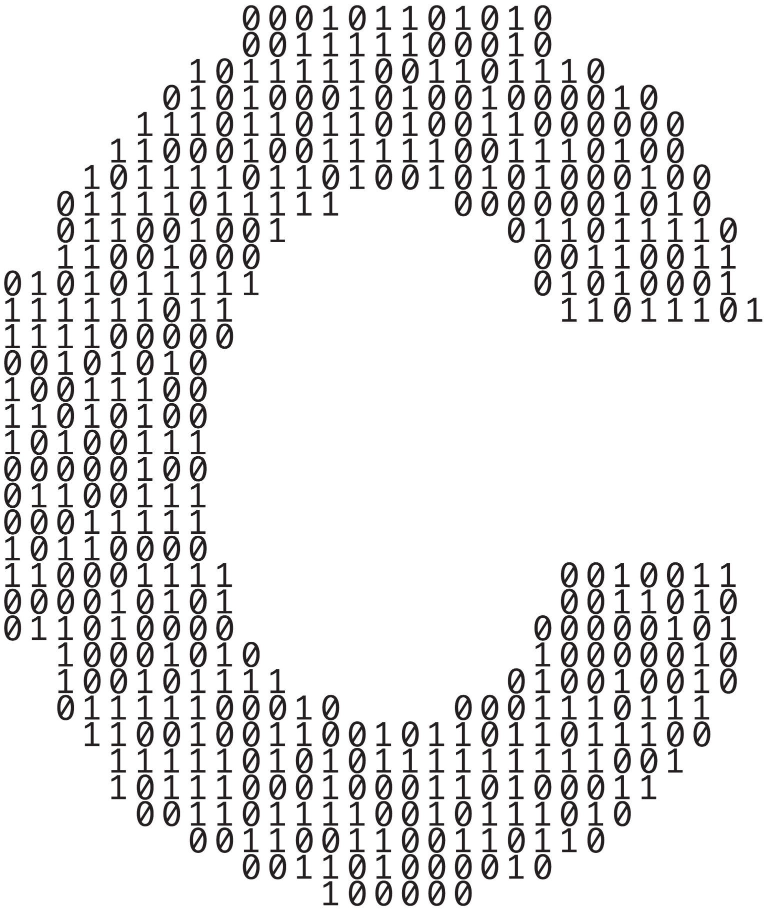
Intro to C Programming

2

Objectives

In this chapter, you'll:

- Write simple C programs.
- 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.
- Begin focusing on secure C programming practices.



Outline

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

Summary | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises*

2.1 Introduction

The C language facilitates a structured and disciplined approach to computer-program design. This chapter introduces C programming and presents several examples illustrating many fundamental C features. We analyze each example one statement at a time. In Chapters 3 and 4, we introduce structured programming—a methodology that will help you produce clear, easy-to-maintain programs. We then use the structured approach throughout the remainder of the text. This chapter concludes with the first of our "Secure C Programming" sections.

2.2 A Simple C Program: Printing a Line of Text

We begin with a simple C program that prints a line of text. The program and its screen output are shown in Fig. 2.1.

```
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     printf("Welcome to C!\n");
8  } // end function main
```

```
Welcome to C!
```

Fig. 2.1 | A first program in C.

Comments

Lines 1 and 2

```
// 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 actions when you execute programs—they're simply ignored. It's our convention in each program to use the line 1 comment to specify the file-name, and the line 2 comment to describe the program's purpose. 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 the shorter // comments because they eliminate common programming errors that occur with /*...*/ comments, such as accidentally omitting the closing */.

⊗ ERR

#include Preprocessor Directive

Line 3

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

is a **C preprocessor directive**. The preprocessor handles lines beginning with # before compilation. Line 3 tells the preprocessor to include the contents of the **standard input/output header** (**<stdio.h>**). This is a file containing information the compiler uses to ensure that you correctly use standard input/output library functions such as printf (line 7). Chapter 5 explains the contents of headers in more detail.

Blank Lines and White Space

We simply left line 4 blank. You use blank lines, space characters and tab characters to make programs easier to read. Together, these are known as **white space** and are generally 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 consist of functions, one of which must be main. Every program begins executing at the function main. As a good practice, precede every function by a comment (as in line 5) stating the function's purpose.

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" in Chapter 4 when we use a math function to perform a calculation and in Chapter 5 when we create custom functions. 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 15, we'll show an example of main receiving information.

A left brace, {, begins each function's **body** (end of line 6). A corresponding **right brace**, }, ends each function's body (line 8). When a program reaches main's closing right brace, the program terminates. The braces and the portion of the program between them form a *block*—an important program unit that we'll discuss more in subsequent chapters.

An Output Statement

Line 7

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

instructs the computer to perform an **action**, namely to display on the screen the **string** of characters enclosed in the quotation marks. A string is sometimes called a **character string**, a **message** or a **literal**.

The entire line 7—including the “call” to the `printf` function to perform its task, the `printf`’s **argument** within the parentheses and the semicolon (`;`)—is called a **statement**. Every statement must end with a semicolon **statement terminator**. The “`f`” in `printf` stands for “formatted.” When line 7 executes, it displays the message `Welcome to C!` on the screen. The characters usually print as they appear between the double quotes, but notice that the characters `\n` were not displayed.

Escape Sequences

In a string, the backslash (\) is an **escape character**. It indicates that `printf` should do something out of the ordinary. In a string, the compiler combines a backslash with the next character to form an **escape sequence**. The escape sequence **`\n`** means **newline**. When `printf` encounters a newline in a string, it positions the output cursor to the beginning of the next line. Some common escape sequences are listed below:

Escape sequence	Description
`\n`	Moves the cursor to the beginning of the next line.
`\t`	Moves the cursor to the next horizontal tab stop.
`\a`	Produces a sound or visible alert without changing the current cursor position.
`\\`	Because the backslash has special meaning in a string, `\\` is required to insert a backslash character in a string.
`\"`	Because strings are enclosed in double quotes, `\"` is required to insert a double-quote character in a string.

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 compiling a `printf` statement, the compiler 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 functions in the object program. Now the object program is complete and ready to execute. The linked program is called an **executable**. If the function name is misspelled, the linker will spot the error—it will not be able to match the name in the program with the name of any known function in the libraries.

ERR ⊗

Indentation Conventions

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

Set a convention for the indent size you prefer and uniformly apply that convention. The tab key may be used to create indents, but tab stops can vary. Professional style guides often recommend using spaces rather than tabs. Some code editors actually insert spaces when you press the *Tab* key.

Using Multiple `printf`s

The `printf` function can display `Welcome to C!` several different ways. For example, Fig. 2.2 uses two statements to produce the same output as Fig. 2.1. This works because each `printf` resumes printing where the previous one finished. Line 7 displays `Welcome` followed by a space (but no newline). Line 8's `printf` begins printing on the same line immediately following the space.

```
1  // fig02_02.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     printf("Welcome ");
8     printf("to C!\n");
9  } // end function main
```

```
Welcome to C!
```

Fig. 2.2 | Printing one line with two `printf` statements.

Displaying Multiple Lines with a Single `printf`

One `printf` can display several lines, as in Fig. 2.3. Each `\n` moves the output cursor to the beginning of the next line.

```
// fig02_03.c
// Printing multiple lines with a single printf.
#include <stdio.h>

// function main begins program execution
int main(void) {
   printf("Welcome\nto\nC!\n");
} // end function main
```

```
Welcome
to
C!
```

Fig. 2.3 | Printing multiple lines with a single `printf`.

Self Check

1 *(Multiple Choice)* Consider the code:

```
int main(void)
```

Which of the following statements is *false*?

a) The parentheses after `main` indicate that it is a function.

b) The keyword `int` to the left of `main` indicates that `main` returns an integer value, and the `void` in parentheses means that `main` does not receive any information.

c) A left parenthesis, `(`, begins every function's body. A corresponding right parenthesis, `)`, ends each function's body.

d) When execution reaches the end of `main`, the program terminates.

Answer: c) is *false*. Actually, a left brace, `{`, begins every function's body, and a corresponding right brace, `}`, ends each function's body.

2 *(Multiple Choice)* Which of the following statements is *false*?

a) Each `printf` resumes printing where the previous one stopped printing.

b) In the following code, the first `printf` displays `Welcome` followed by a space, and the second `printf` begins printing on the next line of output:

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

c) The following `printf` prints several lines of text:

```
printf("Welcome\nto\nC!\n");
```

d) Each time a `\n` escape sequence is encountered, output continues at the beginning of the next line.

Answer: b) is *false*. Actually, the second `printf` begins printing immediately following the space output by the first `printf`.

2.3 Another Simple C Program: Adding Two Integers

Our next program uses the `scanf` standard library function to obtain two integers typed by a user at the keyboard, then computes their sum and displays the result using `printf`. The program and sample output are shown in Fig. 2.4. In the input/output dialog box of Fig. 2.4, we emphasize the numbers entered by the user in **bold**.

```
// fig02_04.c
// Addition program.
#include <stdio.h>

// function main begins program execution
int main(void) {
   int integer1 = 0; // will hold first number user enters
   int integer2 = 0; // will hold second number user enters
```

Fig. 2.4 | Addition program. (Part 1 of 2.)

```
 9
10     printf("Enter first integer: "); // prompt
11     scanf("%d", &integer1); // read an integer
12
13     printf("Enter second integer: "); // prompt
14     scanf("%d", &integer2); // read an integer
15
16     int sum = 0; // variable in which sum will be stored
17     sum = integer1 + integer2; // assign total to sum
18
19     printf("Sum is %d\n", sum); // print sum
20  } // end function main
```

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

Fig. 2.4 | Addition program. (Part 2 of 2.)

The comment in line 2 states the program's purpose. Again, the program begins execution with `main` (lines 6–20)—the braces at lines 6 and 20 mark the beginning and end of `main`'s body, respectively.

Variables and Variable Definitions

Lines 7 and 8

```
int integer1 = 0; // will hold first number user enters
int integer2 = 0; // will hold second number user enters
```

are **definitions**. The names `integer1` and `integer2` are **variables**—locations in memory where the program can store values for later use. These definitions specify that `integer1` and `integer2` have type `int`. This means they'll hold whole-number integer values, such as 7, –11, 0 and 31914. Lines 7 and 8 **initialize** each variable to `0` by following the variable's name with an `=` and a value. Although it's not necessary to explicitly initialize every variable, doing so will help avoid many common problems.

Define Variables Before They Are Used

All variables must be defined with a name and a type before they can be used in a program. You can place each variable definition anywhere in `main` before that variable's first use in the code. In general, you should define variables close to their first use.

Identifiers and Case Sensitivity

A variable name can be any valid **identifier**. Each identifier may consist of letters, digits and underscores (_), but may not begin with a digit. C is **case sensitive**, so `a1` and `A1` are different identifiers. A variable name should start with a lowercase letter. Later in the text, we'll assign special significance to identifiers that begin with a capital letter and identifiers that use all capital letters.

Choosing meaningful variable names helps make a program self-documenting, so fewer comments are needed. Avoid starting identifiers with an underscore (_) to prevent conflicts with compiler-generated identifiers and standard library identifiers. Multiple-word variable names can make programs more readable. For such names:

- separate the words with underscores, as in `total_commissions`, or
- run the words together and begin each subsequent word with a capital letter, as in `totalCommissions`.

The latter style is called **camel casing** because the pattern of uppercase and lowercase letters resembles a camel's silhouette. We prefer camel casing.

Prompting Messages

Line 10

```
printf("Enter first integer: "); // prompt
```

displays `"Enter first integer: "`. This message is called a **prompt** because it tells the user to take a specific action.

The `scanf` Function and Formatted Inputs

Line 11

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

uses **`scanf`** to obtain a value from the user. The function reads from the standard input, which is usually the keyboard.

The "`f`" in `scanf` stands for "formatted." This `scanf` has two arguments—`"%d"` and `&integer1`. The `"%d"` is the **format control string**. It indicates the type of data the user should enter. The **`%d` conversion specification** specifies that the data should be an integer—the `d` stands for "decimal integer". A `%` character begins each conversion specification.

`scanf`'s second argument begins with an ampersand (`&`) followed by the variable name. The `&` is the **address operator** and, when combined with the variable name, tells `scanf` the location (or address) in memory of the variable `integer1`. `scanf` then stores the value the user enters at that memory location.

Using the ampersand (`&`) is often confusing to novice programmers and 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 `&` will become clear after we study pointers in Chapter 7.

ERR ⊗

Forgetting the ampersand (`&`) before a variable in a `scanf` statement typically 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.

When line 11 executes, the computer waits for the user to enter a value for `integer1`. The user types an integer, then presses the ***Enter* key** (or *Return* key) to send the number to the computer. The computer then places the number (or value) in `integer1`. Any subsequent references to `integer1` in the program use this same value.

Functions `printf` and `scanf` facilitate interaction between the user and the computer. This interaction resembles a dialogue and is often called interactive computing.

Prompting for and Inputting the Second Integer

Line 13

```
printf("Enter second integer: "); // prompt
```

prompts the user to enter the second integer, then line 14

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

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

Defining the sum Variable

Line 16

```
int sum = 0; // variable in which sum will be stored
```

defines the `int` variable `sum` and initializes it to `0` before we use `sum` in line 17.

Assignment Statement

The assignment statement in line 17

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

calculates the total of variables `integer1` and `integer2`, then assigns the result to variable `sum` using the assignment operator (=). The statement is read as, "`sum` *gets* the value of the expression `integer1 + integer2`." Most calculations are performed in assignments.

Binary Operators

The = operator and the + operator are binary operators—each has two operands. The + operator's operands are `integer1` and `integer2`. The = operator's operands are `sum` and the value of the expression `integer1 + integer2`. Place spaces on either side of a binary operator to make the operator stand out and make the program more readable.

Printing with a Format Control String

The format control string `"Sum is %d\n"` in line 19

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

contains some literal characters to display (`"Sum is "`) and the conversion specification `%d`, which is a placeholder for an integer. The `sum` is the value to insert in place of `%d`. The conversion specification for an integer (`%d`) is the same in both `printf` and `scanf`—this is true for most, but not all, C data types.

Combining a Variable Definition and Assignment Statement

You can initialize a variable in its definition. For example, lines 16 and 17 can add the variables `integer1` and `integer2`, then initialize the variable `sum` with the result:

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

Calculations in `printf` Statements

Actually, we do not need the variable `sum`, because we can perform the calculation in the `printf` statement. So, lines 16–19 can be replaced with

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

Self Check

1 *(Multiple Choice)* Which statement correctly prompts the user for input?

a) `printf("Enter the day of the week: ")`
b) `printf(Enter the day of the week: );`
c) `printf(’Enter the day of the week: ’);`
d) `printf("Enter the day of the week: ");`

Answer: d.

2 *(Multiple Choice)* The following statement is read as, "`sum` *gets* the value of the expression `integer1 + integer2`." In the statement, `=` is the ________ operator.

```
sum = integer1 + integer2;
```

a) equality.
b) comparison.
c) assignment.
d) None of the above.

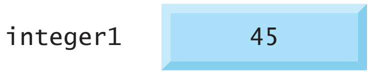
Answer: c.

2.4 Memory Concepts

Every variable has a name, a type, a value and a location in the computer's memory. In Fig. 2.4's addition program, when line 11

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

executes, the program places the user's input into `integer1`'s memory location. Suppose the user enters `45` as `integer1`'s value. Conceptually, memory appears as follows:

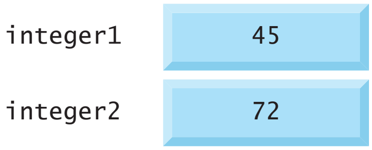


When a value is placed in a memory location, it replaces the location's previous value, which is lost. So, this process is said to be **destructive**.

Returning to our addition program again, when line 14

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

executes, suppose the user enters `72`. Conceptually, memory appears as follows:

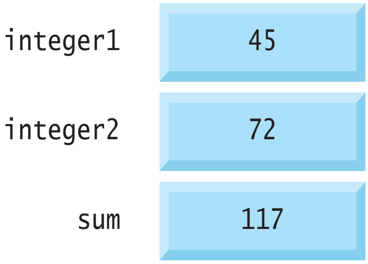


These locations are not necessarily adjacent in memory.

Once we have values for `integer1` and `integer2`, line 18

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

adds these values and places the total into variable `sum`, replacing its previous value. Conceptually, memory now appears as follows:



The `integer1` and `integer2` values are unchanged by the calculation, which uses but does not destroy the values. Thus, reading a value from a memory location is **nondestructive**.

✓ Self Check

1 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?

a) Variable names correspond to locations in the computer's memory.

b) Every variable has a name, a type and a value.

c) When a value is placed in a memory location, it replaces the previous value in that location. The previous value is lost, so this process is said to be destructive. When a value is read from a memory location, the process is said to be nondestructive.

d) All of the above statements are *true*.

Answer: d.

2.5 Arithmetic in C

Most C programs perform calculations using the following binary **arithmetic operators**:

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

Note the use of various special symbols not used in algebra. The **asterisk** (`*`) indicates multiplication, and the **percent sign** (`%`) denotes the remainder operator (introduced below). In algebra, to multiply *a* times *b*, we place these single-letter variable names side-by-side, as in *ab*. In C, `ab` would be interpreted as a single, two-letter name (or identifier). Most programming languages denote multiplication by using the `*` operator, as in `a * b`.

Integer Division and the Remainder Operator

Integer division (that is, dividing one integer by another) yields an integer result, so `7 / 4` evaluates to `1`, and `17 / 5` evaluates to `3`. The integer-only **remainder operator**, **`%`**, yields the remainder after integer division, so `7 % 4` yields `3` and `17 % 5` yields `2`. We'll discuss several interesting applications of the remainder operator.

ERR ⊗

An attempt to divide by zero usually is undefined on computer systems. Generally, it results in a fatal 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 must be written in **straight-line form** to facilitate entering programs into a computer. Expressions like "`a` divided by `b`" must be written as `a/b` with all operators and operands in a straight line. The algebraic notation

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

is generally not acceptable to compilers, although some special-purpose software packages 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. Expressions grouped in parentheses evaluate first. Parentheses are said to be at the "highest level of precedence." In **nested parentheses**, such as

   ```
   ((a + b) + c)
   ```

   the operators in the innermost pair of parentheses are applied first.
2. `*`, `/` and `%` are applied next. If an expression contains several `*`, `/` and `%` operators, evaluation proceeds left-to-right. These three operators are said to be on the same level of precedence.
3. `+` and `-` are evaluated next. If an expression contains `+` and `-` operators, evaluation proceeds left-to-right. These two operators have the same level of precedence, which is lower than that of `*`, `/` and `%`.
4. The assignment operator (`=`) is evaluated last.

The operator precedence rules specify the order C uses to evaluate expressions.[1] When we say evaluation proceeds left-to-right, we're referring to the operator's **grouping**, which is sometimes called **associativity**. Some operators group right-to-left.

Sample Algebraic and C Expressions

Let's consider the evaluation of several expressions. Each example lists an algebraic expression and its C equivalent. The following expression calculates the average (arithmetic mean) of five terms:

Algebra: $m = \dfrac{a + b + c + d + e}{5}$

C:
```
m = (a + b + c + d + e) / 5;
```

In the C statement, 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 we erroneously omit the parentheses, 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:
```
y = m * x + b;
```

No parentheses are required. Multiplication evaluates first because it has higher precedence than addition.

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

Algebra: $z = pr \bmod q + w/x - y$

C:
```
z  =  p  *  r  %  q  +  w  /  x  -  y;
   6     1     2     4     3     5
```

The circled numbers indicate the order in which C evaluates the operators. The multiplication, remainder and division evaluate first left-to-right (that is, they group left-to-right) because they have higher precedence than addition and subtraction. Next, the addition and subtraction evaluate left-to-right. Finally, the result is assigned to `z`.

Parentheses "on the Same Level"

Not all expressions with several pairs of parentheses contain nested parentheses. In the following expression, the parentheses are said to be "on the same level":

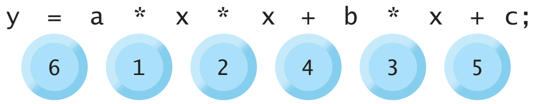
```
a * (b + c) + c * (d + e)
```

In this case, the parenthesized expressions evaluate left-to-right.

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

Evaluation of a Second-Degree Polynomial

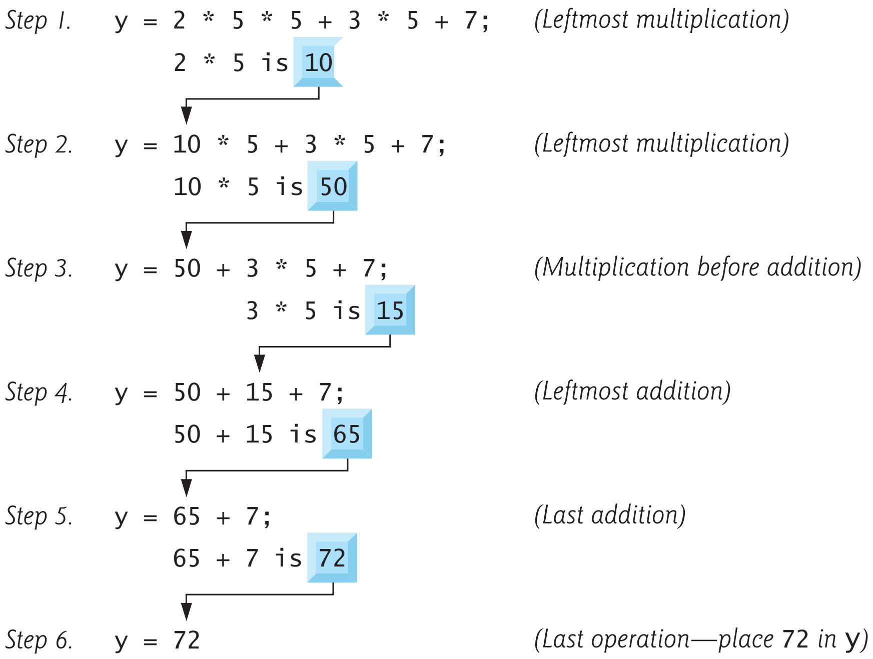
To develop a better understanding of the operator precedence rules, let's take a look at how C evaluates a second-degree polynomial.



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

The circled numbers under the statement indicate the order in which C performs the operations. C does not have an exponentiation operator, so we represent x² as `x * x`. The standard library's `pow` ("power") function performs exponentiation, as you'll see in Chapter 4.

In the preceding second-degree polynomial, suppose `a = 2`, `b = 3`, `c = 7` and `x = 5`. The following diagram illustrates the order in which the operators are applied:



```
Step 1.   y = 2 * 5 * 5 + 3 * 5 + 7;     (Leftmost multiplication)
              2 * 5 is 10

Step 2.   y = 10 * 5 + 3 * 5 + 7;        (Leftmost multiplication)
              10 * 5 is 50

Step 3.   y = 50 + 3 * 5 + 7;            (Multiplication before addition)
                   3 * 5 is 15

Step 4.   y = 50 + 15 + 7;               (Leftmost addition)
              50 + 15 is 65

Step 5.   y = 65 + 7;                    (Last addition)
              65 + 7 is 72

Step 6.   y = 72                         (Last operation—place 72 in y)
```

Using Parentheses for Clarity

As in algebra, it's acceptable to use **redundant parentheses** to make an expression clearer. So, the preceding statement could be parenthesized as follows:

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

Self Check

1 *(Multiple Choice)* Which, if any, of the following expressions properly does the C calculation "add `3` to the quantity `4` times `5`?"

a) `3 + 4 * 5`
b) `3 + (4 * 5)`
c) `(3 + (4 * 5))`
d) All of the above.

Answer: d.

2 *(Multiple Choice)* Consider the statement:

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

Which of the following variations of the preceding statement contain(s) redundant parentheses?

a) `y = (a * x * x + b * x + c);`
b) `y = a * (x * x) + (b * x) + c;`
c) `y = (a * x * x) + (b * x) + c;`
d) All of the above.

Answer: d.

2.6 Decision Making: Equality and Relational Operators

Executable statements either perform actions like calculations, input and output, or, as you're about to see, make decisions. For example, a program might determine whether a person's grade on an exam is greater than or equal to 60, so it can decide whether to print the message "Congratulations! You passed."

A condition is an expression that can be *true* (that is, the condition is met) or *false* (that is, the condition isn't met). This section introduces the `if` statement, which allows a program to make a decision based on a condition's value. If the condition is *true*, the statement in the `if` statement's body executes; otherwise, it does not.

Equality and Relational Operators

Conditions are formed using the following equality and relational operators:

Algebraic equality or relational operator	C equality or relational operator	Sample C condition	Meaning of C condition
Relational operators			
>	`>`	`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
Equality operators			
=	`==`	`x == y`	x is equal to y
≠	`!=`	`x != y`	x is not equal to y

The relational operators <, <=, > and >= have the same precedence and group left-to-right. The equality operators == and != have the same precedence, which is lower than that of the relational operators, and also group left-to-right. In C, a condition may actually be any expression that generates a zero (*false*) or nonzero (*true*) value.

Confusing the Equality Operator == with the Assignment Operator

Confusing == with the assignment operator (=) is a common programming error. To avoid this confusion, read the equality operator as “double equals” and the assignment operator as “gets” or “is assigned the value of.” As you’ll see, confusing these operators can cause difficult-to-find logic errors rather than compilation errors.

Demonstrating the `if` Statement

Figure 2.5 uses six `if` statements to compare two numbers entered by the user. For each `if` statement with a *true* condition, the corresponding `printf` executes. The program and three sample execution outputs are shown in the figure.

```c
// fig02_05.c
// Using if statements, relational
// operators, and equality operators.
#include <stdio.h>

// function main begins program execution
int main(void) {
   printf("Enter two integers, and I will tell you\n");
   printf("the relationships they satisfy: ");

   int number1 = 0; // first number to be read from user
   int number2 = 0; // second number to be read from user

   scanf("%d %d", &number1, &number2); // read two integers

   if (number1 == number2) {
      printf("%d is equal to %d\n", number1, number2);
   } // end if

   if (number1 != number2) {
      printf("%d is not equal to %d\n", number1, number2);
   } // end if

   if (number1 < number2) {
      printf("%d is less than %d\n", number1, number2);
   } // end if

   if (number1 > number2) {
      printf("%d is greater than %d\n", number1, number2);
   } // end if

   if (number1 <= number2) {
      printf("%d is less than or equal to %d\n", number1, number2);
   } // end if

   if (number1 >= number2) {
      printf("%d is greater than or equal to %d\n", number1, number2);
   } // end if
} // end function main
```

Fig. 2.5 | Using `if` statements, relational operators, and equality operators. (Part 1 of 2.)

```
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
```

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

The program uses `scanf` (line 14) to read two integers into the `int` variables `number1` and `number2`. The first `%d` converts a value to be stored in the variable `number1`. The second converts a value to be stored in the variable `number2`.

Comparing Numbers

The `if` statement in lines 16–18

```
if (number1 == number2) {
   printf("%d is equal to %d\n", number1, number2);
} // end if
```

compares `number1`'s and `number2`'s values for equality. If the values are equal, line 17 displays a line of text indicating that the numbers are equal. For each *true* condition in the `if` statements starting in lines 20, 24, 28, 32 and 36, the corresponding body statement displays a line of text. Indenting each `if` statement's body and placing blank lines above and below each `if` statement enhances program readability.

A left brace, `{`, begins the body of each `if` statement (e.g., line 16). A corresponding right brace, `}`, ends each `if` statement's body (e.g., line 18). Any number of statements can be placed in an `if` statement's body.[2]

ERR

Placing a semicolon immediately to the right of the right parenthesis after an `if` statement's condition is a common error. In this case, the semicolon is treated as an empty statement that does not perform a task—the statement that was intended to be part of the `if` statement's body no longer is and always executes.

[2] Using braces to delimit an `if` statement's body is optional for a one-statement body, but it's considered good practice to always use these braces. In Chapter 3, we'll explain the issues.

Operators Introduced So Far

The table below lists from highest-to-lowest precedence the operators introduced so far:

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

The assignment operator (=) groups right-to-left. 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.

Keywords

Some words that we've used in this chapter's examples, such as `int`, `if` and `void`, are **keywords** or reserved words of the language and have special meaning to the compiler. The following table contains the C keywords. Do not use them as identifiers.

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

Keywords added in the C99 standard

`_Bool` `_Complex` `_Imaginary` `inline` `restrict`

Keywords added in the C11 standard

`_Alignas` `_Alignof` `_Atomic` `_Generic` `_Noreturn` `_Static_assert` `_Thread_local`

Self Check

1 *(Multiple Choice)* Which of the following statements is *false*?

a) A condition is an expression that can be *true* or *false*.

b) A condition may be any expression that generates a zero *(true)* or nonzero *(false)* value.

c) The `if` statement makes a decision based on a condition's value. If the condition is *true*, the statement in the `if` statement's body executes; otherwise, it does not.

d) You form conditions using the equality operators and relational operators.

Answer: b) is *false*. Actually, in C, a condition may be any expression that generates a zero (*false*) or nonzero (*true*) value.

2 *(Multiple Choice)* Which of the following statements is *false*?

a) The following `if` statement executes its body if `number1` equals `number2`:

```
if (number1 == number2) {
   printf("%d is equal to %d\n", number1, number2);
} // end if
```

b) If you're uncertain about a complex expression's evaluation order, use parentheses to group expressions or break the statement into simpler statements.

c) Any number of statements can be placed in the body of an `if` statement. Using braces to delimit the body of an `if` statement is required.

d) Some of C's operators, such as the assignment operator (=), group right-to-left rather than left-to-right.

Answer: c) is *false*. Actually, using braces to delimit the body of an `if` statement is optional when the body contains only one statement. Nevertheless, always using these braces helps avoid errors.

2.7 Secure C Programming

SEC

We mentioned *SEI CERT C Coding Standard* in the Preface and indicated that we'd follow certain guidelines to help you avoid programming practices that open systems to attacks.

Avoid Single-Argument `printf`s

SEC

One such guideline is to avoid using `printf` with a single string argument.[3] `printf`'s first argument is a format string, which `printf` inspects for conversion specifications. It then replaces each conversion specification with a subsequent argument's value. It tries to do this regardless of whether there is a subsequent argument to use.

In a later chapter, you'll learn how to input strings from users. Though the first `printf` argument typically is a string literal, it could be a variable containing a string that was input from a user. In such cases, an attacker can craft a user-input format string with more conversion specifications than there are additional `printf` arguments. This exploit has been used by attackers to read memory that they should not be able to access.[4]

[3] For more information, see CERT rule FIO30-C (`https://wiki.sei.cmu.edu/confluence/display/c/FIO30-C.+Exclude+user+input+from+format+strings`). Chapter 6's Secure C Programming section explains the notion of user input as referred to by this CERT guideline.

[4] "Format String Attack," Format String Software Attack | OWASP Foundation. Accessed July 22, 2020, `https://owasp.org/www-community/attacks/Format_string_attack`.

There are a couple of preventative measures you can take to prevent such an attack. If you need to display a string that terminates with a newline, rather than `printf`, use the **`puts` function**, which displays its string argument followed by a newline. For example, in Fig. 2.1, line 7

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

should be written as

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

Function `puts` simply displays its string argument's contents, so a conversion specification would be displayed as its individual characters.

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 specification** is a placeholder for a string. For example, in Fig. 2.2, line 7

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

should be written as

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

As with `puts`, if `printf`'s second argument contains a conversion specification, it will be displayed as its individual characters.

As written, this chapter's `printf`s actually are secure, but these changes are responsible coding practices that will eliminate certain security vulnerabilities as we get deeper into C. We'll explain the rationales later in the book. From this point forward, we use these practices in our examples, and you should use them in your code.

`scanf`, `printf`, `scanf_s` and `printf_s`

SEC

We'll be saying more about `scanf` and `printf` in subsequent Secure C Programming sections, beginning with Section 3.13. We'll also discuss `scanf_s` and `printf_s`, which were introduced in C11 as an attempt to eliminate various `scanf` and `printf` security vulnerabilities. In a later Secure C Programming section, we'll discuss well-known `scanf` security vulnerabilities and how to avoid them.

Self Check

1 *(Code)* Rewrite the following statement as an equivalent secure `puts` statement:

```
printf("Enter your age:\n");
```

Answer: `puts("Enter your age:");`

2 *(Code)* Rewrite the following statement as an equivalent secure `printf` statement:

```
printf("Enter your age:");
```

Answer: `printf("%s", "Enter your age:");`

Summary

This chapter introduced many important C features, 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**.

Section 2.1 Introduction

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

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

- **Comments** (p. 108) begin with `//`. They **document programs** (p. 108) and improve program readability. **Multi-line comments** begin with `/*` and end with `*/` (p. 109).
- Comments are ignored by the compiler.
- The **preprocessor** processes lines beginning with `#` before the program is compiled. The **`#include` directive** tells the preprocessor (p. 109) to include the contents of another file.
- The **`<stdio.h>` header** (p. 109) contains information used by the compiler to ensure that you correctly use standard input/output library functions, such as `printf`.
- The function **`main`** is a part of every program. The parentheses after `main` indicate that `main` is a program building block called a **function** (p. 109). Programs contain one or more functions, one of which must be `main`, which is where the program begins executing.
- 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 every function's **body** (p. 109). A corresponding **right brace**, **`}`**, ends each function (p. 109). A pair of braces and the code between them is called a **block**.
- The **`printf` function** (p. 110) instructs the computer to display information on the screen.
- A **string** is sometimes called a **character string**, a **message** or a **literal** (p. 110).
- Every **statement** (p. 110) must end with the **semicolon statement terminator** (p. 110).
- In `\n` (p. 110), the backslash (`\`) is an **escape character** (p. 110). When encountering a backslash in a string, the compiler combines it with the next character to form an **escape sequence** (p. 110). The escape sequence **`\n`** means **newline**.
- When a newline appears in a string output by `printf`, the output cursor positions to the beginning of the next line.
- The **double backslash (`\\`) escape sequence** places 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** (p. 113) is a location in memory where a value can be stored for use by a program.
- Variables of type **`int`** (p. 113) hold **whole-number integer values**.
- All variables must be defined with a name and a **type** before they can be used in a program.
- A variable name in C is any valid **identifier** (p. 113). An identifier is a series of characters consisting of letters, digits and underscores (_) that does not begin with a digit.
- C is **case sensitive** (p. 113).
- **Function `scanf`** (p. 114) gets input from the standard input—usually the keyboard.
- The **`scanf` format control string** (p. 114) indicates the type(s) of data to input.

- The **%d conversion specification** (p. 114) indicates an integer (the letter d stands for "decimal integer"). The % begins each conversion specification.
- The arguments that follow scanf's format control string begin with an **ampersand (&)** followed by a variable name. In this context, the ampersand—called the **address operator** (p. 114)—tells scanf the variable's memory location. The computer then stores the value at that location.
- Most calculations are performed in **assignment statements** (p. 115).
- The = operator and the + operator are **binary operators**—each has two operands (p. 115).
- In a printf that specifies a format control string as its first argument, the conversion specifications indicate placeholders for data to output.

Section 2.4 Memory Concepts

- Every **variable** has a **name**, a **type**, a **value** and a memory location.
- When a value is placed in a memory location, it replaces the location's previous value, which is lost. So this process is said to be **destructive** (p. 116).
- Reading a value from a memory location is **nondestructive** (p. 117).

Section 2.5 Arithmetic in C

- Most programming languages denote multiplication with the * operator, as in a * b.
- **Arithmetic expressions** must be written in **straight-line form** (p. 118) to facilitate entering programs into the computer.
- **Parentheses** 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** (p. 118), which are generally the same as those followed in algebra.
- Expressions containing several +, / and % operations evaluate left-to-right. These three operators are on the same level of precedence.
- Expressions containing several + and - operations evaluate left-to-right. These two operators have the same level of precedence, which is lower than that of *, / and %.
- Operator **grouping** (p. 119) specifies whether operators evaluate left-to-right or 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** (p. 121) allows a program to make a decision based on whether a condition (p. 121) is ***true*** (p. 121) or ***false*** (p. 121). If the condition is *true*, the if statement's body executes; otherwise, it does not.
- You form conditions in if statements using the **equality** and **relational operators** (p. 121).
- The relational operators all have the same level of precedence and group left-to-right. The equality operators have lower precedence than the relational operators and also group 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."
- The compiler usually ignores **white-space characters** such as tabs, newlines and spaces.
- **Keywords** (p. 124; 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** (p. 126), which displays its string argument followed by a newline character.
- To display a string without a trailing newline character, use `printf` with the `"%s"` conversion specification (p. 126) as the first argument and the string to display as the second argument.

Self-Review Exercises

2.1 Fill-In the blanks in each of the following.

a) Every C program begins execution at the function ________.
b) Every function's body begins with ________ and ends with ________.
c) Every statement ends with a(n) ________.
d) The ________ standard library function displays information on the screen.
e) The escape sequence `\n` represents the ________ character, which causes the cursor to position to the beginning of the next line on the screen.
f) The ________ standard library function obtains data from the keyboard.
g) The conversion specification ________ in a `printf` or `scanf` format control string indicates that an integer will be output or input, respectively.
h) 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 ________.
i) When a value is read from a memory location, the value in that location is preserved; this process is said to be ________.
j) The ________ statement is used to make decisions.

2.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Function `printf` always begins printing at the beginning of a new line.
b) Comments cause the computer to display the text after `//` on the screen when the program is executed.
c) The escape sequence `\n` in a `printf` format control string positions the output cursor to the beginning of the next line.
d) All variables must be defined before they're used.
e) 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 precedence.
k) A program that prints three lines of output must contain three `printf`s.

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

a) Define the variable `number` to be of type `int` and initialize it to `0`.
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 in integer variable a.
d) If `number` is not equal to `7`, display `"number is not equal to 7."`
e) Display `"This is a C program."` on one line.
f) Display `"This is a C program."` on two lines so the first line ends with `C`.
g) Display `"This is a C program."` with each word on a separate line.
h) Display `"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) Prompt the user to enter three integers.
c) Define the variable `x` to be of type `int` and initialize it to `0`.
d) Define the variable `y` to be of type `int` and initialize it to `0`.
e) Define the variable `z` to be of type `int` and initialize it to `0`.
f) Read three integers from the keyboard and store them in variables `x`, `y` and `z`.
g) Define the variable `result`, compute the product of the integers in the variables `x`, `y` and `z`, and use that product to initialize the variable `result`.
h) Display `"The product is"` followed by the value of the `int` 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);{
   puts("C is less than 7");
}
```
d)
```
if (c => 7) {
   puts("C is greater than or equal to 7");
}
```

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 See the answers below:
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 different.
g) *True*.

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 See the answers below:

a) `int number = 0;`

b) `printf("%s", "Enter an integer: ");`

c) `scanf("%d", &a);`

d)
```
if (number != 7) {
   puts("The variable number is not equal to 7.");
}
```

e) `puts("This is a C program.");`

f) `puts("This is a C\nprogram.");`

g) `puts("This\nis\na\nC\nprogram.");`

h) `puts("This\tis\ta\tC\tprogram.");`

2.4 See the answers below:

a) `// Calculate the product of three integers`

b) `printf("%s", "Enter three integers: ");`

c) `int x;`

d) `int y;`

e) `int z;`

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

g) `int result = x * y * z;`

h) `printf("The product is %d\n", result);`

2.5 See below.

```
// Calculate the product of three integers
#include <stdio.h>

int main(void) {
   printf("Enter three integers: "); // prompt

   int x = 0;
   int y = 0;
   int z = 0;
   scanf("%d%d%d", &x, &y, &z); // read three integers

   int result = x * y * z; // multiply values
   printf("The product is %d\n", result); // display result
} // end function main
```

2.6 See the answers below:

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. The `puts` will execute whether or not the `if` statement's condition is true. The semicolon after the right parenthesis is an empty statement that does nothing
Correction: Remove the semicolon after the right parenthesis.

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 sum of %c and %c is %c /n", x, y);`

c) `a + b + c = sum;`

d) `if (number >= largest);`
`    largest == number;`

e) `\\ Program to determine the largest of three integers`

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

g) `printf("Remainder of %d divided by %d is \n", x, y, x / y);`

h) `if (x => y);`
`    printf("%d is greater than or equal to %d\n, x, y");`

i) `print("The product is &d\n," x * y);`

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

2.8 Fill in the blanks in each of the following:

a) All ________ must be declared before being used in a program.

b) C is ________. Uppercase and lowercase letters are different in C.

c) Single-line comments begin with ________.

d) ________ are words reserved by C and cannot be used.

e) ________ and ________ are ignored by the compiler.

2.9 Write a single C statement or line to accomplish each of the following:

a) Display the message "`Have a nice day.`"

b) Assign the sum of variables `b` and `c` to variable `a`.

c) Check if the value of variable `a` is greater than variable `b`. If it is, store the difference of the two in variable `c`.

d) Input three integer values from the keyboard, and place them in `int` variables `p`, `q`, and `r`.

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

a) C regards the functions `main` and `Main` as identical.
b) The associativity of the operators specifies whether they evaluate from left to right or from right to left.
c) The statement `if(a = b)` checks whether the variables `a` and `b` are equal.
d) Conditions in `if` statements are formed by using assignment operators.
e) The following are all valid variable names: `_3g, my_val, h22, 123greetings, July98`.

2.11 Fill in the blanks in each of the following:

a) The ________ statement allows a program to perform different actions based on a condition.
b) If the result of an integer division, where both the numerator and the denominator are integers, is a fraction, the fractional part is ________.
c) The directive ________ tells the preprocessor to include the contents of the input/output stream header files.

2.12 What, if anything, displays when each of the following statements is performed? If nothing displays, then answer "Nothing." Assume `a = 15, b = 4`, and `c = 7`.

a) `printf("%d", a % b);`
b) `printf("%d", a % c + b);`
c) `printf("b=");`
d) `printf("a = 15");`
e) `printf("%d = a + b", a + b);`
f) `c = a + b;`
g) `scanf("%d%d", &a, &b);`
h) `// printf("Now a and b changes to  %d  and %d", a, b);`
i) `printf("\n");`

2.13 Given the following code snippet, what are the output of the `printf`s?

```
int a = 2, b = 4;
a = b * a;
printf("a=%d, b=%d\n", a, b); // What is the output?
scanf("%d", &a); // User enters 5
scanf("%d", &b); // User enters 6
printf("a=%d, b=%d\n", a, b); // What is the output?
```

2.14 Given the equation $y = ax^3 - bx^2 - 6$, which of the following, if any, are correct C statements for this equation?

a) `y = a * x * x * x - b * x * x - 6;`
b) `y = a * x * x * x * b * x * x - 6;`
c) `a * (x * x * x) - b * x * x * (-6);`
d) `a * (x * x * x) - b * (x * x) - 6;`
e) `a * x * x * x - (b * x * x - 6);`
f) `(a * x * 3 - b * x * 2) - 6;`

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 = 8 + 15 * (6 - 2) - 1;`

b) `x = 5 % 5 + 5 * 5 - 5 / 5;`

c) `x = (5 * 7 * (5 + (7 * 5 / (7))));`

2.16 *(Arithmetic)* Write a program that reads two integers from the user then displays their sum, product, difference, quotient and remainder.

2.17 *(Final Velocity)* Write a program that asks the user to enter the initial velocity and acceleration of an object, and the time that has elapsed, places them in the variables `u`, `a`, and `t`, and prints the final velocity, `v`, and distance traversed, `s`, using the following equations.

a) $v = u + at$

b) $s = ut + \frac{1}{2}at^2$

2.18 *(Comparing Values)* Write a program that asks the user to enter the highest rainfall ever recorded in one season for a country, and the rainfall in the current year for that country, obtains the values from the user, checks if the current rainfall exceeds the highest rainfall, and prints an appropriate message on the screen. If the current rainfall is higher, it assigns that value as the highest rainfall ever. 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 displays the sum, the average, the product, and 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 *(Converting from Seconds to Hours, Minutes, and Seconds)* Write a program that asks the user to enter the total time elapsed, in seconds, since an event occurred and converts the time to hours, minutes, and seconds. The time should be displayed as hours:minutes:seconds. [*Hint*: Use the modulus operator].

2.21 What does the following code display?

```
printf("%s", "*\n**\n***\n****\n*****\n");
```

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

2.23 *(Multiples)* Write a program that reads two integers and determines and displays whether the first is a multiple of the second. Use the remainder operator.

2.24 Given the following code snippets, describe the type of the error, and explain what will happen when the program is executed.

a)
```
int a = 2;
int b = a / 0;
```

b)
```
int a = 2;
if (a = 3) { printf("a is 3"); }
else { printf("a is not 3"); }
```

2.25 *(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 and lowercase letters as an integer. You can display the integer equivalent of uppercase `A` by executing the statement

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

A letter is represented by `char` data type in C, and you can perform a `scanf` using `%c`. For example

```
char a; scanf("%c", &a);
```

Write a C program that takes in a letter from user input, and displays both the letter and the integer representation of the letter. For example, if the input is `A`, the output should be `A = 65`.

2.26 *(Integer Value of a Character)* This exercise is the inverse of **Exercise 2.25**. Write a program that takes in an integer from user input and displays the letter or character representation by the given integer. For example, if the input is `65`, the output should be `65 = A`.

2.27 *(Summing the Digits of an Integer)* Write a program that inputs one 4-digit number, sums each of the individual digits, and displays the result. [*Hint*: Use division and remainder operation]. For example, if the input is `3581`, the output should be `17`. (Explanation: `3 + 5 + 8 + 1 = 17`).

2.28 *(Target Heart-Rate Calculator)* While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your doctors and trainers. According to the American Heart Association (AHA) (`http://bit.ly/AHATargetHeartRates`), the formula for calculating your maximum heart rate in beats per minute is 220 minus your age in years. Your target heart rate is 50–85% of your maximum heart rate. Write a program that prompts for and inputs the user's age and calculates and displays the user's maximum heart rate and the range of the user's target heart rate. [**These formulas are estimates provided by the AHA; maximum and target heart rates may vary based on the health, fitness, and gender of the individual. Always consult a physician or qualified healthcare professional before beginning or modifying an exercise program.**]

2.29 *(Sort in Ascending Order)* Write a program that inputs three different numbers from the user. Display the numbers in increasing order. Recall that an `if` statement's body can contain more than one statement. Prove that your script works by running it on all six possible orderings of the numbers. Does your script work with duplicate numbers? [This is challenging. In later chapters you'll do this more conveniently and with many more numbers.]

This page is intentionally left blank