is evaluated first. If there are several pairs of parentheses “on the same level” (i.e., not nested), they’re evaluated left to right.
*	Multiplication	Evaluated second. If there are several, they’re evaluated left to right.
/	Division	
%	Remainder	
+	Addition	Evaluated third. If there are several, they’re evaluated left to right.
-	Subtraction	
=	Assignment	Evaluated last.

Fig. 2.10 | Precedence of arithmetic operators.

Sample Algebraic and C Expressions

Now let’s consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its C equivalent. The following expression calculates the arithmetic mean (average) of five terms.

Algebra:	$m = \frac{a + b + c + d + e}{5}$
C:	<code>m = (a + b + c + d + e) / 5;</code>

The parentheses are required to group the additions because division has higher precedence than addition. The entire quantity (a + b + c + d + e) should be divided by 5. If the parentheses are erroneously omitted, we obtain a + b + c + d + e / 5, which evaluates incorrectly as

$$a + b + c + d + \frac{e}{5}$$

The following expression is the equation of a straight line:

Algebra:	$y = mx + b$
C:	<code>y = m * x + b;</code>

No parentheses are required. The multiplication is evaluated first because multiplication has a higher precedence than addition.

The following expression contains remainder (%), multiplication, division, addition, subtraction and assignment operations:

Algebra:	$z = pr \% q + w / x - y$
C:	<code>z = p * r % q + w / x - y;</code>
	<div><div>6</div><div>1</div><div>2</div><div>4</div><div>3</div><div>5</div></div>

The circled numbers indicate the order in which C evaluates the operators. The multiplication, remainder and division are evaluated first in left-to-right order (i.e., they associate

from left to right) because they have higher precedence than addition and subtraction. The addition and subtraction are evaluated next. They're also evaluated left to right. Finally, the result is assigned to the variable *z*.

Not all expressions with several pairs of parentheses contain nested parentheses. For example, the following expression does *not* contain nested parentheses—instead, the parentheses are said to be “on the same level.”

```
a * ( b + c ) + c * ( d + e )
```

Evaluation of a Second-Degree Polynomial

To develop a better understanding of the rules of operator precedence, let's see how C evaluates a second-degree polynomial.

```
y = a * x * x + b * x + c;
```

The circled numbers under the statement indicate the order in which C performs the operations. There's no arithmetic operator for exponentiation in C, so we've represented x^2 as $x * x$. The C Standard Library includes the `pow` (“power”) function to perform exponentiation. Because of some subtle issues related to the data types required by `pow`, we defer a detailed explanation of `pow` until Chapter 4.

Suppose variables *a*, *b*, *c* and *x* in the preceding second-degree polynomial are initialized as follows: *a* = 2, *b* = 3, *c* = 7 and *x* = 5. Figure 2.11 illustrates the order in which the operators are applied.

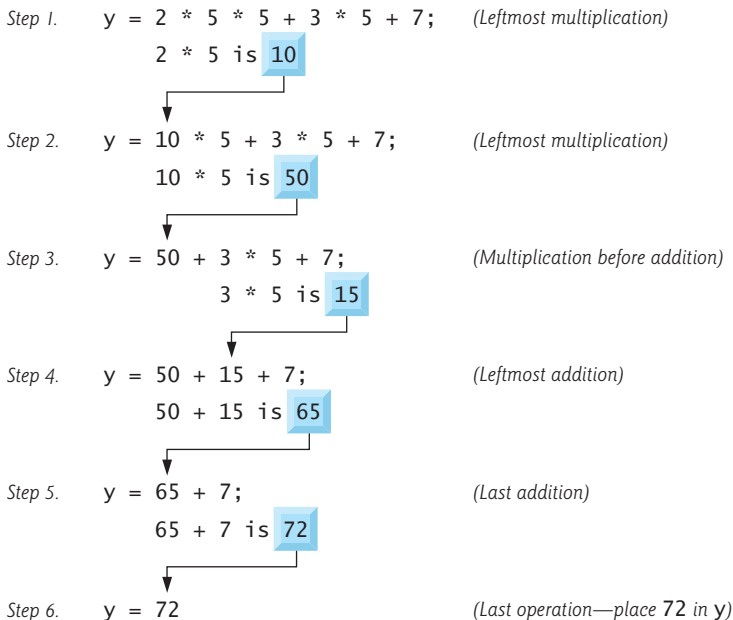


Fig. 2.11 | Order in which a second-degree polynomial is evaluated.


As in algebra, it's acceptable to place unnecessary parentheses in an expression to make the expression clearer. These are called **redundant parentheses**. For example, the preceding statement could be parenthesized as follows:

```
y = ( a * x * x ) + ( b * x ) + c ;
```


2.6 Decision Making: Equality and Relational Operators

Executable statements either perform actions (such as calculations or input or output of data) or make **decisions** (we'll soon see several examples of these). We might make a decision in a program, for example, to determine whether a person's grade on an exam is greater than or equal to 60 and whether the program should print the message "Congratulations! You passed." This section introduces a simple version of C's **if statement** that allows a program to make a decision based on the truth or falsity of a statement of fact called a **condition**. If the condition is **true** (i.e., the condition is met), the statement in the body of the if statement is executed. If the condition is **false** (i.e., the condition isn't met), the body statement isn't executed. Whether the body statement is executed or not, after the if statement completes, execution proceeds with the next statement after the if statement.

Conditions in if statements are formed by using the **equality operators** and **relational operators** summarized in Fig. 2.12. The relational operators all have the same level of precedence and they associate left to right. The equality operators have a lower level of precedence than the relational operators and they also associate left to right. [Note: In C, a condition may actually be any expression that generates a zero (false) or nonzero (true) value.]



Common Programming Error 2.8
A syntax error occurs if the two symbols in any of the operators ==, !=, >= and <= are separated by spaces.



Common Programming Error 2.9
Confusing the equality operator == with the assignment operator. To avoid this confusion, the equality operator should be read "double equals" and the assignment operator should be read "gets" or "is assigned the value of." As you'll see, confusing these operators may not cause an easy-to-recognize compilation error, but may cause extremely subtle logic errors.

Algebraic equality or relational operator	C equality or relational operator	Example of C condition	Meaning of C condition
<i>Equality operators</i>			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
<i>Relational operators</i>			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Fig. 2.12 | Equality and relational operators.

Figure 2.13 uses six if statements to compare two numbers entered by the user. If the condition in any of these if statements is true, the printf statement associated with that if executes. The program and three sample execution outputs are shown in the figure.

```

1 // Fig. 2.13: fig02_13.c
2 // Using if statements, relational
3 // operators, and equality operators.
4 #include <stdio.h>
5
6 // function main begins program execution
7 int main( void )
8 {
9     int num1; // first number to be read from user
10    int num2; // second number to be read from user
11
12    printf( "Enter two integers, and I will tell you\n" );
13    printf( "the relationships they satisfy: " );
14
15    scanf( "%d%d", &num1, &num2 ); // read two integers
16
17    if ( num1 == num2 ) {
18        printf( "%d is equal to %d\n", num1, num2 );
19    } // end if
20
21    if ( num1 != num2 ) {
22        printf( "%d is not equal to %d\n", num1, num2 );
23    } // end if
24
25    if ( num1 < num2 ) {
26        printf( "%d is less than %d\n", num1, num2 );
27    } // end if
28
29    if ( num1 > num2 ) {
30        printf( "%d is greater than %d\n", num1, num2 );
31    } // end if
32
33    if ( num1 <= num2 ) {
34        printf( "%d is less than or equal to %d\n", num1, num2 );
35    } // end if
36
37    if ( num1 >= num2 ) {
38        printf( "%d is greater than or equal to %d\n", num1, num2 );
39    } // end if
40 } // end function main

```

```

Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7

```

Fig. 2.13 | Using if statements, relational operators, and equality operators. (Part I of 2.)

```

Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12

```

```

Enter two integers, and I will tell you
the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7

```

Fig. 2.13 | Using `if` statements, relational operators, and equality operators. (Part 2 of 2.)

The program uses `scanf` (line 15) to input two numbers. Each conversion specifier has a corresponding argument in which a value will be stored. The first `%d` converts a value to be stored in the variable `num1`, and the second `%d` converts a value to be stored in the variable `num2`.



Good Programming Practice 2.10

Although it's allowed, there should be no more than one statement per line in a program.



Common Programming Error 2.10

Placing commas (when none are needed) between conversion specifiers in the format control string of a `scanf` statement.

Comparing Numbers

The `if` statement in lines 17–19

```

if ( num1 == num2 ) {
    printf( "%d is equal to %d\n", num1, num2 );
}

```

compares the values of variables `num1` and `num2` to test for equality. If the values are equal, the statement in line 18 displays a line of text indicating that the numbers are equal. If the conditions are true in one or more of the `if` statements starting in lines 21, 25, 29, 33 and 37, the corresponding body statement displays an appropriate line of text. Indenting the body of each `if` statement and placing blank lines above and below each `if` statement enhances program readability.



Common Programming Error 2.11

Placing a semicolon immediately to the right of the right parenthesis after the condition in an `if` statement.

A left brace, `{`, begins the body of each `if` statement (e.g., line 17). A corresponding right brace, `}`, ends each `if` statement's body (e.g., line 19). Any number of statements can be placed in the body of an `if` statement.²



Good Programming Practice 2.11

A lengthy statement may be spread over several lines. If a statement must be split across lines, choose breaking points that make sense (such as after a comma in a comma-separated list). If a statement is split across two or more lines, indent all subsequent lines. It's not correct to split identifiers.

Figure 2.14 lists from highest to lowest the precedence of the operators introduced in this chapter. Operators are shown top to bottom in decreasing order of precedence. The equals sign is also an operator. All these operators, with the exception of the assignment operator `=`, associate from left to right. The assignment operator (`=`) associates from right to left.



Good Programming Practice 2.12

Refer to the operator precedence chart when writing expressions containing many operators. Confirm that the operators in the expression are applied in the proper order. If you're uncertain about the order of evaluation in a complex expression, use parentheses to group expressions or break the statement into several simpler statements. Be sure to observe that some of C's operators such as the assignment operator (`=`) associate from right to left rather than from left to right.

Operators	Associativity
()	left to right
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
=	right to left

Fig. 2.14 | Precedence and associativity of the operators discussed so far.

Some of the words we've used in the C programs in this chapter—in particular `int` and `if`—are **keywords** or reserved words of the language. Figure 2.15 contains the C keywords. These words have special meaning to the C compiler, so you must be careful not to use these as identifiers such as variable names.

In this chapter, we've introduced many important features of the C programming language, including displaying data on the screen, inputting data from the user, performing calculations and making decisions. In the next chapter, we build upon these techniques as we introduce structured programming. You'll become more familiar with indentation techniques. We'll study how to specify the *order in which statements are executed*—this is called **flow of control**.

- Using braces to delimit the body of an `if` statement is optional when the body contains only one statement. Many programmers consider it good practice to always use these braces. In Chapter 3, we'll explain the issues.

Keywords			
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
<i>Keywords added in C99 standard</i>			
_Bool _Complex _Imaginary inline restrict			
<i>Keywords added in C11 draft standard</i>			
_Alignas _Alignof _Atomic _Generic _Noreturn _Static_assert _Thread_local			

Fig. 2.15 | C's keywords.

2.7 Secure C Programming

We mentioned *The CERT C Secure Coding Standard* in the Preface and indicated that we would follow certain guidelines that will help you avoid programming practices that open systems to attacks.

*Avoid Single-Argument `printf`s*³

One such guideline is to *avoid using `printf` with a single string argument*. If you need to display a string that *terminates with a newline*, use the **`puts` function**, which displays its string argument followed by a newline character. For example, in Fig. 2.1, line 8

```
printf( "Welcome to C!\n" );
```

should be written as:

```
puts( "Welcome to C!" );
```

We did not include `\n` in the preceding string because `puts` adds it automatically.

If you need to display a string *without* a terminating newline character, use `printf` with *two* arguments—a `%s` format control string and the string to display. The **`%s` conversion specifier** is for displaying a string. For example, in Fig. 2.3, line 8

```
printf( "Welcome " );
```

should be written as:

```
printf( "%s", "Welcome " );
```

3. For more information, see CERT C Secure Coding rule FIO30-C (www.securecoding.cert.org/confluence/display/seccode/FIO30-C.+Exclude+user+input+from+format+strings). In Chapter 6's Secure C Programming section, we'll explain the notion of user input as referred to by this CERT guideline.

Although the `printf`s in this chapter as written are actually *not* insecure, these changes are responsible coding practices that will eliminate certain security vulnerabilities as we get deeper into C—we'll explain the rationale later in the book. From this point forward, we use these practices in the chapter examples and you should use them in your exercise solutions.

scanf and printf, scanf_s and printf_s

We introduced `scanf` and `printf` in this chapter. We'll be saying more about these in subsequent Secure C Coding Guidelines sections. We'll also discuss `scanf_s` and `printf_s`, which were introduced in C11.

Summary

Section 2.1 Introduction

- The C language facilitates a structured and disciplined approach to computer-program design.

Section 2.2 A Simple C Program: Printing a Line of Text

- Comments begin with `//`. Comments document programs and improve program readability. C also supports older-style multi-line comments that begin with `/*` and end with `*/`.
- Comments do not cause the computer to perform any action when the program is run. They're ignored by the C compiler and do not cause any machine-language object code to be generated.
- Lines beginning with `#` are processed by the preprocessor before the program is compiled. The `#include` directive tells the preprocessor to include the contents of another file.
- The `<stdio.h>` header contains information used by the compiler when compiling calls to standard input/output library functions such as `printf`.
- The function `main` is a part of every C program. The parentheses after `main` indicate that `main` is a program building block called a function. C programs contain one or more functions, one of which must be `main`. Every program in C begins executing at the function `main`.
- Functions can return information. The keyword `int` to the left of `main` indicates that `main` "returns" an integer (whole-number) value.
- Functions can receive information when they're called upon to execute. The `void` in parentheses after `main` indicates that `main` does not receive any information.
- A left brace, `{`, begins the body of every function. A corresponding right brace, `}`, ends each function. This pair of braces and the portion of the program between the braces is called a block.
- The `printf` function instructs the computer to display information on the screen.
- A string is sometimes called a character string, a message or a literal.
- Every statement must end with a semicolon (also known as the statement terminator).
- In `\n`, the backslash (`\`) is called an escape character. When encountering a backslash in a string, the compiler looks ahead at the next character and combines it with the backslash to form an escape sequence. The escape sequence `\n` means newline.
- When a newline appears in the string output by a `printf`, the newline causes the cursor to position to the beginning of the next line on the screen.
- The double backslash (`\\`) escape sequence can be used to place a single backslash in a string.
- The escape sequence `\"` represents a literal double-quote character.

Section 2.3 Another Simple C Program: Adding Two Integers

- A variable is a location in memory where a value can be stored for use by a program.

- Variables of type `int` hold integer values, i.e., whole numbers such as 7, -11, 0, 31914.
- All variables must be defined with a name and a data type before they can be used in a program.
- A variable name in C is any valid identifier. An identifier is a series of characters consisting of letters, digits and underscores (`_`) that does not begin with a digit.
- C is case sensitive—uppercase and lowercase letters are different in C.
- Microsoft Visual C++ requires variable definitions in C programs to be placed after the left brace of a function and before *any* executable statements. GNU gcc and some other compilers do not have this restriction.
- A syntax error is caused when the compiler cannot recognize a statement. The compiler normally issues an error message to help you locate and fix the incorrect statement. Syntax errors are violations of the language. Syntax errors are also called compile errors, or compile-time errors.
- Standard Library function `scanf` can be used to obtain input from the standard input, which is usually the keyboard.
- The `scanf` format control string indicates the type(s) of data that should be input.
- The `%d` conversion specifier indicates that the data should be an integer (the letter `d` stands for “decimal integer”). The `%` in this context is treated by `scanf` (and `printf`) as a special character that begins a conversion specifier.
- The arguments that follow `scanf`’s format control string begin with an ampersand (`&`)—called the address operator in C—followed by a variable name. The ampersand, when combined with a variable name, tells `scanf` the location in memory at which the variable is located. The computer then stores the value for the variable at that location.
- Most calculations are performed in assignment statements.
- The `=` operator and the `+` operator are binary operators—each has two operands.
- In a `printf` that specifies a format control string as its first argument the conversion specifiers indicate placeholders for data to output.

Section 2.4 Memory Concepts

- Variable names correspond to locations in the computer’s memory. Every variable has a name, a type and a value.
- Whenever a value is placed in a memory location, the value replaces the previous value in that location; thus, placing a new value into a memory location is said to be destructive.
- When a value is read from a memory location, the process is said to be nondestructive.

Section 2.5 Arithmetic in C

- In algebra, if we want to multiply a times b , we can simply place these single-letter variable names side by side as in ab . In C, however, if we were to do this, ab would be interpreted as a single, two-letter name (or identifier). Therefore, C (like other programming languages, in general) requires that multiplication be explicitly denoted by using the `*` operator, as in $a * b$.
- Arithmetic expressions in C must be written in straight-line form to facilitate entering programs into the computer. Thus, expressions such as “ a divided by b ” must be written as a/b , so that all operators and operands appear in a straight line.
- Parentheses are used to group terms in C expressions in much the same manner as in algebraic expressions.
- C evaluates arithmetic expressions in a precise sequence determined by the following rules of operator precedence, which are generally the same as those followed in algebra.

- Multiplication, division and remainder operations are applied first. If an expression contains several multiplication, division and remainder operations, evaluation proceeds from left to right. Multiplication, division and remainder are said to be on the same level of precedence.
- Addition and subtraction operations are evaluated next. If an expression contains several addition and subtraction operations, evaluation proceeds from left to right. Addition and subtraction also have the same level of precedence, which is lower than the precedence of the multiplication, division and remainder operators.
- The rules of operator precedence specify the order C uses to evaluate expressions. The associativity of the operators specifies whether they evaluate from left to right or from right to left.

Section 2.6 Decision Making: Equality and Relational Operators

- Executable C statements either perform actions or make decisions.
- C's `if` statement allows a program to make a decision based on the truth or falsity of a statement of fact called a condition. If the condition is met (i.e., the condition is true) the statement in the body of the `if` statement executes. If the condition isn't met (i.e., the condition is false) the body statement does not execute. Whether the body statement is executed or not, after the `if` statement completes, execution proceeds with the next statement after the `if` statement.
- Conditions in `if` statements are formed by using the equality operators and relational operators.
- The relational operators all have the same level of precedence and associate left to right. The equality operators have a lower level of precedence than the relational operators and they also associate left to right.
- To avoid confusing assignment (`=`) and equality (`==`), the assignment operator should be read "gets" and the equality operator should be read "double equals."
- In C programs, white-space characters such as tabs, newlines and spaces are normally ignored. So, statements may be split over several lines. It's not correct to split identifiers.
- Keywords (or reserved words) have special meaning to the C compiler, so you cannot use them as identifiers such as variable names.

Section 2.7 Secure C Programming

- One practice to help avoid leaving systems open to attacks is to avoid using `printf` with a single string argument.
- To display a string followed by a newline character, use the `puts` function, which displays its string argument followed by a newline character.
- To display a string without a trailing newline character, you can use `printf` the format string argument `"%s"` followed by a second argument representing the string to display. The conversion specification `%s` is for displaying a string.

Terminology

+ addition operator 50
 / division operator 50
 * multiplication operator 50
 % remainder operator 50
 - subtraction operator 50
 %d conversion specifier 47
 %s conversion specifier 58
 action 43
 action/decision model 43
 address operator (&) 47

argument 43
 arithmetic operators 50
 assignment statement 48
 associativity 51
 body 42
 C preprocessor 42
 case sensitive 46
 character string 43
 comment (//) 41
 comment (/...*/) 42

condition 54	nested parentheses 51
decision 43	newline (\n) 43
definition 46	nondestructive 50
destructive 49	operand 48
document a program 41	percent sign (%) 50
embedded parentheses 51	printf function 43
Enter key 47	prompt 47
equality operator 54	puts function 58
escape character 43	redundant parentheses 54
escape sequence 43	relational operator 54
executable 44	right brace { } 42
false 54	rules of operator precedence 51
flow of control 57	scanf function 47
format control string 47	single-line comment (//) 41
function 42	statement 43
identifier 46	statement terminator (;) 43
if statement 54	<stdio.h> (standard input/output) header 42
int type 46	straight-line form 51
integer 46	string 43
integer division 50	true 54
interactive computing 48	type 49
keyword 57	value 49
literal 43	variable 46
message 43	white space 42

Self-Review Exercises

- 2.1** Fill in the blanks in each of the following.
- Every C program begins execution at the function _____.
 - Every function's body begins with _____ and ends with _____.
 - Every statement ends with a(n) _____.
 - The _____ standard library function displays information on the screen.
 - The escape sequence \n represents the _____ character, which causes the cursor to position to the beginning of the next line on the screen.
 - The _____ Standard Library function is used to obtain data from the keyboard.
 - The conversion specifier _____ is used in a scanf format control string to indicate that an integer will be input and in a printf format control string to indicate that an integer will be output.
 - Whenever a new value is placed in a memory location, that value overrides the previous value in that location. This process is said to be _____.
 - When a value is read from a memory location, the value in that location is preserved; this process is said to be _____.
 - The _____ statement is used to make decisions.
- 2.2** State whether each of the following is *true* or *false*. If *false*, explain why.
- Function printf always begins printing at the beginning of a new line.
 - Comments cause the computer to display the text after // on the screen when the program is executed.
 - The escape sequence \n when used in a printf format control string causes the cursor to position to the beginning of the next line on the screen.
 - All variables must be defined before they're used.
 - All variables must be given a type when they're defined.

- f) C considers the variables `number` and `NumBEr` to be identical.
- g) Definitions can appear anywhere in the body of a function.
- h) All arguments following the format control string in a `printf` function must be preceded by an ampersand (&).
- i) The remainder operator (%) can be used only with integer operands.
- j) The arithmetic operators *, /, %, + and - all have the same level of precedence.
- k) A program that prints three lines of output must contain three `printf` statements.

2.3 Write a single C statement to accomplish each of the following:

- a) Define the variables `c`, `thisVariable`, `q76354` and `number` to be of type `int`.
- b) Prompt the user to enter an integer. End your prompting message with a colon (:) followed by a space and leave the cursor positioned after the space.
- c) Read an integer from the keyboard and store the value entered in integer variable `a`.
- d) If `number` is not equal to 7, print "The variable `number` is not equal to 7."
- e) Print the message "This is a C program." on one line.
- f) Print the message "This is a C program." on two lines so that the first line ends with C.
- g) Print the message "This is a C program." with each word on a separate line.
- h) Print the message "This is a C program." with the words separated by tabs.

2.4 Write a statement (or comment) to accomplish each of the following:

- a) State that a program will calculate the product of three integers.
- b) Define the variables `x`, `y`, `z` and `result` to be of type `int`.
- c) Prompt the user to enter three integers.
- d) Read three integers from the keyboard and store them in the variables `x`, `y` and `z`.
- e) Compute the product of the three integers contained in variables `x`, `y` and `z`, and assign the result to the variable `result`.
- f) Print "The product is" followed by the value of the integer variable `result`.

2.5 Using the statements you wrote in Exercise 2.4, write a complete program that calculates the product of three integers.

2.6 Identify and correct the errors in each of the following statements:

- a) `printf("The value is %d\n", &number);`
- b) `scanf("%d%d", &number1, number2);`
- c) `if (c < 7);{`
`printf("C is less than 7\n");`
`}`
- d) `if (c == 7) {`
`printf("C is greater than or equal to 7\n");`
`}`

Answers to Self-Review Exercises

2.1 a) `main`. b) left brace (`{`), right brace (`}`). c) semicolon. d) `printf`. e) newline. f) `scanf`. g) `%d`. h) destructive. i) nondestructive. j) `if`.

- 2.2** a) False. Function `printf` always begins printing where the cursor is positioned, and this may be anywhere on a line of the screen.
- b) False. Comments do not cause any action to be performed when the program is executed. They're used to document programs and improve their readability.
- c) True.
- d) True.
- e) True.
- f) False. C is case sensitive, so these variables are unique.

- g) False. A variable's definition must appear before its first use in the code. In Microsoft Visual C++, variable definitions must appear immediately following the left brace that begins the body of `main`. Later in the book we'll discuss this in more depth as we encounter additional C features that can affect this issue.
- h) False. Arguments in a `printf` function ordinarily should not be preceded by an ampersand. Arguments following the format control string in a `scanf` function ordinarily should be preceded by an ampersand. We'll discuss exceptions to these rules in Chapter 6 and Chapter 7.
- i) True.
- j) False. The operators `*`, `/` and `%` are on the same level of precedence, and the operators `+` and `-` are on a lower level of precedence.
- k) False. A `printf` statement with multiple `\n` escape sequences can print several lines.

2.3

- a) `int c, thisVariable, q76354, number;`
- b) `printf("Enter an integer: ");`
- c) `scanf("%d", &a);`
- d) `if (number != 7) {`
`printf("The variable number is not equal to 7.\n");`
`}`
- e) `printf("This is a C program.\n");`
- f) `printf("This is a C\nprogram.\n");`
- g) `printf("This\nis\nna\nC\nprogram.\n");`
- h) `printf("This\tis\ta\tC\tprogram.\n");`

2.4

- a) `// Calculate the product of three integers`
- b) `int x, y, z, result;`
- c) `printf("Enter three integers: ");`
- d) `scanf("%d%d%d", &x, &y, &z);`
- e) `result = x * y * z;`
- f) `printf("The product is %d\n", result);`

2.5

See below.

```

1 // Calculate the product of three integers
2 #include <stdio.h>
3
4 int main( void )
5 {
6     int x, y, z, result; // declare variables
7
8     printf( "Enter three integers: " ); // prompt
9     scanf( "%d%d%d", &x, &y, &z ); // read three integers
10    result = x * y * z; // multiply values
11    printf( "The product is %d\n", result ); // display result
12 } // end function main

```

2.6

- a) Error: `&number`.
Correction: Eliminate the `&`. We discuss exceptions to this later.
- b) Error: `number2` does not have an ampersand.
Correction: `number2` should be `&number2`. Later in the text we discuss exceptions to this.
- c) Error: Semicolon after the right parenthesis of the condition in the `if` statement.
Correction: Remove the semicolon after the right parenthesis. [Note: The result of this error is that the `printf` statement will be executed whether or not the condition in the

if statement is true. The semicolon after the right parenthesis is considered an empty statement—a statement that does nothing.]

- d) Error: `=>` is not an operator in C.

Correction: The relational operator `=>` should be changed to `>=` (greater than or equal to).

Exercises

2.7 Identify and correct the errors in each of the following statements. (*Note:* There may be more than one error per statement.)

- a) `scanf("d", value);`
- b) `printf("The product of %d and %d is %d\n", x, y);`
- c) `firstNumber + secondNumber = sumOfNumbers`
- d) `if (number => largest)`
 `largest == number;`
- e) `/* Program to determine the largest of three integers */`
- f) `Scanf("%d", anInteger);`
- g) `printf("Remainder of %d divided by %d is\n", x, y, x % y);`
- h) `if (x = y);`
 `printf(%d is equal to %d\n", x, y);`
- i) `print("The sum is %d\n," x + y);`
- j) `Printf("The value you entered is: %d\n", &value);`

2.8 Fill in the blanks in each of the following:

- a) _____ are used to document a program and improve its readability.
- b) The function used to display information on the screen is _____.
- c) A C statement that makes a decision is _____.
- d) Calculations are normally performed by _____ statements.
- e) The _____ function inputs values from the keyboard.

2.9 Write a single C statement or line that accomplishes each of the following:

- a) Print the message "Enter two numbers."
- b) Assign the product of variables `b` and `c` to variable `a`.
- c) State that a program performs a sample payroll calculation (i.e., use text that helps to document a program).
- d) Input three integer values from the keyboard and place them in integer variables `a`, `b` and `c`.

2.10 State which of the following are *true* and which are *false*. If *false*, explain your answer.

- a) C operators are evaluated from left to right.
- b) The following are all valid variable names: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales`, `his_account_total`, `a`, `b`, `c`, `z`, `z2`.
- c) The statement `printf("a = 5;");` is a typical example of an assignment statement.
- d) A valid arithmetic expression containing no parentheses is evaluated from left to right.
- e) The following are all invalid variable names: `3g`, `87`, `67h2`, `h22`, `2h`.

2.11 Fill in the blanks in each of the following:

- a) What arithmetic operations are on the same level of precedence as multiplication?
_____.
- b) When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression? _____.
- c) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a _____.

2.12 What, if anything, prints when each of the following statements is performed? If nothing prints, then answer "Nothing." Assume `x = 2` and `y = 3`.

- a) `printf("%d", x);`
- b) `printf("%d", x + x);`
- c) `printf("x=");`
- d) `printf("x=%d", x);`
- e) `printf("%d = %d", x + y, y + x);`
- f) `z = x + y;`
- g) `scanf("%d%d", &x, &y);`
- h) `// printf("x + y = %d", x + y);`
- i) `printf("\n");`

2.13 Which, if any, of the following C statements contain variables whose values are replaced?

- a) `scanf("%d%d%d%d%d", &b, &c, &d, &e, &f);`
- b) `p = i + j + k + 7;`
- c) `printf("Values are replaced");`
- d) `printf("a = 5");`

2.14 Given the equation $y = ax^3 + 7$, which of the following, if any, are correct C statements for this equation?

- a) `y = a * x * x * x + 7;`
- b) `y = a * x * x * (x + 7);`
- c) `y = (a * x) * x * (x + 7);`
- d) `y = (a * x) * x * x + 7;`
- e) `y = a * (x * x * x) + 7;`
- f) `y = a * x * (x * x + 7);`

2.15 State the order of evaluation of the operators in each of the following C statements and show the value of `x` after each statement is performed.

- a) `x = 7 + 3 * 6 / 2 - 1;`
- b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
- c) `x = (3 * 9 * (3 + (9 * 3 / (3))));`

2.16 (*Arithmetic*) Write a program that asks the user to enter two numbers, obtains them from the user and prints their sum, product, difference, quotient and remainder.

2.17 (*Printing Values with printf*) Write a program that prints the numbers 1 to 4 on the same line. Write the program using the following methods.

- a) Using one `printf` statement with no conversion specifiers.
- b) Using one `printf` statement with four conversion specifiers.
- c) Using four `printf` statements.

2.18 (*Comparing Integers*) Write a program that asks the user to enter two integers, obtains the numbers from the user, then prints the larger number followed by the words "is larger." If the numbers are equal, print the message "These numbers are equal." Use only the single-selection form of the `if` statement you learned in this chapter.

2.19 (*Arithmetic, Largest Value and Smallest Value*) Write a program that inputs three different integers from the keyboard, then prints the sum, the average, the product, the smallest and the largest of these numbers. Use only the single-selection form of the `if` statement you learned in this chapter. The screen dialogue should appear as follows:

```
Enter three different integers: 13 27 14
Sum is 54
Average is 18
Product is 4914
Smallest is 13
Largest is 27
```

2.20 (*Diameter, Circumference and Area of a Circle*) Write a program that reads in the radius of a circle and prints the circle's diameter, circumference and area. Use the constant value 3.14159 for π . Perform each of these calculations inside the `printf` statement(s) and use the conversion specifier `%f`. [Note: In this chapter, we've discussed only integer constants and variables. In Chapter 3 we'll discuss floating-point numbers, i.e., values that can have decimal points.]

2.21 (*Shapes with Asterisks*) Write a program that prints the following shapes with asterisks.

```

*****      ***      *      *
*      *      *      *      ***      *
*      *      *      *      *****
*      *      *      *      *      *
*      *      *      *      *      *
*      *      *      *      *      *
*      *      *      *      *      *
*      *      *      *      *      *
*****      ***      *      *

```

2.22 What does the following code print?

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

2.23 (*Largest and Smallest Integers*) Write a program that reads in three integers and then determines and prints the largest and the smallest integers in the group. Use only the programming techniques you have learned in this chapter.

2.24 (*Odd or Even*) Write a program that reads an integer and determines and prints whether it's odd or even. [Hint: Use the remainder operator. An even number is a multiple of two. Any multiple of two leaves a remainder of zero when divided by 2.]

2.25 Print your initials in block letters down the page. Construct each block letter out of the letter it represents, as shown below.

```

PPPPPPPP
 P  P
  P  P
   P  P
    P P

 JJ
 J
J
 J
  JJJJJJ

 DDDDDDDD
 D      D
 D      D
 D      D
 D      D
 DDDDD

```

2.26 (*Multiples*) Write a program that reads in two integers and determines and prints whether the first is a multiple of the second. [Hint: Use the remainder operator.]

2.27 (*Checkerboard Pattern of Asterisks*) Display the following checkerboard pattern with eight `printf` statements and then display the same pattern with as few `printf` statements as possible.

```

* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *

```

2.28 Distinguish between the terms fatal error and nonfatal error. Why might you prefer to experience a fatal error rather than a nonfatal error?

2.29 (*Integer Value of a Character*) Here's a peek ahead. In this chapter you learned about integers and the type `int`. C can also represent uppercase letters, lowercase letters and a considerable variety of special symbols. C uses small integers internally to represent each different character. The set of characters a computer uses together with the corresponding integer representations for those characters is called that computer's character set. You can print the integer equivalent of uppercase A, for example, by executing the statement

```
printf( "%d", 'A' );
```

Write a C program that prints the integer equivalents of some uppercase letters, lowercase letters, digits and special symbols. As a minimum, determine the integer equivalents of the following: A B C a b c 0 1 2 \$ * + / and the blank character.

2.30 (*Separating Digits in an Integer*) Write a program that inputs one five-digit number, separates the number into its individual digits and prints the digits separated from one another by three spaces each. [*Hint*: Use combinations of integer division and the remainder operation.] For example, if the user types in 42139, the program should print

```
4   2   1   3   9
```

2.31 (*Table of Squares and Cubes*) Using only the techniques you learned in this chapter, write a program that calculates the squares and cubes of the numbers from 0 to 10 and uses tabs to print the following table of values:

number	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

Making a Difference

2.32 (*Body Mass Index Calculator*) We introduced the body mass index (BMI) calculator in Exercise 1.14. The formulas for calculating BMI are

$$BMI = \frac{weightInPounds \times 703}{heightInInches \times heightInInches}$$

or

$$BMI = \frac{weightInKilograms}{heightInMeters \times heightInMeters}$$

Create a BMI calculator application that reads the user's weight in pounds and height in inches (or, if you prefer, the user's weight in kilograms and height in meters), then calculates and displays the user's body mass index. Also, the application should display the following information from the Department of Health and Human Services/National Institutes of Health so the user can evaluate his/her BMI:

BMI VALUES

Underweight: less than 18.5
Normal: between 18.5 and 24.9
Overweight: between 25 and 29.9
Obese: 30 or greater

[*Note:* In this chapter, you learned to use the `int` type to represent whole numbers. The BMI calculations when done with `int` values will both produce whole-number results. In Chapter 4 you'll learn to use the `double` type to represent numbers with decimal points. When the BMI calculations are performed with `doubles`, they'll both produce numbers with decimal points—these are called “floating-point” numbers.]

2.33 (*Car-Pool Savings Calculator*) Research several car-pooling websites. Create an application that calculates your daily driving cost, so that you can estimate how much money could be saved by car pooling, which also has other advantages such as reducing carbon emissions and reducing traffic congestion. The application should input the following information and display the user's cost per day of driving to work:

- a) Total miles driven per day.
- b) Cost per gallon of gasoline.
- c) Average miles per gallon.
- d) Parking fees per day.
- e) Tolls per day.