CHAPTER 2

Variables, Constants, Operators, and Writing Programs Using Sequential Statements

After studying this chapter, you will be able to:

- Name variables and use appropriate data types
- Declare and initialize variables
- Understand and use unnamed and named constants
- Use arithmetic operators in expressions
- Use assignment operators in assignment statements
- Write C++ comments
- Write programs using sequential statements and interactive input statements

In this chapter, you learn about writing programs that use variables, constants, and arithmetic operators. You also learn about programs that receive interactive input from a user of your programs. You begin by reviewing variables and constants and learning how to use them in a C++ program. You should do the exercises and labs in this chapter only after you have finished Chapter 2 and Chapter 3 of your book, *Programming Logic and Design, Eighth Edition*, by Joyce Farrell.

Variables

As you know, a **variable** is a named location in the computer's memory whose contents can vary (thus the term *variable*). You use a variable in a program when you need to store values. The values stored in variables often change as a program executes.

In C++, you must declare variables before you can use them in a program. Declaring a variable is a two-part process: first, you give the variable a name, and then you specify its data type. You will learn about data types shortly. But first, you will focus on the rules for naming variables in C++.

Variable Names

Variable names in C++ can consist of letters, numerical digits, and the underscore character, but they cannot begin with a digit. Also, you cannot use a C++ keyword for a variable name. As you learned in Chapter 1 of this book, a keyword is a word with a special meaning in C++. The following are all examples of legal variable names in C++: `my_var`, `num6`, `intValue`, and `firstName`. Table 2-1 lists some examples of invalid variable names and explains why each is invalid.

A variable is sometimes referred to as an identifier.

Name of Variable	Explanation
`3 wrong`	Invalid because it begins with a digit and includes a space
`$don't`	Invalid because it contains a single quotation mark and begins with a dollar sign
`int`	Invalid because it is a C++ keyword
`first name`	Invalid because it contains a space

Table 2-1 Invalid variable names

When naming variables, keep in mind that C++ is case sensitive—in other words, C++ knows the difference between uppercase and lowercase characters. That means `value`, `Value`, and `VaLuE` are three different variable names in C++.

In C++, variable names can be as long as you want. A good rule is to give variables meaningful names that are long enough to describe how the variable is used, but they should not be so long that you make your program hard to read or cause yourself unnecessary typing. For example, a variable named `firstName` will clearly be used to store someone's first name. The variable name `freshmanStudentFirstName` is descriptive but inconveniently long; the variable name `fn` is too short and not meaningful.

One of the naming conventions used by C++ programmers is called **camel case**. This means:

- Variable names are made up of multiple words, with no spaces between them.
- The first character in the variable name is lowercase.
- The first character of each word after the first word is a capitalized character.

Examples of C++ variable names in camel case include `firstName`, `myAge`, and `salePrice`. You do not include spaces between the words in a variable name.

C++ Data Types

In addition to specifying a name for a variable, you also need to specify a particular data type for that variable. A variable's **data type** dictates the amount of memory that is allocated for the variable, the type of data you can store in the variable, and the types of operations that can be performed on the variable. There are many different kinds of data types, but this book focuses on the most basic kind of data types, known as **primitive data types**. There are five primitive data types in C++: `int`, `float`, `double`, `char`, and `bool`. Some of these data types (such as `int`, `double`, and `float`) are used for variables that store numeric values, and they are referred to as numeric data types. The others have specialized purposes. For example, the `bool` data type is used to store a value of either `true` or `false`, and the `char` data type is used to store a single character.

You used the `int` data type in Chapter 1 as the return type for the `main()` function in the Hello World program.

In the programs you write for this book, you will not use all of the primitive data types found in C++. Instead, you will focus on two of the numeric data types (`int` and `double`). The `int` data type is used for values that are whole numbers. For example, you could use a variable with the data type `int` to store someone's age (for example, 25) or the number of students in a class (for example, 35). A variable of the `int` data type consists of 32 bits (4 bytes) of space in memory.

The actual size of the built-in data types may be different on different computers, but the sizes noted in this book indicate the usual sizes on a 32-bit or 64-bit computer.

You use the data type `double` to store a floating-point value (that is, a fractional value), such as the price of an item (2.95) or a measurement (2.5 feet). A variable of the `double` data type occupies 64 bits (8 bytes) of space in memory. You will learn about using other data types as you continue to learn more about C++ in subsequent courses.

In *Programming Logic and Design*, the data type `num` is used to refer to all numeric data types. That book does not make a distinction between `int` and `double` because the pseudocode used in the book is not specific to any one programming language. However, in C++ this distinction is always maintained.

The `int` and `double` data types are adequate for all the numeric variables you will use in this book. But what about when you need to store a group of characters (such as a person's name) in a variable? In programming, you refer to a group of one or more characters as a **string**. An example of a string is the last name *Wallace* or a product type such as a *desk*. There is no primitive data type in C++ for storing strings; instead, they are stored in an object known as a `string` object. In addition to working with the `int` and `double` data types in this book, you will also work with `string`s.

Exercise 2-1: Using C++ Variables, Data Types, and Keywords

In this exercise, you use what you have learned about naming variables, data types, and keywords to answer Questions 1–2.

1. Is each of the following a legal C++ variable name? (Answer yes or no.)

`myIDNumber`	_____	`this_is_a_var`	_____	`AMOUNT`	_____
`yourIDNumber`	_____	`amount`	_____	`$amount`	_____
`int`	_____	`10amount`	_____	`doubleAmount`	_____
`July31`	_____	`one amount`	_____	`Amount`	_____

2. What data type (`int`, `double`, or `string`) is appropriate for storing each of the following values?

   A person's height (in inches) _____

   The amount of interest on a loan, such as 1 percent _____

   The amount of your car payment _____

   Your grandmother's first name _____

   The number of siblings you have _____

Declaring and Initializing Variables

Now that you understand the rules for naming a variable, and you understand the concept of a data type, you are ready to learn how to declare a variable. In C++, you must declare all variables before you can use them in a program. When you **declare** a variable, you tell the compiler that you are going to use the variable. In the process of declaring a variable, you must specify the variable's name and its data type. Declaring a variable tells the compiler that it needs to reserve a memory location for the variable. A line of code that declares a variable is known as a **variable declaration**. The C++ syntax for a variable declaration is as follows:

```
dataType variableName;
```

For example, the declaration statement `int counter;` declares a variable named `counter` of the `int` data type. The compiler reserves the amount of memory space allotted to an `int` variable (32 bits, or 4 bytes) for the variable named `counter`. The compiler then assigns the new variable a specific memory address. In Figure 2-1, the memory address for the variable named `counter` is 1000, although you would not typically know the memory address of the variables included in your C++ programs.

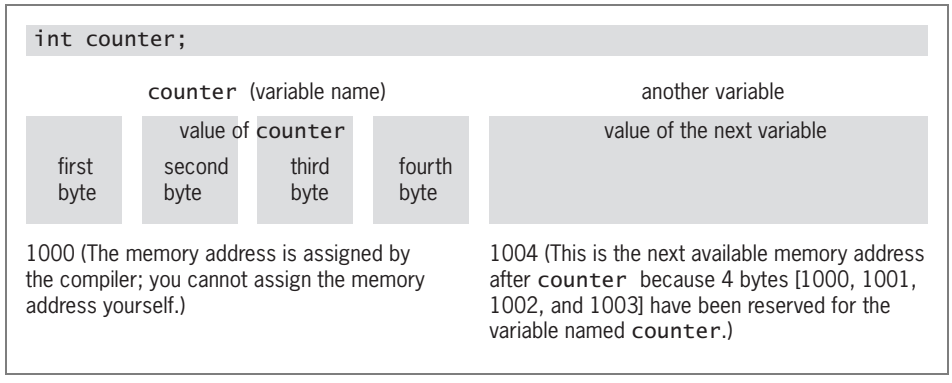

Figure 2-1 Declaration of variable and memory allocation

In C++, variables are not automatically initialized with a value. They contain undetermined values unless you explicitly provide a value.

You can also initialize a C++ variable when you declare it. When you **initialize** a C++ variable, you give it an initial value. For example, you can assign an initial value of 8 to the `counter` variable when you declare it, as shown in the following code:

```
int counter = 8;
```

You can also declare and initialize variables of data type `double` and `string` variables (objects) as shown in the following code:

```
double salary;
double cost = 12.95;
string firstName;
string homeAddress = "123 Main Street";
```

You can declare more than one variable in one statement as long as they have the same data type. For example, the following statement declares two variables, named `counter` and `value`. Both variables are of the `int` data type.

```
int counter, value;
```

Exercise 2-2: Declaring and Initializing C++ Variables

In this exercise, you use what you have learned about declaring and initializing variables to answer Questions 1–2.

1. Write a C++ variable declaration for each of the following. Use `int`, `double`, or `string`, and choose meaningful variable names.

   Declare a variable to store a student ID number (1 to 1000). _____

   Declare a variable to store the number of inches in a foot. _____

   Declare a variable to store a temperature (0.0 to 130.0). _____

   Declare a variable to store the name of the state you live in. _____

2. Declare and initialize variables to represent the following values. Use `int`, `double`, or `string`, and choose meaningful variable names.

   Your hat size (7.5) _____

   The number of days in a leap year _____

   The name of your cat, Mindre _____

   Your cell phone number (999-999-9999) _____

Lab 2-1: Declaring and Initializing C++ Variables

In this lab, you declare and initialize variables in a C++ program provided with the data files for this book. The program, which is saved in a file named `NewAge.cpp`, calculates your age in the year 2050.

1. Open the source code file named `NewAge.cpp` using Notepad or the text editor of your choice.
2. Declare an integer variable named `myNewAge`.
3. Declare and initialize an integer variable named `myCurrentAge`. Initialize this variable with your current age.
4. Declare and initialize an integer variable named `currentYear`. Initialize this variable with the value of the current year. Use four digits for the year.
5. Save this source code file in a directory of your choice, and then make that directory your working directory.
6. Compile the source code file `NewAge.cpp`.
7. Execute the program. Record the output of this program.

   _____

   _____

Constants

As you know, a **constant** is a value that never changes. In C++, you can use both unnamed constants as well as named constants in a program. You will learn about named constants shortly. But first, this section focuses on unnamed constants in C++.

Unnamed Constants

Computers are able to deal with two basic types of data: text and numeric. When you use a specific numeric value, such as 35, in a program, you write it using the numerals, without quotation marks. A specific numeric value is called a **numeric constant** because it does not change; a 35 always has the value 35. When you use a specific text value, or string of characters, such as "William", you enclose the **string constant** in double quotation marks. Both of the preceding examples, 35 and "William", are examples of **unnamed constants** because they do not have specified names as variables do.

Named Constants

In addition to variables, C++ allows you to create named constants. A **named constant** is similar to a variable, except it can be assigned a value only once. You use a named constant when you want to assign a name to a value that will never be changed when a program executes.

To declare a named constant in C++, you use the keyword `const` followed by the data type, followed by the name of the constant. Named constants must be initialized when they are declared, and their contents may not be changed during the execution of the program. For example, the following statement declares an `int` constant named `MAX_STUDENTS` and initializes `MAX_STUDENTS` with the value 35.

```
const int MAX_STUDENTS = 35;
```

By convention, in C++ the names of constants are written in all uppercase letters, and multiple words are separated by the underscore character. This makes it easier for you to spot named constants in a long block of code.

Exercise 2-3: Declaring and Initializing C++ Constants

In this exercise, you use what you have learned about declaring and initializing constants to answer the following question.

1. Declare and initialize constants to represent the following values. Use `int`, `double`, or `string`, and choose meaningful names.

   The price of a pizza is $14.95. _____

   The number of days in September is 30. _____

   The name of your cat, Yogi. _____

   The height of an NBA basketball hoop is 10 feet. _____

Lab 2-2: Declaring and Initializing C++ Constants

In this lab, you declare and initialize constants in a C++ program provided with the data files for this book. The program, which is saved in a file named `NewAge2.cpp`, calculates your age in the year 2050.

1. Open the source code file named `NewAge2.cpp` using Notepad or the text editor of your choice.
2. Declare a constant named `YEAR`, and initialize `YEAR` with the value 2050.
3. Edit the statement `myNewAge = myCurrentAge + (2050 - currentYear)` so it uses the constant named `YEAR`.
4. Edit the statement `cout << "I will be " << myNewAge << " in 2050." << endl;` so it uses the constant named YEAR.
5. Save this source code file as `NewAge2.cpp` in a directory of your choice, and then make that directory your working directory.
6. Compile the source code file `NewAge2.cpp`
7. Execute the program. Record the output of this program.

_____

_____

Arithmetic and Assignment Operators

After you declare a variable, you can use it in various tasks. For example, you can use variables in simple arithmetic calculations, such as adding, subtracting, and multiplying. You can also perform other kinds of operations with variables, such as comparing one variable to another to determine which is greater.

To write C++ code that manipulates variables in this way, you need to be familiar with operators. An **operator** is a symbol that tells the computer to perform a mathematical or logical operation. C++ has a large assortment of operators. The discussion begins with a group of operators known as the arithmetic operators.

Arithmetic Operators

Arithmetic operators are the symbols used to perform arithmetic calculations. You are probably already very familiar with the arithmetic operators for addition (+) and subtraction (–). Table 2-2 lists and explains the arithmetic operators found in C++.

Operator Name and Symbol	Example	Comment
Addition +	`num1 + num2`	
Subtraction –	`num1 - num2`	
Multiplication *	`num1 * num2`	
Division /	`15/2`	Integer division; result is 7; fraction is truncated
	`15.0 / 2.0`	Floating point division; result is 7.5
	`15.0 / 2`	Floating point division because one of the operands is a floating point number; result is 7.5
Modulus %	`hours % 24`	Performs division and finds the remainder; result is 1 if the value of `hours` is 25
Unary plus +	`+num1`	Maintains the value of the expression; if the value of `num1` is 3, then `+num1` is 3
Unary minus –	`-(num1 - num2)`	If value of `(num1 - num2)` is 10, then `-(num1 - num2)` is –10.

Table 2-2 C++ arithmetic operators

You can combine arithmetic operators and variables to create **expressions**. The computer evaluates each expression, and the result is a value. To give you an idea of how this works, assume that the value of `num1` is 3 and `num2` is 20, and that both are of data type `int`. With this information in mind, study the examples of expressions and their values in Table 2-3.

Expression	Value	Explanation
`num1 + num2`	23	3 + 20 = 23
`num1 - num2`	–17	3 – 20 = –17
`num2 % num1`	2	20 / 3 = 6 remainder 2
`num1 * num2`	60	3 * 20 = 60
`num2 / num1`	6	20 / 3 = 6 (remainder is truncated)
`-num1`	–3	Value of `num1` is `3`, therefore `-num1` is `-3`

Table 2-3 Expressions and values

Assignment Operators and the Assignment Statement

Another type of operator is an **assignment operator**. You use an assignment operator to assign a value to a variable. A statement that assigns a value to a variable is known as an **assignment statement**. In C++, there are several types of assignment operators. The one you

will use most often is the = assignment operator, which simply assigns a value to a variable. Table 2-4 lists and explains some of the assignment operators found in C++.

Operator Name and Symbol	Example	Comment
Assignment =	`count = 5;`	Places the value on the right side into the memory location named on the left side
Initialization =	`int count = 5;`	Places the value on the right side into the memory location named on the left side when the variable is declared
Assignment +=	`num += 20;`	Equivalent to `num = num + 20;`
-=	`num -= 20;`	Equivalent to `num = num - 20;`
*=	`num *= 20;`	Equivalent to `num = num * 20;`
/=	`num /= 20;`	Equivalent to `num = num / 20;`
%=	`num %= 20;`	Equivalent to `num = num % 20;`

Table 2-4 C++ assignment operators

When an assignment statement executes, the computer evaluates the expression on the right side of the assignment operator and then assigns the result to the memory location associated with the variable named on the left side of the assignment operator. An example of an assignment statement is shown in the following code. Notice that the statement ends with a semicolon. In C++, assignment statements always end with a semicolon.

```
answer = num1 * num2;
```

This assignment statement causes the computer to evaluate the expression `num1 * num2`. After evaluating the expression, the computer stores the results in the memory location associated with `answer`. If the value stored in the variable named `num1` is 3, and the value stored in the variable named `num2` is 20, then the value 60 is assigned to the variable named `answer`.

Here is another example:

```
answer += num1;
```

This statement is equivalent to the following statement:

```
answer = answer + num1;
```

If the value of `answer` is currently 10 and the value of `num1` is 3, then the expression on the right side of the assignment statement `answer + num1;` evaluates to 13, and the computer assigns the value 13 to `answer`.

Precedence and Associativity

Once you start to write code that includes operators, you need to be aware of the order in which a series of operations is performed. In other words, you need to be aware of the

precedence of operations in your code. Each operator is assigned a certain level of precedence. For example, multiplication has a higher level of precedence than addition. So in the expression `3 * 7 + 2`, the `3 * 7` is multiplied first; only after the multiplication is completed would the 2 be added.

But what happens when two operators have the same precedence? The rules of **associativity** determine the order in which operations are evaluated in an expression containing two or more operators with the same precedence. For example, in the expression `3 + 7 - 2`, the addition and subtraction operators have the same precedence. As shown in Table 2-5, the addition and subtraction operators have left-to-right associativity, which causes the expression to be evaluated from left to right (`3 + 7` added first; then `2` is subtracted). Table 2-5 shows the precedence and associativity of the operators discussed in this chapter.

Operator Name	Operator Symbol(s)	Order of Precedence	Associativity
Parentheses	`()`	First	Left to right
Unary	`- +`	Second	Right to left
Multiplication, division, and modulus	`* / %`	Third	Left to right
Addition and subtraction	`+ -`	Fourth	Left to right
Assignment	`= += -+`	Fifth	Right to left
	`*= /= %=`		

Table 2-5 Order of precedence and associativity

As you can see in Table 2-5, the parentheses operator, `( )`, has the highest precedence. You use this operator to change the order in which operations are performed. Note the following example:

```
average = test1 + test2 / 2;
```

The task of this statement is to find the average of two test scores. The way this statement is currently written, the compiler will divide the value in the `test2` variable by 2, and then add it to the value in the `test1` variable. So, for example, if the value of `test1` is 90 and the value of `test2` is 88, then the value assigned to `average` will be 134, which is obviously not the correct average of these two test scores. By using the parentheses operator in this example, you can force the addition to occur before the division. The correct statement looks like this:

```
average = (test1 + test2) / 2;
```

In this example, the value of `test1`, 90, is added to the value of `test2`, 88, and then the sum is divided by 2. The value assigned to `average`, 89, is the correct result.

Exercise 2-4: Understanding Operator Precedence and Associativity

In this exercise, you use what you have learned about operator precedence and associativity in C++. Study the following code, and then answer Questions 1–2.


```
// This program demonstrates the precedence and
// associativity of operators.
#include <iostream>
using namespace std;
int main()
{
   int number1 = 20;
   int number2 = 5;
   int number3 = 17;
   int answer1, answer2, answer3;
   int answer4, answer5, answer6;
   answer1 = number1 * number2 + number3;
   cout << "Answer 1: " << answer1 << endl;
   answer2 = number1 * (number2 + number3);
   cout << "Answer 2: " << answer2 << endl;
   answer3 = number1 + number2 - number3;
   cout << "Answer 3: "<< answer3 << endl;
   answer4 = number1 + (number2 - number3);
   cout << "Answer 4: " << answer4 << endl;
   answer5 = number1 + number2 * number3;
   cout << "Answer 5: " << answer5 << endl;
   answer6 = number3 / number2;
   cout << "Answer 6: " << answer6 << endl;
   return 0;
}
```

1. What are the values of `answer1`, `answer2`, `answer3`, `answer4`, `answer5`, and `answer6`?

   _____

2. Explain how precedence and associativity affect the result.

   _____

   _____

   _____

   _____

Lab 2-3: Understanding Operator Precedence and Associativity

In this lab, you complete a partially written C++ program that is provided in the data files for this book. The program, which was written for a furniture company, prints the name of the furniture item, its retail price, its wholesale price, the profit made on the piece of furniture, a sale price, and the profit made when the sale price is used.

1. Open the file named `Furniture.cpp` using Notepad or the text editor of your choice.
2. The file includes variable declarations and output statements. Read them carefully before you proceed to the next step.
3. Design the logic and write the C++ code that will use assignment statements to first calculate the profit, then calculate the sale price, and finally calculate the profit when the sale price is used. Profit is defined as the retail price minus the wholesale price. The sale price is 25 percent deducted from the retail price. The sale profit is defined as the sale price minus the wholesale price. Perform the appropriate calculations as part of your assignment statements.
4. Save the source code file in a directory of your choice, and then make that directory your working directory.
5. Compile the program.
6. Execute the program. Your output should be as follows:

```
Item Name: TV Stand
Retail Price: $325
Wholesale Price: $200
Profit: $125
Sale Price: $243.75
Sale Profit: $43.75
```

In Chapter 9 of this book, you will learn how to control the number of places after the decimal point when you want to create output with floating-point values.

Next, you will see how to put together all you have learned in this chapter to write a C++ program that uses sequential statements, comments, and interactive input statements.

Sequential Statements, Comments, and Interactive Input Statements

The term **sequential statements** (or **sequence**) refers to a series of statements that must be performed in sequential order, one after another. You use a sequence in programs when you want to perform actions one after the other. A sequence can contain any number of actions, but those actions must be in the proper order, and no action in the sequence can be skipped. Note that a sequence can contain comments, which are not considered part of the sequence itself.

Comments serve as documentation, explaining the code to the programmer and any other people who might read it. In Chapter 2 of *Programming Logic and Design*, you learned about program comments, which are statements that do not execute. You use comments in C++ programs to explain your logic to people who read your source code. The C++ compiler ignores comments.

You are responsible for including well-written, meaningful comments in all of the programs that you write. In fact, some people think that commenting your source code is as important as the source code itself.

You can choose from two commenting styles in C++. In the first, you type two forward slash characters // at the beginning of each line that you want the compiler to ignore. This style is useful when you only want to mark a single line as a comment. In the second style, you enclose a block of lines with the characters /* and */. This style is useful when you want to mark several lines as a comment. You may place comments anywhere in a C++ program.

The C++ program in the following example shows both styles of comments included in the Temperature program. The first six lines of the program make up a multiline block comment that explains some basic information about the program. Additionally, several single-line comments are included throughout to describe various parts of the program.

A sequence often includes **interactive input statements**, which are statements that ask, or **prompt**, the user to input data. The C++ program in the following example uses sequential statements and interactive input statements to convert a Fahrenheit temperature to its Celsius equivalent:

```
/* Temperature.cpp - This program converts a Fahrenheit
   temperature to Celsius.
   Input: Interactive
   Output: Fahrenheit temperature followed by Celsius
   temperature
*/
#include <iostream>
using namespace std;
int main()
{
   double fahrenheit;
   double celsius;
   // Prompt user
   cout << "Enter Fahrenheit temperature: ";
   // Get interactive user input
   cin >> fahrenheit;
   // Calculate celsius
   celsius = (fahrenheit - 32.0) * (5.0 / 9.0);
   // Output
   cout << "Fahrenheit temperature:" << fahrenheit << endl;
   cout << "Celsius temperature:" << celsius << endl;
   return 0;
}
```

This program is made up of sequential statements that execute one after the other. It also includes comments explaining the code. The comments are the lines enclosed within the /* and */ characters, as well as those lines that begin with //. After the variables `fahrenheit` and `celsius` are declared (using the `double` data type), the following statements execute:

```
// Prompt user
cout << "Enter Fahrenheit temperature: ";
// Get interactive user input
cin >> fahrenheit;
```

The `cout` statement is used to prompt the user for the Fahrenheit temperature so the program can convert it to Celsius. Notice that `endl` is not included in this `cout` statement. This is done to position the cursor on the same line as the displayed prompt.

Remember that `endl` causes the cursor to be displayed on the line following the output.

The next statement, `cin >> fahrenheit;`, is used to retrieve the user's input. This line consists of multiple parts. The first part, `cin`, is the name of an object that represents the user's standard input device, which is usually a keyboard. Next, you see >>, which is called the **extraction** or **input** operator. After >>, you see `fahrenheit`, which is the name of the variable that will store the data that is going to be extracted. The C++ extraction operator >> automatically converts input typed at the keyboard to the appropriate data type. In this example, >> will convert the user's input to the `double` data type because the variable `fahrenheit` is declared as data type `double`.

The next statement to execute is an assignment statement, as follows:

```
celsius = (fahrenheit - 32.0) * (5.0 / 9.0);
```

The formula that converts Fahrenheit temperatures to Celsius is used on the right side of this assignment statement. Notice the use of parentheses in the expression to control precedence. The expression is evaluated, and the resulting value is assigned to the variable named `celsius`.

Notice that the division uses the values 5.0 and 9.0. This is an example of floating-point division, which results in a value that includes a fraction. If the values 5 and 9 were used, integer division would be performed, and the fractional portion would be truncated.

The next two statements to execute in sequence are both output statements, as follows:

```
cout << "Fahrenheit temperature:" << fahrenheit << endl;
cout << "Celsius temperature:" << celsius << endl;
```

The statement `cout << "Fahrenheit temperature:" << fahrenheit << endl;` is used to display the string `Fahrenheit temperature:` followed by the value stored in the variable `fahrenheit`, followed by a newline character. The second output statement displays the words `Celsius temperature:` followed by the value stored in the variable `celsius`, followed by a newline character.

The last statement in this program is `return 0;`. As you learned in Chapter 1 of this book, this statement instructs the compiler to return the value 0 from the `main()` function.

This program is saved in a file named `Temperature.cpp` and is included in the student files for this chapter. You can see the output produced by the Temperature program in Figure 2-2.

Now that you have seen a complete C++ program that uses sequential statements and interactive input statements, it is time for you to begin writing your own programs.

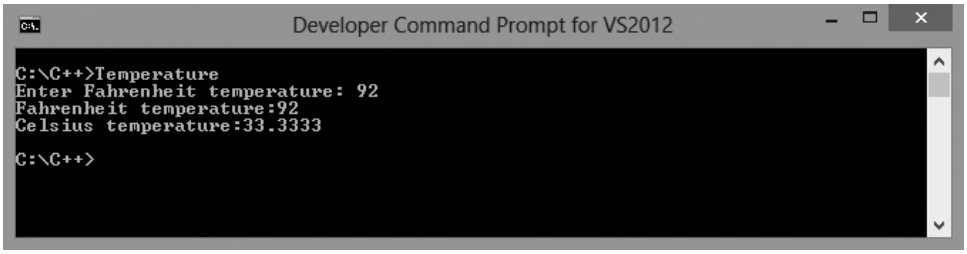

Figure 2-2 Output from `Temperature.cpp` program

Exercise 2-5: Understanding Sequential Statements

In this exercise, you use what you have learned about sequential statements to read a scenario and then answer Questions 1–4.

Suppose you have written a C++ program that calculates the amount of paint you need to cover the walls in your family room. Two walls are 9 feet high and 19.5 feet wide. The other two walls are 9 feet high and 20.0 feet wide. The salesperson at the home improvement store told you to buy 1 gallon of paint for every 150 square feet of wall you need to paint. Suppose you wrote the following code, but your program is not compiling. Take a few minutes to study this code, and then answer Questions 1–4.

```
// This program calculates the number of gallons of paint needed.
#include <iostream>
using namespace std;
int main()
{
   double height1 = 9;
   double height2 = 9;
   int width1 = 19.5;
   double width2 = 20.0;
   double squareFeet;
   int numGallons;
   numGallons = squareFeet / 150;
   squareFeet = (width1 * height1 + width2 * height2) * 2;
   cout << "Number of Gallons: " << numGallons << endl;
   return 0;
}
```

1. When you compile this program, you receive a warning message from the `cl` compiler that is similar to the following:

   ```
   c:\users\jo ann\c++ pal\chapter_2\student\paint.cpp(12) :
   warning C4700: uninitialized local variable 'squareFeet'
   used
   ```

There are multiple problems with this program even though the compiler issued only one warning message, which is pointing out an uninitialized variable on line number 12 in the source code file. You would know the compiler is complaining about line 12 because of the (12) in the warning message. On the following lines, describe how to fix all of the problems you can identify.

2. You have two variables declared in this program to represent the height of your walls, `height1` and `height2`. Do you need both of these variables? If not, how would you change the program? Be sure to identify all of the changes you would make.

Lab 2-4: Understanding Sequential Statements

In this lab, you complete a C++ program provided with the data files for this book. The program calculates the amount of tax withheld from an employee's weekly salary, the tax deduction to which the employee is entitled for each dependent, and the employee's take-home pay. The program output includes state tax withheld, federal tax withheld, dependent tax deductions, salary, and take-home pay.

1. Open the source code file named `Payroll.cpp` using Notepad or the text editor of your choice.
2. Variables have been declared and initialized for you as needed, and the input and output statements have been written. Read the code carefully before you proceed to the next step.
3. Write the C++ code needed to perform the following:
   a. Calculate state withholding tax at 6.5 percent, and calculate federal withholding tax at 28.0.
   b. Calculate dependent deductions at 2.5 percent of the employee's salary for each dependent.
   c. Calculate total withholding. (Total withholding is the total state withholding combined with the total federal withholding.)
   d. Calculate take-home pay as salary minus total withholding plus deductions.
4. Save this source code file in a directory of your choice, and then make that directory your working directory.

5. Compile the program.
6. Execute the program. You should get the following output:

```
State Tax: $81.25
Federal Tax: $350
Dependents: $62.5
Salary: $1250
Take-Home Pay: $881.25
```

In Chapter 9 of this book, you will learn how to control the number of places after the decimal point when you want to output floating-point values.

7. In this program, the variables `salary` and `numDependents` are initialized with the values 1250.00 and 2. To make this program more flexible, modify it to accept interactive input for `salary` and `numDependents`. Name the modified version `Payroll2.cpp`.