C++ Language Companion for

Starting Out with Programming Logic and Design, 5^{th} Edition

By Tony Gaddis

Table of Contents

Introduction

Welcome to the C++ Language Companion for *Starting Out with Programming Logic and Design, 5th Edition*, by Tony Gaddis. You can use this guide as a reference for the C++ Programming Language as you work through the textbook. Each chapter in this guide corresponds to the same numbered chapter in the textbook. As you work through a chapter in the textbook, you can refer to the corresponding chapter in this guide to see how the chapter's topics are implemented in the C++ programming language. In this book you will also find C++ versions of many of the pseudocode programs that are presented in the textbook.

Note: This booklet does not have a chapter corresponding to Chapter 15 of your textbook because C++ does not provide a GUI programming library.

Chapter 1

This chapter accompanies Chapter 1 of Starting Out with Programming Logic and Design, 5th Edition

Introduction to Computers and Programming

A Brief History of C++

The C++ programming language was based on the C programming language. C was created in 1972 by Dennis Ritchie at Bell Laboratories for writing system software. *System software* controls the operation of a computer. For example, an operating system like Windows, Linux, or Mac OS is system software. Because system software must be efficient and fast, the C programming language was designed as a high performance language.

The C++ language was created by Bjarne Stroustrup at Bell Laboratories in the early 1980s, as an extension of the C language. C++ retains the speed and efficiency of C, and adds numerous modern features that make it a good choice for developing large applications. Today, many commercial software applications are written in C++.

The Core Language and Libraries

The C++ language consists of two parts: The core language and the standard library. The *core language* is the set of key words shown in Table 1-1. Each of the key words in the table has a specific meaning and cannot be used for any other purpose. These key words allow a program to perform essential operations, but they do not perform input, output, or other complex procedures. For example, there are no key words in the core language for displaying output on the screen or reading input from the keyboard. To perform these types of operations you use the standard library. The *standard library* is a collection of prewritten code for performing common operations that are beyond those performed by the core language. In addition to input and output, the standard library provides code for performing complex mathematical operations, writing data to files, and other useful tasks.

Writing and Compiling a C++ Program

When a C++ program is written, it must be typed into the computer and saved to a file. A *text editor*, which is similar to a word processing program, is used for this task. The C++ programming statements written by the programmer are called *source code*, and the file they are saved in is called a *source file*.

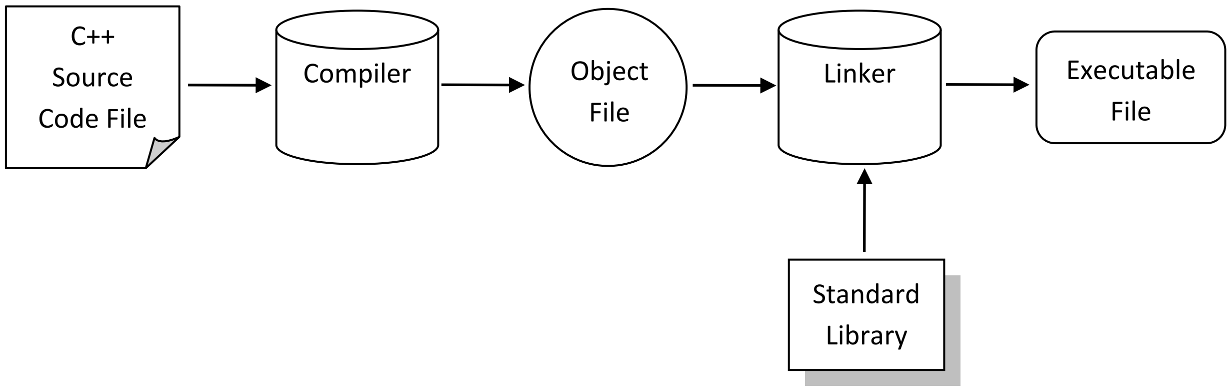
After the programmer saves the source code to a file, he or she runs the C++ compiler. A *compiler* is a program that translates source code into an executable form. During the translation process, the compiler uncovers any syntax errors that may be in the program. *Syntax errors* are mistakes that the programmer has made that violate the rules of the C++ programming language. These errors must be corrected before the compiler can successfully translate the source code. If the program is free of syntax errors, the compiler stores the translated machine language instructions, which are called *object code*, in an *object file*.

Although an object file contains machine language instructions, it is not a complete program because it does not contain any code that the program needs from the standard library. Another program, known as the linker, combines the object file with the necessary library routines. Once the linker has finished with this step, an *executable file* is created. The executable file contains machine language instructions, or *executable code*, and is ready to run on the computer. Figure 1-1 illustrates the process of compiling and linking a C++ program.

Table 1-1: Key words in the C++ language

alignas	alignof	and	and_eq	asm
auto	bitand	bitor	bool	break
case	catch	char	char16_t	char32_t
class	compl	concept	const	constexpr
const_cast	continue	decltype	default	delete
do	double	dynamic_cast	else	enum
explicit	export	extern	false	float
for	friend	goto	if	inline
int	long	mutable	namespace	new
noexcept	not	not_eq	nullptr	operator
or	or_eq	private	protected	public
register	reinterpret_cast	requires	return	short
signed	sizeof	static	static_assert	static_cast
struct	switch	template	this	thread_local
throw	true	try	typedef	typeid
typename	union	unsigned	using	virtual
void	volatile	wchar_t	while	xor
xor_eq				

Figure 1-1 Compiling and Linking a C++ Program



The Parts of a C++ Program

As you work through this book, all of the C++ programs that you will write will contain the following code:

```
#include <iostream>
using namespace std;

int main()
{

    return 0;
}
```

You can think of this as a skeleton program. As it is, it does absolutely nothing. But you can add additional code to this program to make it perform an operation. Let’s take a closer look at the parts of the skeleton program.

> **Note:** At this point, it is not critical that you understand everything about the skeleton program. The description that follows is meant to demystify the code, at least a little. Because you are a beginning programmer, you should expect that some of the following concepts will be unclear. As you dig deeper into the C++ language, you will understand these concepts. So, don't despair! Your journey is just beginning.

The first line reads:

```
#include <iostream>
```

This is called an *include directive*. It causes the contents of a file named `iostream` to be included in the program. The `iostream` file contains prewritten code that allows a C++ program to display output on the screen and read input from the keyboard. The next line reads:

```
using namespace std;
```

A program usually contains several items that have names. C++ uses *namespaces* to organize the names of program entities. The statement `using namespace std;` declares that the program will be accessing entities whose names are part of a namespace called `std`. The reason the program needs access to the `std` namespace is because every name created by the `iostream` file is part of that namespace. In order for a program to use the entities in `iostream`, it must have access to the `std` namespace. (Notice that the statement ends with a semicolon. More about that in a moment.)

The following code appears next:

```
int main()
{

    return 0;
}
```

This is called a *function*. You will learn a great deal about functions later, but for now, you simply need to know that a function is a group of programming statements that collectively has a name. The name of this function is `main`. Every C++ program must have a function named `main`, which serves as the program's starting point.

Notice that a set of curly braces appears below the line that reads `int main()`. The purpose of these braces is to enclose the statements that are in the `main` function. In this particular program, the `main` function contains only one statement, which is:

```
return 0;
```

This statement returns the number 0 to the operating system when the program finishes executing. When you write your first C++ programs, you will write other statements inside the main function's curly braces, as indicated in Figure 1-2.

> **NOTE:** C++ is a case-sensitive language. That means it regards uppercase letters as being entirely different characters than their lowercase counterparts. In C++, the name of the `main` function must be written in all lowercase letters. C++ doesn't see `Main` the same as `main`, or `INT` the same as `int`. This is true for all the C++ key words.

Figure 1-2 The skeleton program

```
#include <iostream>
using namespace std;

int main()
{

    return 0;
}
```

You will write other C++ statements in this area.

Semicolons

In C++, a complete programming *statement* ends with a semicolon. You'll notice that some lines of code in the skeleton program do not end with a semicolon, however. For example, the `include` directive does not end with a semicolon because, technically speaking, directives are not statements. The `main` function header does not end with a semicolon because it marks the beginning of a function. Also, the curly braces are not followed by a semicolon because they are not statements because they form a container that holds statements. If this is confusing, don't despair! As you practice writing C++ programs more and more, you will develop an intuitive understanding of the difference between statements and lines of code that are not considered statements.

Chapter 2

This chapter accompanies Chapter 2 of Starting Out with Programming Logic and Design, 5th Edition

Input, Processing, and Output

Displaying Screen Output

To display output on the screen in C++ you write a `cout` statement (pronounced *see out*). A `cout` statement begins with the word `cout`, followed by the << operator, followed by an item of data that is to be displayed. The statement ends with a semicolon. Program 2-1 demonstrates.

Program 2-1

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6      cout << "Hello world";
7      return 0;
8   }
```

Program Output

```
Hello world
```

Remember, these line numbers are **NOT** part of the program! Don't type the line numbers when you are entering program code. All of the programs in this booklet will show line numbers for reference purposes only.

The << operator is known as the *stream insertion operator*. It always appears on the left side of the item of data that you want to display. Notice that in line 6, the << operator appears to the left of the string `"Hello world"`. When the program runs, *Hello world* is displayed.

You can display multiple items with a single `cout` statement, as long as a << operator appears to the left of each item. Program 2-2 shows an example. In line 6, three items of data are being displayed: the string `"Programming "`, the string `"is "`, and the string `"fun."`. Notice that the << operator appears to the left of each item.

When you display output with `cout`, the output is displayed as one continuous line on the screen. for example, look at Program 2-3. Even though the program has three `cout` statements, its output appears on one line.

Program 2-2

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Programming " << "is " << "fun.";
    return 0;
}
```

Program Output

```
Programming is fun.
```

Program 2-3

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Programming ";
    cout << "is ";
    cout << "fun.";
    return 0;
}
```

Program Output

```
Programming is fun.
```

The output comes out as one long line is because the `cout` statement does not start a new line unless told to do so. You can use the `endl` manipulator to instruct `cout` to start a new line. Program 2-4 shows an example. (Program 2-4 is the C++ equivalent of Program 2-1 in your textbook.)

Program 2-4

This program is the C++ version of **Program 2-1** in your textbook.

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Kate Austen" << endl;
    cout << "1234 Walnut Street" << endl;
    cout << "Asheville, NC 28899" << endl;
    return 0;
}
```

Program Output

```
Kate Austen
1234 Walnut Street
Asheville, NC 28899
```

Variables

In C++, variables must be declared before they can be used in a program. A variable declaration statement is written in the following general format:

```
DataType VariableName;
```

In the general format, *`DataType`* is the name of a C++ data type, and *`VariableName`* is the name of the variable that you are declaring. The declaration statement ends with a semicolon. For example, the key word `int` is the name of the integer data type in C++, so the following statement declares a variable named `number`.

```
int number;
```

Table 2-1 lists some of the C++ data types, gives their memory size in bytes, and describes the type of data that each can hold. Note that in this book we will primarily use the `int`, `double`, and `string` data types.[1]

Table 2-1 C++ Data Types

Data Type	Size	What It Can Hold
`short`	2 bytes	Integers in the range of –32,768 to +32,767
`int`	4 bytes	Integers in the range of –2,147,483,648 to +2,147,483,647
`long`	4 bytes	Integers in the range of –2,147,483,648 to +2,147,483,647
`float`	4 bytes	Floating-point numbers in the range of $\pm 3.4\times10^{-38}$ to $\pm 3.4\times10^{38}$, with 7 digits of accuracy
`double`	8 bytes	Floating-point numbers in the range of $\pm 1.7\times10^{-308}$ to $\pm 1.7\times10^{308}$, with 15 digits of accuracy
`char`	1 byte	Can store integers in the range of –128 to +127. Typically used to store characters.
`string`	Varies	Strings of text.
`bool`	1 byte	Stores the values `true` or `false`

Here are some other examples of variable declarations:

```
int speed;
double distance;
```

[1] To use the `string` data type, you must write the directive `#include <string>` at the top of your program. To be correct, `string` is not a data type in C++, it is a class. We use it as a data type, though.

```
String name;
```

Several variables of the same data type can be declared with the same declaration statement. For example, the following statement declares three `int` variables named `width`, `height`, and `length`.

```
int width, height, length;
```

You can also initialize variables with starting values when you declare them. The following statement declares an `int` variable named `hours`, initialized with the starting value 40:

```
int hours = 40;
```

Variable Names

You may choose your own variable names in C++, as long as you do not use any of the C++ key words (previously shown in Table 1-1). The key words make up the core of the language and each has a specific purpose. The following are some additional rules that must be followed when naming variables:

- The first character must be one of the letters a–z, A–Z, or an underscore (_).
- After the first character, you may use the letters a–z or A–Z, the digits 0–9, underscores (_).
- Uppercase and lowercase characters are distinct. This means `itemsOrdered` is not the same as `itemsordered`.
- Variable names cannot include spaces.

Program 2-5 shows an example with three variable declarations. Notice that, because we are using a `string` variable, we have the `#include <string>` directive in line 2. Line 7 declares a `string` variable named `name`, initialized with the string "Jeremy Smith". Line 8 declares an `int` variable named `hours` initialized with the value 40. Line 9 declares a `double` variable named `pay`, initialized with the value 852.99. Notice that in lines 11 through 13 we use `cout` to display the contents of each variable.

Program 2-5

```
 1 #include <iostream>
 2 #include <string>
 3 using namespace std;
 4
 5 int main()
 6 {
 7    string name = "Jeremy Smith";
 8    int hours = 40;
 9    double pay = 852.99;
10
11    cout << name << endl;
12    cout << hours << endl;
13    cout << pay << endl;
14    return 0;
```

```
}
```

Program Output

```
Jeremy Smith
40
852.99
```

Reading Keyboard Input

To read keyboard input in C++ you write a `cin` statement (pronounced *see in*). A `cin` statement begins with the word `cin`, followed by the >> operator, followed by the name of a variable. The statement ends with a semicolon. When the statement executes, the program will wait for the user to enter input at the keyboard, and press the Enter key. When the user presses Enter, the input will be assigned to the variable that is listed after the >> operator. (The >> operator is known as the *stream extraction operator*.) Program 2-6 demonstrates.

Program 2-6

This program is the C++ version of **Program 2-2** in your textbook.

```
#include <iostream>
using namespace std;

int main()
{
   int age;

   cout << "What is your age?" << endl;
   cin >> age;
   cout << "Here is the value that you entered:" << endl;
   cout << age;
   return 0;
}
```

Program Output

```
What is your age?
```
24 [*Enter*]
```
Here is the value that you entered:
24
```

The program shown in Program 2-7 uses `cin` statements to read a string, an integer, and a `double`.

Program 2-7

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string name;
   double payRate;
   int hours;
```

```


    cout << "Enter your first name." << endl;
    cin >> name;
    cout << "Enter your hourly pay rate." << endl;
    cin >> payRate;
    cout << "Enter the number of hours worked." << endl;
    cin >> hours;

    cout << "Here are the values that you entered:" << endl;
    cout << name << endl;
    cout << payRate << endl;
    cout << hours << endl;
    return 0;
}
```

Program Output

```
Enter your first name.
Connie [Enter]
Enter your hourly pay rate.
55.25 [Enter]
Enter the number of hours worked.
40 [Enter]
Here are the values that you entered:
Connie
55.25
40
```

Program 2-8 shows an example that reads input from the user, and then uses that input in messages that are displayed on the screen. Notice that line 15 displays the string "Hello ", followed by the value of the `name` variable. Line 16 displays the string "You are ", followed by the value of the `age` variable, followed by the string " years old."

Program 2-8

> This program is the C++ version of **Program 2-4** in your textbook.

```
 1 #include <iostream>
 2 #include <string>
 3 using namespace std;
 4
 5 int main()
 6 {
 7    int age;
 8    string name;
 9
10    cout << "Enter your name." << endl;
11    cin >> name;
12    cout << "Enter your age." << endl;
```

```
    cin >> age;

    cout << "Hello " << name << endl;
    cout << "You are " << age << " years old." << endl;
    return 0;
}
```

Program Output

```
Enter your name.
```
Andrea [*Enter*]
```
Enter your age.
```
24 [*Enter*]
```
Hello Andrea
You are 24 years old.
```

Reading String Input Containing Spaces

A `cin` statement can read one-word string input, as previously shown in Program 2-8, but it does not behave as you would expect when the user's input is a string containing multiple words, separated by spaces. If you want to read a string that contains multiple words, you must use the `getline` function. The `getline` function reads an entire line of input, including embedded spaces, and stores it in a `string` variable. The `getline` function looks like the following, where `inputLine` is the name of the `string` variable receiving the input.

```
getline(cin, inputLine);
```

Program 2-9 shows an example of how the `getline` function is used.

Program 2-9

```
// This program demonstrates using the getline function
// to read input into a string variable.
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string name;
    string city;

    cout << "Please enter your name." << endl;
    getline(cin, name);

    cout << "Enter the city you live in." << endl;
    getline(cin, city);

    cout << "Hello, " << name << endl;
    cout << "You live in " << city << endl;
```

```
    return 0;
}
```

Program Output

```
Please enter your name.
Kate Smith [Enter]
Enter the city you live in.
West Jefferson [Enter]
Hello, Kate Smith
You live in West Jefferson
```

Performing Calculations

Table 2-3 shows the C++ arithmetic operators, which are nearly the same as those presented in Table 2-1 in your textbook.

Table 2-3 C++'s Arithmetic Operators

Symbol	Operation	Description
+	Addition	Adds two numbers
-	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies two numbers
/	Division	Divides one number by another and gives the quotient
%	Modulus	Divides one integer by another and gives the remainder

Here are some examples of statements that use an arithmetic operator to calculate a value, and assign that value to a variable:

```
total = price + tax;
sale = price - discount;
population = population * 2;
half = number / 2;
leftOver = 17 % 3;
```

Program 2-10 shows an example program that performs mathematical calculations (This program is the C++ version of pseudocode Program 2-9 in your textbook.)

Perhaps you noticed that Table 2-3 does not show an exponent operator. C++ does not provide such an operator, but it does provide a function named `pow` for this purpose. Here is an example of how the `pow` function is used:

```
result = pow(4.0, 2.0);
```

The function takes two double arguments (the numbers shown inside the parentheses). It raises the first argument to the power of the second argument, and returns the result as a double. In this example, 4.0 is raised to the power of 2.0. This statement is equivalent to the following algebraic statement:

result = 4^2

Program 2-10

```
#include <iostream>
using namespace std;

int main()
{
   double originalPrice, discount, salePrice;

   cout << "Enter the item's original price." << endl;
   cin >> originalPrice;
   discount = originalPrice * 0.2;
   salePrice = originalPrice - discount;

   cout << "The sale price is $" << salePrice << endl;
   return 0;
}
```

This program is the C++ version of **Program 2-9** in your textbook.

Program Output

```
Enter the item's original price: 100 [Enter]
The sale price is $80
```

Named Constants

You create named constants in C++ by using the const key word in a variable declaration. The word const is written just before the data type. Here is an example:

```
const double INTEREST_RATE = 0.069;
```

This statement looks just like a regular variable declaration except that the word const appears before the data type, and the variable name is written in all uppercase letters. It is not required that the variable name appear in all uppercase letters, but many programmers prefer to write them this way so they are easily distinguishable from regular variable names.

An initialization value must be given when declaring a variable with the const modifier, or an error will result when the program is compiled. A compiler error will also result if there are any statements in the program that attempt to change the value of a const variable.

Documenting a Program with Comments

To write a line comment in C++ you simply place two forward slashes (`//`) where you want the comment to begin. The compiler ignores everything from that point to the end of the line. Here is an example:

```
// This program calculates an employee's gross pay.
```

Multi-line comments, or block comments, start with `/*` (a forward slash followed by an asterisk) and end with `*/` (an asterisk followed by a forward slash). Everything between these markers is ignored. Here is an example:

```
/*
     This program calculates an employee's gross pay.
     Written by Matt Hoyle.
*/
```

Chapter 3

This chapter accompanies Chapter 3 of Starting Out with Programming Logic and Design, 5th Edition

Modules

Chapter 3 in your textbook discusses modules as named groups of statements that perform specific tasks in a program. You use modules to break a program down into small, manageable units. In C++, we use *functions* for this purpose. In this chapter we will discuss how to define and call C++ functions, use local variables in a function, and pass arguments to a function. We also discuss global variables, and the use of global constants.

Defining and Calling a Function

To create a function you must write its *definition*, which consists of two general parts: a header and a body. The *function header* is the line that appears at the beginning of a function definition. It lists several things about the function, including the function's name. The *function body* is a collection of statements that are performed when the function is executed. These statements are enclosed inside a set of curly braces.

As you already know, every complete C++ program must have a `main` function. C++ programs can have other functions as well. Here is an example of a simple function that displays a message on the screen:

```
void showMessage()
{
    cout << "Hello world" << endl;
}
```

For now, the headers for all of the C++ functions that you will write will begin with the key word `void`. Following this you write the name of the function, and a set of parentheses. Remember that a function header never ends with a semicolon!

Calling a Function

A function executes when it is called. The `main` function is automatically called when a program starts, but other functions are executed by function call statements. When a function is called, the program branches to that function and executes the statements in its body. Here is an example of a function call statement that calls the `showMessage` function we previously examined:

```
showMessage();
```

The statement is simply the name of the function followed by a set of parentheses. Because it is a complete statement, it is terminated with a semicolon.

Program 3-1 shows a C++ program that demonstrates the `showMessage` function. This is the C++ version of pseudocode Program 3-1 in your textbook.

Program 3-1

This program is the C++ version of **Program 3-1** in your textbook.

```
1 #include <iostream>
2 using namespace std;
3
4 void showMessage();
5
6 int main()
7 {
8    cout << "I have a message for you." << endl;
9    showMessage();
10   cout << "That's all, folks!" << endl;
11   return 0;
12 }
13
14 void showMessage()
15 {
16   cout << "Hello world" << endl;
17 }
```

Program Output

```
I have a message for you.
Hello world
That's all, folks!
```

The program has two functions: `main` and `showMessage`. The `main` function appears in lines 6 through 12, and the `showMessage` function appears in lines 14 through 17. When the program runs, the `main` function executes. The statement in line 8 displays "I have a message for you." Then the statement in line 9 calls the `showMessage` function. This causes the program to branch to the `showMessage` function and execute the statement that appears in line 16. This displays "Hello world". The program then branches back to the `main` function and resumes execution at line 10. This displays "That's all, folks!"

Notice the statement that appears in line 4:

```
void showMessage();
```

This line of code is a function prototype. A *function prototype* is a statement that declares the existence of a function, but does not define the function. It is merely a way of telling the compiler that a particular function exists in the program, and its definition appears at a later point. Without this statement, an error would occur when the program is compiled.

Local Variables

Variables that are declared inside a function are known as local variables. They are called *local* because they are local to the function in which they are declared. Statements outside a function cannot access that function's local variables.

Because a function's local variables are hidden from other functions, the other functions may have their own local variables with the same name. For example, look at Program 3-2. In addition to the `main` function, this program has two other functions: `texas` and `california`. These two functions each have a local variable named `birds`.

Program 3-2

```
#include <iostream>
using namespace std;

// Function prototypes
void texas();
void california();

// Definition of the main function
int main()
{
   // Call the texas function.
   texas();

   // Call the california function.
   california();
   return 0;
}

// Definition of the texas function
void texas()
{
   // Local variable named birds
   int birds = 1000;

   // Display the value of the birds variable.
   cout << "The texas function has " << birds
        << " birds." << endl;
}

// Definition of the california function
void california()
{
   // Local variable named birds
   int birds = 200;

   // Display the value of the birds variable.
   cout << "The california function has " << birds
        << " birds." << endl;
}
```

Program Output

```
The texas function has 1000 birds.
The california function has 200 birds.
```

Although there are two variables named `birds`, the program can only see one of them at a time because they are in different functions. When the `texas` function is executing, the `birds` variable declared inside `texas` is visible. When the `california` function is executing, the `birds` variable declared inside `california` is visible.

It's worth noting that although different functions can have a local variable with the same name, you cannot declare two local variables with the same name in the same function.

Passing Arguments to Functions

If you want to be able to pass an argument into a function, you must declare a parameter variable in that function's header. The parameter variable will receive the argument that is passed when the function is called. Here is the definition of a function that uses a parameter:

```
void displayValue(int num)
{
    cout << "The value is " << num << endl;
}
```

Notice the integer variable declaration that appears inside the parentheses (`int num`). This is the declaration of a parameter variable, which enables the `displayValue` function to accept an integer value as an argument. Here is an example of a call to the `displayValue` function, passing 5 as an argument:

```
displayValue(5);
```

This statement executes the `displayValue` function. The argument that is listed inside the parentheses is copied into the function's parameter variable, `num`.

Program 3-3 shows a complete program with a function that accepts an argument. This is the C++ version of pseudocode Program 3-5 in your textbook. When the program runs, line 9 calls the `doubleNumber` function, passing the value 4 as an argument.

The `doubleNumber` function is defined in lines 14 through 24. The function has an `int` parameter variable named `value`. A local `int` variable named `result` is declared in line 17, and in line 20 the `value` parameter is multiplied by 2 and the result is assigned to the `result` variable. In line 23 the value of the `result` variable is displayed.

Program 3-3

```
1 #include <iostream>
2 using namespace std;
3
4 // Function prototype
5 void doubleNumber(int);
```

This program is the C++ version of **Program 3-5** in your textbook.

```
 6
 7 int main()
 8 {
 9    doubleNumber(4);
10    return 0;
11 }
12
13 // Definition of the doubleNumber function
14 void doubleNumber(int value)
15 {
16    // Local variable to hold the result
17    int result;
18
19    // Multiply the value parameter times 2.
20    result = value * 2;
21
22    // Display the result.
23    cout << result << endl;
24 }
```

Program Output

```
8
```

Notice the function prototype for the `doubleNumber` function in line 5:

```
void doubleNumber(int);
```

Inside the parentheses, the data type of the parameter variable is listed. It is not necessary to list the name of the parameter variable inside the parentheses. Only its data type is required. (You will not cause an error if you write the names of parameters in a function prototype. Since they are not required, the compiler merely ignores them.)

Program 3-4 shows another program that uses the `doubleNumber` function. This is the C++ version of pseudocode Program 3-6 in your textbook. When the program runs, it prompts the user to enter a number. Line 17 reads an integer from the keyboard and assigns it to the `number` variable. Line 21 calls the `doubleNumber` function, passing the `number` variable as an argument.

The `doubleNumber` function is defined in lines 26 through 36. The function has an `int` parameter variable named `value`. A local `int` variable named `result` is declared in line 29, and in line 32 the `value` parameter is multiplied by 2 and the result is assigned to the `result` variable. In line 35 the value of the `result` variable is displayed.

Program 3-4

This program is the C++ version of **Program 3-6** in your textbook.

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 // Function prototype
 5 void doubleNumber(int);
```

```cpp

int main()
{
    // Declare a variable to hold a number.
    int number;

    // Prompt the user for a number
    cout << "Enter a number and I will display" << endl;
    cout << "that number doubled." << endl;

    // Read an integer from the keyboard.
    cin >> number;

    // Call the doubleNumber function passing
    // number as an argument.
    doubleNumber(number);
    return 0;
}

// Definition of the doubleNumber function
void doubleNumber(int value)
{
    // Local variable to hold the result
    int result;

    // Multiply the value parameter times 2.
    result = value * 2;

    // Display the result.
    cout << result << endl;
}
```

Program Output

```
Enter a number and I will display
that number doubled.
22 [Enter]
44
```

Passing Multiple Arguments

Often it is useful to pass more than one argument to a function. When you define a function, you must declare a parameter variable for each argument that you want passed into the function. Program 3-5 shows an example. This is the C++ version of pseudocode Program 3-7 in your textbook.

Program 3-5

```
#include <iostream>
using namespace std;

// Function prototype
void showSum(int, int);

int main()
{
   cout << "The sum of 12 and 45 is:" << endl;
   showSum(12, 45);
   return 0;
}

// Definition of the showSum function
void showSum(int num1, int num2)
{
   int result;
   result = num1 + num2;
   cout << result << endl;
}
```

This program is the C++ version of **Program 3-7** in your textbook.

Program Output

```
The sum of 12 and 45 is:
57
```

Passing Arguments by Reference

When an argument is passed by reference, it means that the function has access to the argument and make changes to it. C++ provides a special type of variable called a *reference variable* that, when used as a function parameter, allows access to the original argument.

A reference variable is an alias for another variable. Any changes made to the reference variable are actually performed on the variable for which it is an alias. By using a reference variable as a parameter, a function may change a variable that is defined in another function.

Reference variables are declared like regular variables, except you place an ampersand (&) in front of the name. Program 3-6 shows an example. In the program, an `int` argument is passed by reference to the `setToZero` function. The `setToZero` function sets its parameter variable to 0, which also sets the original variable that was passed as an argument to 0.

Program 3-6

```
#include <iostream>
using namespace std;

// Function prototype
void setToZero(int &);

int main()
{
   int value = 99;
   cout << "The value is " << value << endl;
   setToZero(value);
   cout << "Now the value is " << value << endl;
   return 0;
}

// Definition of the setToZero function
void setToZero(int &num)
{
   num = 0;
}
```

Program Output

```
The value is 99
Now the value is 0
```

Global Variables and Global Constants

To declare a global variable or constant in a C++ program, you write the declaration outside of all functions, and above the definitions of the functions. As a result, all of the functions in the program have access to the variable or constant.

Chapter 3 in your textbook warns against the use of global variables because they make programs difficult to debug. Global constants are permissible, however, because statements in the program cannot change their value. Program 3-7 demonstrates how to declare such a constant. Notice that in line 9 we have declared a constant named INTEREST_RATE . The declaration is not inside any of the functions, and is written above all function definitions. As a result, the constant is available to all of the functions in the program.

Program 3-7

```
 1 #include <iostream>
 2 using namespace std;
 3 
 4 // Function prototypes
 5 void function2();
```

```
void function3();

// Global constant
const double INTEREST_RATE = 0.05;

int main()
{
   // Statements here have access to
   // the INTEREST_RATE constant.
   return 0;
}

void function2()
{
   // Statements here have access to
   // the INTEREST_RATE constant.
}

void function3()
{
   // Statements here have access to
   // the INTEREST_RATE constant.
}
```

Chapter 4

This chapter accompanies Chapter 4 of Starting Out with Programming Logic and Design, 5th Edition

Decision Structures and Boolean Logic

Relational Operators and the `if` Statement

C++'s relational operators, shown in Table 4-1, are exactly the same as those discussed in your textbook.

Table 4-1 Relational Operators

Operator	Meaning
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`==`	Equal to
`!=`	Not equal to

The relational operators are used to create Boolean expressions, and are commonly used with `if` statements. Here is the general format of the `if` statement in C++:

```
if (BooleanExpression)
{
   statement;
   statement;
   etc;
}
```

The statement begins with the word `if`, followed by a Boolean expression that is enclosed in a set of parentheses. Beginning on the next line is a set of statements that are enclosed in curly braces. When the `if` statement executes, it evaluates the Boolean expression. If the expression is true, the statements inside the curly braces are executed. If the Boolean expression is false, the statements inside the curly braces are skipped. We sometimes say that the statements inside the curly braces are conditionally executed because they are executed only under the condition that the Boolean expression is true.

If you are writing an `if` statement that has only one conditionally executed statement, you do not have to enclose the conditionally executed statement inside curly braces. Such an `if` statement can be written in the following general format:

```
if (BooleanExpression)
   statement;
```

When an `if` statement written in this format executes, the Boolean expression is tested. If it is true, the one statement that appears on the next line will be executed. If the Boolean expression is false, however, that one statement is skipped.

Program 4-1 demonstrates the `if` statement. This is the C++ version of pseudocode Program 4-1 in your textbook.

Program 4-1

This program is the C++ version of **Program 4-1** in your textbook.

```cpp
#include <iostream>
using namespace std;

int main()
{
   // Declare variables
   double test1, test2, test3, average;

   // Get test 1
   cout << "Enter the score for test #1." << endl;
   cin >> test1;

   // Get test 2
   cout << "Enter the score for test #2." << endl;
   cin >> test2;

   // Get test 3
   cout << "Enter the score for test #3." << endl;
   cin >> test3;

   // Calculate the average score.
   average = (test1 + test2 + test3) / 3;

   // Display the average.
   cout << "The average is " << average << endl;

   // If the average is greater than 95
   // congratulate the user.
   if (average > 95)
      cout << "Way to go! Great average!" << endl;

   return 0;
}
```

Program Output

```
Enter the score for test #1.
```

```
100 [Enter]
Enter the score for test #2.
90 [Enter]
Enter the score for test #3.
95 [Enter]
The average is 95
```

Dual Alternative Decision Structures

You use the `if-else` statement in C++ to create a dual alternative decision structure. This is the format of the `if-else` statement:

```
if (BooleanExpression)
{
    statement;
    statement;
    etc;
}
else
{
    statement;
    statement;
    etc;
}
```

An `if-else` statement has two parts: an `if` clause and an `else` clause. Just like a regular `if` statement, the `if-else` statement tests a Boolean expression. If the expression is true, the block of statements following the `if` clause is executed, and then control of the program jumps to the statement that follows the `if-else` statement. If the Boolean expression is false, the block of statements following the `else` clause is executed, and then control of the program jumps to the statement that follows the `if-else` statement.

The `if-else` statement has two sets of conditionally executed statements. One set is executed only under the condition that the Boolean expression is true, and the other set is executed only under the condition that the Boolean expression is false. Under no circumstances will both sets of conditionally executed statement be executed.

If either set of conditionally executed statements contains only one statement, the curly braces are not required. For example, the following general format shows only one statement following the `if` clause and only one statement following the `else` clause:

```
if (BooleanExpression)
    statement;
else
    statement;
```

Program 4-2 shows an example. This is the C++ version of pseudocode Program 4-2 in your textbook. The program gets the number of hours that the user has worked (line 19) and the user's hourly pay rate (line 23). The `if-else` statement in lines 26 through 29 determines whether the user has worked more than 40 hours. If so, the `calcPayWithOT` function is called in line 27. Otherwise the `calcRegularPay` function is called in line 29.

Program 4-2

This program is the C++ version of **Program 4-2** in your textbook.

```
1 #include <iostream>
2 using namespace std;
3
4 // Globally visible constants.
5 const int BASE_HOURS = 40;
6 const double OT_MULTIPLIER = 1.5;
7
8 // Function prototypes
9 void getHoursWorked(double &);
10 void getPayRate(double &);
11 void calcPayWithOT(double, double, double &);
12 void calcRegularPay(double, double, double &);
13
14 int main()
15 {
16    // Declare local variables
17    double hoursWorked, payRate, grossPay;
18
19    // Get the number of hours worked.
20    cout << "Enter the number of hours worked." << endl;
21    cin >> hoursWorked;
22
23    // Get the hourly pay rate.
24    cout << "Enter the hourly pay rate." << endl;
25    cin >> payRate;
26
27    // Calculate the gross pay.
28    if (hoursWorked > BASE_HOURS)
29       calcPayWithOT(hoursWorked, payRate, grossPay);
30    else
31       calcRegularPay(hoursWorked, payRate, grossPay);
32
33    // Display the gross pay.
34    cout << "The gross pay is $" << grossPay << endl;
35
36    return 0;
37 }
38
39 // The getHoursWorked function gets the number of
40 // hours worked and stores it in the hours parameter.
41 void getHoursWorked(double &hours)
42 {
```

```cpp
    cout << "Enter the number or hours worked." << endl;
    cin >> hours;
}

// The getPayRate function gets the hourly pay rate
// and stores it in the rate parameter.
void getPayRate(double &rate)
{
    cout << "Enter the hourly pay rate." << endl;
    cin >> rate;
}

// The calcPayWithOT function calculates pay with
// overtime. The gross pay is stored in the gross
// parameter.
void calcPayWithOT(double hours, double rate, double &gross)
{
    // Local variables
    double overtimeHours, overtimePay;

    // Calculate the number of overtime hours.
    overtimeHours = hours - BASE_HOURS;

    // Calculate the overtime pay
    overtimePay = overtimeHours * rate * OT_MULTIPLIER;

    // Calculate the gross pay.
    gross = BASE_HOURS * rate + overtimePay;
}

// The calcRegularPay function calculates pay with
// no overtime and stores it in the gross parameter.
void calcRegularPay(double hours, double rate, double &gross)
{
        gross = hours * rate;
}
```

Program Output

Enter the number of hours worked.
100 [*Enter*]
Enter the hourly pay rate.
10 [*Enter*]
The gross pay is $1300

Program 4-3 shows an example that tests the value of a string. This is the C++ version of pseudocode Program 4-3 in your textbook.

Program 4-3

```cpp
#include <iostream>
#include <string>
using namespace std;

int main()
{
   // A variable to hold a password.
   string password;

   // Prompt the user to enter the password.
   cout << "Enter the password." << endl;
   cin >> password;

   // Determine whether the correct password
   // was entered.
   if (password == "prospero")
      cout << "Password accepted." << endl;
   else
      cout << "Sorry, that is not the correct password."
           << endl;

   return 0;
}
```

This program is the C++ version of **Program 4-3** in your textbook.

Program Output

Enter the password.
ferdinand [*Enter*]
Sorry, that is not the correct password.

Program Output

Enter the password.
prospero [*Enter*]
Password accepted

Nested Decision Structures

Program 4-5 shows an example of nested decision structures. As noted in your textbook, this type of nested decision structure can also be written as an `if-else-if` statement, as shown in Program 4-6.

Program 4-5

```cpp
#include <iostream>
using namespace std;

int main()
{
   // A variable to hold the temperature.
```

```
   int temp;

   // Prompt the user to enter the temperature.
   cout << "What is the outside temperature?" << endl;
   cin >> temp;

   // Determine the type of weather we're having.
   if (temp < 30)
      cout << "Wow! That's cold!" << endl;
   else
   {
      if (temp < 50)
        cout << "A little chilly." << endl;
      else
      {
        if (temp < 80)
            cout << "Nice and warm." << endl;
        else
          cout << "Whew! It's hot!" << endl;
      }
   }
   return 0;
}
```

Program Output

```
What is the outside temperature?
```
20 [*Enter*]
```
Wow! That's cold!
```

Program Output

```
What is the outside temperature?
```
45 [*Enter*]
```
A little chilly.
```

Program Output

```
What is the outside temperature?
```
70 [*Enter*]
```
Nice and warm.
```

Program Output

```
What is the outside temperature?
```
90 [*Enter*]
```
Whew! Its hot!
```

Program 4-6

```
#include <iostream>
using namespace std;

int main()
{
   // A variable to hold the temperature.
   int temp;

   // Prompt the user to enter the temperature.
   cout << "What is the outside temperature?" << endl;
   cin >> temp;

   // Determine the type of weather we're having.
   if (temp < 30)
      cout << "Wow! That's cold!" << endl;
   else if (temp < 50)
      cout << "A little chilly." << endl;
   else if (temp < 80)
      cout << "Nice and warm." << endl;
   else
      cout << "Whew! It's hot!" << endl;

   return 0;
}
```

Program Output

```
What is the outside temperature?
20 [Enter]
Wow! That's cold!
```

Program Output

```
What is the outside temperature?
45 [Enter]
A little chilly.
```

Program Output

```
What is the outside temperature?
70 [Enter]
Nice and warm.
```

Program Output

```
What is the outside temperature?
90 [Enter]
Whew! Its hot!
```

The Case Structure (`switch` Statement)

In C++, case structures are written as `switch` statements. Here is the general format of the `switch` statement:

```
switch (testExpression)          This is an integer variable or an expression.
{
   case value_1:
      statement                  These statements are executed if the
      statement                  testExpression is equal to value_1.
      etc.
      break;

   case value_2:
      statement                  These statements are executed if the
      statement                  testExpression is equal to value_2.
      etc.
      break;

  Insert as many case sections as necessary

   case value_N:
      statement                  These statements are executed if the
      statement                  testExpression is equal to value_N.
      etc.
      break;

   default:
      statement                  These statements are executed if the
      statement                  testExpression is not equal to any of the values
      etc.                       listed after the Case statements.
}                                This is the end of the switch statement.
```

The first line of the structure starts with the word `switch`, followed by a *testExpression* which is encloses in parentheses. The *testExpression* is a value or expression of one of these types: `char`, `byte`, `short`, or `int`. Beginning at the next line is a block of code enclosed in curly braces. Inside this block of code is one or more *`case` sections*. A `case` section begins with the word `case`, followed by a value, followed by a colon. Each `case` section contains one or more statements, followed by a `break` statement. At the end is an optional *`default` section*.

When the `switch` statement executes, it compares the value of the *testExpression* with the values that follow each of the `case` statements (from top to bottom). When it finds a `case` value that matches the *testExpression*'s value, the program branches to the `case` statement. The statements that follow the `case` statement are executed, until a break statement is encountered. At that point program jumps out of the `switch` statement. If the *testExpression* does not match any of the `case` values, the program branches to the `default` statement and executes the statements that immediately following it.

For example, the following code performs the same operation as the flowchart in Figure 4-18 in your textbook:

```
switch (month)
{
   case 1:
      cout << "January" << endl;
      break;

   case 2:
      cout << "February" << endl;
      break;

   case 3:
      cout << "March" << endl;
      break;

   default:
      cout << "Error: Invalid month" << endl;
}
```

In this example the *testExpression* is the `month` variable. If the value in the `month` variable is 1, the program will branch to the `case 1:` section and execute the `cout << "January" << endl;` statement that immediately follows it. If the value in the `month` variable is 2, the program will branch to the `case 2:` section and execute the `cout << "February" << endl;` statement that immediately follows it. If the value in the `month` variable is 3, the program will branch to the `case 3:` section and execute the `cout << "March" << endl;` statement that immediately follows it. If the value in the `month` variable is not 1, 2, or 3, the program will branch to the `default:` section and execute the `cout << "Error: Invalid month" << endl;` statement that immediately follows it.

Here are some important points to remember about the `switch` statement:

- The *testExpression* must be a value or expression of one of the integer data types (including `char`).
- The value that follows a `case` statement must be a literal or a named constant of one of the integer data types (including `char`).
- The `break` statement that appears at the end of a `case` section is optional, but in most situations you will need it. If the program executes a `case` section that does not

end with a `break` statement, it will continue executing the code in the very next `case` section.

- The `default` section is optional, but in most situations you should have one. The `default` section is executed when the *testExpression* does not match any of the `case` values.
- Because the `default` section appears at the end of the `switch` statement, it does not need a `break` statement.

Program 4-7 shows a complete example. This is the C++ version of pseudocode Program 4-8 in your textbook.

Program 4-7

This program is the C++ version of **Program 4-8** in your textbook.

```
#include <iostream>
using namespace std;

int main()
{
   // Constants for the TV prices
   const double MODEL_100_PRICE = 199.99;
   const double MODEL_200_PRICE = 269.99;
   const double MODEL_300_PRICE = 349.99;

   // Constants for the TV sizes
   const int MODEL_100_SIZE = 24;
   const int MODEL_200_SIZE = 27;
   const int MODEL_300_SIZE = 32;

   // Variable for the model number
   int modelNumber;

   // Get the model number.
   cout << "Which TV are you interested in?" << endl;
   cout << "The 100, 200, or 300?" << endl;
   cin >> modelNumber;

   // Display the price and size.
   switch (modelNumber)
   {
      case 100:
        cout << "Price: $" << MODEL_100_PRICE << endl;
        cout << "Size: " << MODEL_100_SIZE << endl;
        break;
      case 200:
        cout << "Price: $" << MODEL_200_PRICE << endl;
        cout << "Size: " << MODEL_200_SIZE << endl;
        break;
```

```
      case 300:
        cout << "Price $" << MODEL_300_PRICE << endl;
        cout << "Size: " << MODEL_300_SIZE << endl;
        break;
      default:
        cout << "Invalid model number." << endl;
   }

   return 0;
}
```

Program Output

```
Which TV are you interested in?
The 100, 200, or 300?
100 [Enter]
Price: $199.99
Size: 24
```

Program Output

```
Which TV are you interested in?
The 100, 200, or 300?
200 [Enter]
Price: $269.99
Size: 27
```

Program Output

```
Which TV are you interested in?
The 100, 200, or 300?
300 [Enter]
Price: $349.99
Size: 32
```

Program Output

```
Which TV are you interested in?
The 100, 200, or 300?
500 [Enter]
Invalid model number.
```

Logical Operators

C++'s logical operators look different than the ones used in your textbook's pseudocode, but they work in the same manner. Table 4-2 shows C++'s logical operators.

Table 4-2 C++'s Logical Operators

Operator	Meaning
&&	This is the logical AND operator. It connects two Boolean expressions into one compound expression. Both subexpressions must be true for the compound expression to be true.
||	This is the logical OR operator. It connects two Boolean expressions into one compound expression. One or both subexpressions must be true for the compound expression to be true. It is only necessary for one of the subexpressions to be true, and it does not matter which.
!	This is the logical NOT operator. It is a unary operator, meaning it works with only one operand. The operand must be a Boolean expression. The `!` operator reverses the truth of its operand. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

For example, the following `if` statement checks the value in `x` to determine if it is in the range of 20 through 40:

```
if (x >= 20 && x <= 40)
   cout << x << " is in the acceptable range." << endl;
```

The Boolean expression in the `if` statement will be true only when `x` is greater than or equal to 20 AND less than or equal to 40. The value in `x` must be within the range of 20 through 40 for this expression to be true. The following statement determines whether `x` is outside the range of 20 through 40:

```
if (x < 20 || x > 40)
   cout << x << " is outside the acceptable range." << endl;
```

Here is an `if` statement using the `!` operator:

```
if (!(temperature > 100))
   cout << "This is below the maximum temperature." << endl;
```

First, the expression `(temperature > 100)` is tested and a value of either true or false is the result. Then the `!` operator is applied to that value. If the expression `(temperature > 100)` is true, the `!` operator returns false. If the expression `(temperature > 100)` is false, the `!` operator returns true. The previous code is equivalent to asking: “Is the temperature not greater than 100?”

`bool` Variables

In C++ you use the `bool` data type to create Boolean variables. A `boolean` variable can hold one of two possible values: `true` or `false`. Here is an example of a `bool` variable declaration:

```
bool highScore;
```

Boolean variables are commonly used as flags that signal when some condition exists in the program. When the flag variable is set to false, it indicates the condition does not yet exist. When the flag variable is set to true, it means the condition does exist.

For example, suppose a test grading program has a `bool` variable named `highScore`. The variable might be used to signal that a high score has been achieved by the following code:

```
if (average > 95)
   highScore = true;
```

Later, the same program might use code similar to the following to test the `highScore` variable, in order to determine whether a high score has been achieved:

```
if (highScore)
   cout << "That's a high score!" << endl;
```

Chapter 5 Repetition Structures

This chapter accompanies Chapter 5 of Starting Out with Programming Logic and Design, 5th Edition

Repetition Structures

Incrementing and Decrementing Variables

To *increment* a variable means to increase its value and to *decrement* a variable means to decrease its value. Both of the following statements increment the variable `num` by one:

```
num = num + 1;
num += 1;
```

And `num` is decremented by one in both of the following statements:

```
num = num - 1;
num -= 1;
```

Incrementing and decrementing is so commonly done in programs that C++ provides a set of simple unary operators designed just for incrementing and decrementing variables. The increment operator is ++ and the decrement operator is `--.` The following statement uses the ++ operator to add 1 to `num`:

```
num++;
```

After this statement executes, the value of `num` will increased by one. The following statement uses the `--` operator to subtract 1 from `num`:

```
num--;
```

In these examples, we have written the ++ and `--` operators after their operands (or, on the right side of their operands). This is called *postfix mode*. The operators can also be written before (or, on the left side) of their operands, which is called *prefix mode*. Here are examples:

```
++num;
--num;
```

When you write a simple statement to increment or decrement a variable, such as the ones shown here, it doesn't matter if you use prefix mode or postfix mode. The operators do the same thing in either mode. However, if you write statements that mix these operators with other operators or with other operations, there is a difference in the way the two modes work. Such complex code can be difficult to understand and debug. When we use the increment and decrement operators, we will do so only in ways that are straightforward and easy to understand, such as the statements previously shown.

We introduce these operators at this point because they are commonly used in certain types of loops. When we discuss the `for` loop you will see these operators used often.

The `while` Loop

In C++, the `while` loop is written in the following general format:

```
while (BooleanExpression)
{
    statement;
    statement;
    etc;
}
```

We will refer to the first line as the *`while` clause*. The `while` clause begins with the word `while`, followed by a Boolean expression that is enclosed in parentheses. Beginning on the next line is a block of statements that are enclosed in curly braces. This block of statements is known as the *body* of the loop.

When the `while` loop executes, the Boolean expression is tested. If the Boolean expression is true, the statements that appear in the body of the loop are executed, and then the loop starts over. If the Boolean expression is false, the loop ends and the program resumes execution at the statement immediately following the loop.

We say that the statements in the body of the loop are conditionally executed because they are executed only under the condition that the Boolean expression is true. If you are writing a `while` loop that has only one statement in its body, you do not have to enclose the statement inside curly braces. Such a loop can be written in the following general format:

```
while (BooleanExpression)
    statement;
```

When a `while` loop written in this format executes, the Boolean expression is tested. If it is true, the one statement that appears on the next line will be executed, and then the loop starts over. If the Boolean expression is false, however, the loop ends.

Program 5-1 shows an example of the `while` loop. This is the C++ version of pseudocode Program 5-2 in your textbook.

Program 5-1

This program is the C++ version of **Program 5-2** in your textbook.

```
#include <iostream>
using namespace std;

int main()
{
   // Variable to hold the temperature
   double temperature;

   // Constant for the maximum temperature
   const double MAX_TEMP = 102.5;

```

```cpp
    // Get the substance's temperature.
    cout << "Enter the substance's temperature." << endl;
    cin >> temperature;

    // If necessary, adjust the thermostat.
    while (temperature > MAX_TEMP)
    {
       cout << "The temperature is too high." << endl;
       cout << "Turn the thermostat down and wait" << endl;
       cout << "five minutes. Take the temperature" << endl;
       cout << "again and enter it here." << endl;
       cin >> temperature;
    }

    // Remind the user to check the temperature
    // again in 15 minutes.
    cout << "The temperature is acceptable." << endl;
    cout << "Check it again in 15 minutes." << endl;
    return 0;
}
```

Program Output

```
Enter the substance's temperature.
200 [Enter]
The temperature is too high.
Turn the thermostat down and wait
five minutes. Take the temperature
again and enter it here.
130 [Enter]
The temperature is too high.
Turn the thermostat down and wait
five minutes. Take the temperature
again and enter it here.
100 [Enter]
The temperature is acceptable.
Check it again in 15 minutes.
```

The `do-while` Loop

Here is the general format of the `do-while` loop:

```
do
{
    statement;
    statement;
    etc;
} while (BooleanExpression);
```

As with the `while` loop, the braces are optional if there is only one statement in the body of the loop. This is the general format of the `do-while` loop with only one conditionally executed statement:

```
do
    statement;
while (BooleanExpression);
```

Notice that a semicolon appears at the very end of the `do-while` statement. This semicolon is required, and leaving it out is a common error.

The `for` Loop

The `for` loop is specifically designed to initialize, test, and increment a counter variable. Here is the general format of the `for` loop:

```
for (InitializationExpression; TestExpression; IncrementExpression)
{
    statement;
    statement;
    etc.
}
```

The statements that appear inside the curly braces are the body of the loop. These are the statements that are executed each time the loop iterates. As with other control structures, the curly braces are optional if the body of the loop contains only one statement, as shown in the following general format:

```
for (InitializationExpression; TestExpression; IncrementExpression)
    statement;
```

The first line of the `for` loop is the *loop header*. After the key word `for`, there are three expressions inside the parentheses, separated by semicolons. (Notice there is not a semicolon after the third expression.)

The first expression is the *initialization expression*. It is normally used to initialize a counter variable to its starting value. This is the first action performed by the loop, and it is only done once. The second expression is the *test expression*. This is a Boolean expression that controls the execution of the loop. As long as this expression is true, the body of the `for` loop will repeat. The `for` loop is a pretest loop, so it evaluates the test expression before each iteration. The third expression is the *increment expression*. It executes at the end of each iteration. Typically, this is a statement that increments the loop's counter variable.

Here is an example of a simple `for` loop that prints "Hello" five times:

```
for (count = 1; count <= 5; count++)
{
    cout << "Hello" << endl;
}
```

In this loop, the initialization expression is `count = 1`, the test expression is `count <= 5`, and the increment expression is `count++`. The body of the loop has one statement. This is a summary of what happens when this loop executes:

(1) The initialization expression `count = 1` is executed. This assigns 1 to the `count` variable.
(2) The expression `count <= 5` is tested. If the expression is true, continue with step 3. Otherwise, the loop is finished.
(3) The statement `cout << "Hello" << endl;` is executed.
(4) The increment expression `count++` is executed. This adds 1 to the `count` variable.
(5) Go back to step 2.

Program 5-2 shows an example. This is the C++ version of pseudocode Program 5-8 in your textbook.

Program 5-2

```cpp
#include <iostream>
using namespace std;

int main()
{
   int counter;
   const int MAX_VALUE = 5;

   for (counter = 1; counter <= MAX_VALUE; counter++)
   {
      cout << "Hello world" << endl;
   }

   return 0;
}
```

This program is the C++ version of **Program 5-8** in your textbook.

Program Output

```
Hello world
Hello world
Hello world
Hello world
Hello world
```

Program 5-3 shows another example. The `for` loop in this program uses the value of the counter variable in a calculation in the body of the loop. This is the C++ version of pseudocode Program 5-9 in your textbook.

I should point out the `"\t"` formatting characters that are used in lines 12 and 20. These are special formatting characters known as the *tab escape sequence*. The escape sequence works similarly to the word `Tab` that is used in pseudocode in your textbook. As you can see in the program output, the `"\t"` characters are not displayed on the screen, but rather cause the output cursor to "tab over." It is useful for aligning output in columns on the screen.

Program 5-3

This is the C++ version of **Program 5-9** in your textbook.

```cpp
#include <iostream>
using namespace std;

int main()
{
   // Variables
   int counter, square;

   // Constant for the maximum value
   const int MAX_VALUE = 10;

   // Display table headings.
   cout << "Number\tSquare" << endl;
   cout << "-----------------------" << endl;

   // Display the numbers 1 through 10 and
   // their squares.
   for (counter = 1; counter <= MAX_VALUE; counter++)
   {
      square = counter * counter;
      cout << counter << "\t\t" << square << endl;
   }

   return 0;
}
```

Program Output

```
Number      Square
-----------------------
1           1
2           4
3           9
4           16
5           25
6           36
7           49
8           64
9           81
10          100
```

Incrementing by Values Other Than 1

Program 5-4 demonstrates that the update expression does not have to increment the counter variable by 1. In fact, the update expression can be any expression that you wish to execute at the end of each loop iteration. In Program 5-4 , the counter variable is incremented by 2, causing the statement in line 14 to display only the odd numbers in the range of 1 through 11. This program is the C++ version of pseudocode Program 5-10 in your textbook.

Program 5-4

This is the C++ version of **Program 5-10** in your textbook.

```
#include <iostream>
using namespace std;

int main()
{
   // Declare a counter variable.
   int counter;

   // Constant for the maximum value
   const int MAX_VALUE = 11;

   for (counter = 1; counter <= MAX_VALUE; counter = counter + 2)
   {
      cout << counter << endl;
   }

   return 0;
}
```

Program Output

```
1
3
5
7
9
11
```

Calculating a Running Total

Your textbook discusses the common programming task of calculating the sum of a series of values, also known as calculating a running total. Program 5-5 demonstrates how this is done in C++. The `total` variable that is declared in line 16 is the accumulator variable. Notice that it is initialized with the value 0. During each loop iteration the user enters a number, and in line 30 this number is added to the value already stored in `total`. The `total` variable accumulates the sum of all the numbers entered by the user. This program is the C++ version of pseudocode Program 5-18 in your textbook.

Program 5-5

This is the C++ version of **Program 5-18** in your textbook.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6     // Declare a variable to hold each number
7     // entered by the user.
8     int number;
9
10    // Declare an accumulator variable,
11    // initialized with 0.
12    int total = 0;
13
```

```cpp
   // Declare a counter variable for the loop.
   int counter;

   // Explain what we are doing.
   cout << "This program calculates the" << endl;
   cout << "total of five numbers." << endl;

   // Get five numbers and accumulate them.
   for (counter = 1; counter <= 5; counter++)
   {
      cout << "Enter a number: " << endl;
      cin >> number;
      total = total + number;
   }

   // Display the total of the numbers.
   cout << "The total is " << total << endl;

   return 0;
}
```

Program Output

```
This program calculates the
total of five numbers.
Enter a number:
1 [Enter]
Enter a number:
2 [Enter]
Enter a number:
3 [Enter]
Enter a number:
4 [Enter]
Enter a number:
5 [Enter]
The total is 15
```

Chapter 6

This chapter accompanies Chapter 6 of Starting Out with Programming Logic and Design, 5th Edition

Functions

Terminology

Chapter 6 in your textbook is about functions, which are modules that return a value. In C++, we use the term *function* for both modules and functions. In C++, functions can either return a value or not. Functions that do not return a value are known as *void functions*, and functions that return a value are known as *value-returning functions*.

Generating Random Integers

The C++ library provides a value-returning function named `rand()` that returns a random number. (The `rand()` function requires the directive `#include <cstdlib>`). The random number that is returned from the `rand()` function is an `int`. Here is an example of its usage:

```
y = rand();
```

After this statement executes, the variable `y` will contain a random number. In actuality, the numbers produced by `rand()`are pseudorandom. The function uses an algorithm that produces the same sequence of numbers each time the program is repeated on the same system. For example, suppose the following statements are executed.

```
cout << rand() << endl;
cout << rand() << endl;
cout << rand() << endl;
```

The three numbers displayed will appear to be random, but each time the program runs, the same three values will be generated. In order to randomize the results of `rand()`, the `srand()` function must be used The. `srand()` function accepts an `unsigned int` argument, which acts as a seed value for the algorithm. By specifying different seed values, `rand()` will generate different sequences of random numbers.

A common practice for getting unique seed values is to call the `time()` function, which is part of the C++ standard library. The `time()` function returns the number of seconds that have elapsed since midnight, January 1, 1970. The `time()` function requires the directive `#include <ctime>`. When you call the `time()` function, you pass 0 as an argument. The following code snippet shows an example:

```
// Get the system time.
unsigned seed = time(0);

// Seed the random number generator.
srand(seed);

// Display a random number.
cout << rand() << endl;
```

If you wish to limit the range of the random number, use the following formula.

```
y = 1 + rand() % maxRange;
```

The `maxRange` value is the upper limit of the range. For example, if you wish to generate a random number in the range of 1 through 100, use the following statement.

```
y = 1 + rand() % 100;
```

This is how the statement works: Look at the following expression.

```
rand() % 100
```

Assuming `rand()` returns 37894, the value of the expression above is 94. That is because 37894 divided by 100 is 378 with a remainder of 94. (The modulus operator returns the remainder.) But, what if `rand()` returns a number that is evenly divisible by 100, such as 500? The expression above will return a 0. If we want a number in the range 1 – 100, we must add 1 to the result. That is why we use the expression `1 + rand() % 100`.

Program 6-1 shows a complete demonstration. This is the C++ version of pseudocode Program 6-2 in your textbook. Let's take a closer look at the code:

Program 6-1

```
#include <iostream>
#include <cstdlib> // Needed for the rand and srand functions
#include <ctime>   // Needed for the time function
using namespace std;

int main()
{
```

This is the C++ version of **Program 6-2** in your textbook.

```
    // Declare variables.
    int number, counter;

    // Get the system time.
    unsigned seed = time(0);

    // Seed the random number generator.
    srand(seed);

    // The following loop displays five random
    // numbers, each in the range of 1 through 100.
    for (counter = 1; counter <= 5; counter++)
    {
       number = 1 + rand() % 100;
       cout << number << endl;
    }

    return 0;
}
```

Program Output

```
57
74
10
71
67
```

Writing Your Own Value-Returning Functions

Up to now, all of the functions that you have written have been `void` functions, which means that they do not return a value. Recall that the key word `void` appears in a function header, as shown here:

```
void displayMessage()
```

This is the header for a function named `displayMessage`. Because it is a `void` function, it works like the modules that we discussed in Chapter 3 of your textbook. It is simply a procedure that executes when it is called, and does not return any value back to the statement that called it.

You can also write your own value-returning functions in C++. When you are writing a value-returning function, you must decide what type of value the function will return. This is because, instead of specifying `void` in the function header, you must specify the data type of the value that will be returned. A value-returning function will use `int`, `double`, `string`, `bool`, or any other valid data type in its header. Here is an example of a function that returns an `int` value:

Instead of `void`, this function specifies `int` as its return type.

```
int sum(int num1, int num2)
{
   int result;
   result = num1 + num2;
   return result;
}
```

The name of this function is `sum`. Notice in the function header that instead of the word `void` we have specified `int` as the return type. This code defines a function named `sum` that accepts two `int` arguments. The arguments are passed into the parameter variables `num1` and `num2`. Inside the function, a local variable, `result`, is declared. The parameter variables `num1` and `num2` are added, and their sum is assigned to the `result` variable. The last statement in the function is as follows:

```
return result;
```

This is a *`return` statement*. You must have a `return` statement in a value-returning function. It causes the function to end execution and it returns a value to the statement that called the function. In a value-returning function, the general format of the return statement is as follows:

```
return Expression;
```

`Expression` is the value to be returned. It can be any expression that has a value, such as a variable, literal, or mathematical expression. In this case, the `sum` function returns the value of the `result` variable. Program 6-2 shows a complete C++ program that demonstrates this function. The program is the C++ version of pseudocode Program 6-6 in your textbook.

Program 6-2

This is the C++ version of **Program 6-6** in your textbook.

```
#include <iostream>
using namespace std;

// Function prototype
int sum(int, int);

int main()
{
   // Local variables
   int firstAge, secondAge, total;

   // Get the user's age and the user's
   // best friend's age.
```

```
    cout << "Enter your age." << endl;
    cin >> firstAge;
    cout << "Enter your best friend's age." << endl;
    cin >> secondAge;

    // Get the sum of both ages.
    total = sum(firstAge, secondAge);

    // Display the sum.
    cout << "Together you are " << total
        << " years old." << endl;

    return 0;
}

// The sum function accepts two int arguments and
// returns the sum of those arguments as an int.
int sum(int num1, int num2)
{
    int result;
    result = num1 + num2;
    return result;
}
```

Program Output

```
Enter your age.
22 [Enter]
Enter your best friend's age.
24 [Enter]
Together you are 46 years old.
```

Returning Strings

The following code shows an example of how a function can return string. Notice that the function header specifies `string` as the return type. This function accepts two string arguments (a person's first name and last name). It combines those two strings into a string that contains the person's full name. The full name is then returned.

```
string fullName(string firstName, string lastName)
{
   string name;

   name = firstName + " " + lastName;
   return name;
}
```

The following code snippet shows how we might call the function:

```
string customerName;
customerName = fullName("John", "Martin");
```

After this code executes, the value of the `custerName` variable will be "John Martin".

Returning a `bool` Value

Functions can also return `bool` values. The following function accepts an argument and returns `true` if the argument is within the range of 1 through 100, or `false` otherwise:

```
bool isValid(int number)
{
   bool status;

   if (number >= 1 && number <= 100)
      status = true;
   else
      status = false;

   return status;
}
```

The following code shows an `if-else` statement that uses a call to the function:

```
int value = 20;

if (isValid(value))
   cout << "The value is within range." << endl;
else
   cout << "The value is out of range." << endl;
```

When this code executes, the message "The value is within range." will be displayed.

Standard Library Math Functions

In Chapter 2 you were introduced to the `pow` function, which returns the value of a number raised to a power. Table 6-1 describes several of the standard library's math functions, including `pow`. (To use any of these functions, write the directive `#include <cmath>` in your program.)

Table 6-1 Several Standard Library Math Functions

`abs`	*Example Usage:* `y = abs(x);` *Description:* Returns the absolute value of the argument. This function can accept and return values of the `int` or `long` data types.

`acos`	*Example Usage:* `y = acos(x);` *Description:* Returns the arc-cosine of the argument. The argument should be the cosine of an angle. (The argument's value must be in the range from –1.0 through 1.0.) The function can accept an argument of the `double`, `float, or long double` data types. The value that is returned will be of the same data type as the argument.
`asin`	*Example Usage:* `y = asin(x);` *Description:* Returns the arc-sine of the argument. The argument should be the sine of an angle. (The argument's value must be in the range from –1.0 through 1.0.) The function can accept an argument of the `double`, `float, or long double` data types. The value that is returned will be of the same data type as the argument.
`atan`	*Example Usage:* `y = atan(x);` *Description:* Returns the arc-tangent of the argument. The argument should be the tangent of an angle. The function can accept an argument of the `double`, `float, or long double` data types. The value that is returned will be of the same data type as the argument.
`ceil`	*Example Usage:* `y = ceil(x);` *Description:* Returns the smallest number that is greater than or equal to the argument. The function can accept an argument of the `double`, `float, or long double` data types. The value that is returned will be of the same data type as the argument.
`cos`	*Example Usage:* `y = cos(x);` *Description:* Returns the cosine of the argument. The argument should be an angle expressed in radians. The function can accept an argument of the `double`, `float, or long double` data types. The value that is returned will be of the same data type as the argument.

`exp`	*Example Usage:* `y = exp(x);` *Description:* Computes the exponential function of the argument, which is e^x. The function can accept an argument of the `double`, `float, or long double` data types. The value that is returned will be of the same data type as the argument.
`floor`	*Example Usage:* `y = floor(x);` *Description:* Returns the largest number that is less than or equal to the argument. The function can accept an argument of the `double`, `float, or long double` data types. The value that is returned will be of the same data type as the argument.
`log`	*Example Usage:* `y = log(x);` *Description:* Returns the natural logarithm of the argument. The function can accept an argument of the `double`, `float, or long double` data types. The value that is returned will be of the same data type as the argument..
`pow`	*Example Usage:* `y = pow(x, z);` *Description:* Returns the value of the first argument raised to the power of the second argument. The function can accept arguments of the `double`, `float, or long double` data types. The value that is returned will be of the same data type as the arguments.
`sin`	*Example Usage:* `y = sin(x);` *Description:* Returns the sine of the argument. The argument should be an angle expressed in radians. The function can accept an argument of the `double`, `float, or long double` data types. The value that is returned will be of the same data type as the argument.

`sqrt`	*Example Usage:* `y = sqrt(x);` *Description:* Returns the square root of the argument. The function can accept an argument of the `double`, `float, or long double` data types. The value that is returned will be of the same data type as the argument.
`tan`	*Example Usage:* `y = tan(x);` *Description:* Returns the tangent of the argument. The argument should be an angle expressed in radians. The function can accept an argument of the `double`, `float, or long double` data types. The value that is returned will be of the same data type as the argument.

String Functions

Getting a String's length

The following code snippet shows an example of how you get the length of a string in C++:

```
// Declare and initialize a string variable.
string name = "Charlemagne";

// Assign the length of the string to the stringlen variable.
int stringlen = name.length();
```

This code declares a `string` variable named `name`, and initializes it with the string `"Charlemagne"`. Then, it declares an `int` variable named `stringlen`. The `stringlen` variable is initialized with the value returned from the `name.length()` function. This function returns the length of the string stored in `name`. In this case, the value 11 will be returned from the function.

Appending a String to Another String

Appending a string to another string is called concatenation. In C++ you can perform string concatenation using the + operator. Here is an example of how the + operator works with strings:

```
string lastName = "Conway";
string salutation = "Mr. ";
string properName;
properName = salutation + lastName;
```

After this code executes the `properName` variable will contain the string `"Mr. Conway"`.

The `substr` Function

The `substr` function returns part of another string. (A string that is part of another string is commonly referred to as a "substring.") The first argument specifies the substring's starting position and the second argument specifies the substring's length. The character at the starting position is included in the substring. Here is an example of how the function is used:

```
string fullName = "Cynthia Susan Lee";
string middleName = fullName.substr(8, 5);
cout << "The full name is " << fullName << endl;
cout << "The middle name is " << middleName << endl;
```

The code will produce the following output:

```
The full name is Cynthia Susan Lee
The middle name is Susan
```

The `find` Function

In C++ you can use a string's `find` function to perform a task similar to that of the `contains` function discussed in your textbook. The `find` functions searches for substrings within a string. Here is the general format:

```
string1.find(string2, start)
```

In the general format *string1* and *string2* are strings, and *start* is an integer. The search begins at the position passed into *start* and goes to the end of the string. If the string is found, the beginning position of its first occurrence is returned. Otherwise, the special value `string::npos` is returned. The following code shows an example. It displays the starting positions of each occurrence of the word "and" within a string.

```
string str = "and a one and a two and a three";
int position;

cout << "The word and appears at the following locations." << endl;
position = str.find("and", 0);

while (position != string::npos)
{
   cout << position << endl;
   position = str.find("and", position + 1);
}
```

This code produces the following output:

```
The word and appears at the following locations.
0
10
20
```

Chapter 7

This chapter accompanies Chapter 7 of Starting Out with Programming Logic and Design, 5th Edition

Input Validation

Chapter 7 in your textbook discusses the process of input validation in detail. There are no new language features introduced in the chapter, so here we will simply show you a C++ version of the pseudocode Program 7-2. This program uses an input validation loop in lines 41 through 46 to validate that the value entered by the user is not negative.

Program 7-1

This is the C++ version of **Program 7-2** in your textbook.

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 // Function prototype
6 void showRetail();
7
8 int main()
9 {
10    // Local variable
11    string doAnother;
12
13    do
14    {
15       // Calculate and display a retail price.
16       showRetail();
17
18       // Do this again?
19       cout << "Do you have another item? (Enter y for yes.)" << endl;
20       cin >> doAnother;
21    } while (doAnother == "y" || doAnother == "Y");
22
23    return 0;
24 }
25
26 // The showRetail function gets an item's wholesale cost
27 // from the user and displays its retail price.
28 void showRetail()
29 {
30    // Local variables
31    double wholesale, retail;
32
33    // Constant for the markup percentage
34    const double MARKUP = 2.5;
35
36    // Get the wholesale cost.
37    cout << "Enter an item's wholesale cost." << endl;
38    cin >> wholesale;
39
40    // Validate the wholesale cost.
```

```
   while (wholesale < 0)
   {
      cout << "The cost cannot be negative. Please" << endl;
      cout << "enter the correct wholesale cost." << endl;
      cin >> wholesale;
   }

   // Calculate the retail price.
   retail = wholesale * MARKUP;

   // Display the retail price.
   cout << "The retail price is $" << retail << endl;
}
```

Program Output

```
Enter an item's wholesale cost.
-1 [Enter]
The cost cannot be negative. Please
enter the correct wholesale cost.
1.50 [Enter]
The retail price is $3.75
Do you have another item? (Enter y for yes.)
n [Enter]
```

Chapter 8

This chapter accompanies Chapter 8 of Starting Out with Programming Logic and Design, 5th Edition

Arrays

Here is an example of an array declaration in C++:

```
int numbers [6];
```

This statement declares `numbers` as an `int` array. The size declarator specifies that the array has 6 elements. As mentioned in your textbook, it is a good practice to use a named constant for the size declarator, as shown here:

```
const int SIZE = 6;
int numbers[SIZE];
```

Here is another example:

```
const int SIZE = 200;
double temperatures[SIZE];
```

This code snippet declares `temperatures` as an array of 200 `doubles`. Here is one more:

```
const int SIZE = 10;
string names[SIZE];
```

This declares `names` as an array of 10 `strings`.

Array Elements and Subscripts

You access each element of an array with a subscript. As discussed in your textbook, the first element's subscript is 0, the second element's subscript is 1, and so forth. The last element's subscript is the array size minus 1. Program 8-1 shows an example of an array being used to hold values entered by the user. This is the C++ version of pseudocode Program 8-1 in your textbook.

Program 8-1

This is the C++ version of **Program 8-1** in your textbook.

```
#include <iostream>
using namespace std;

int main()
{
   // Create a constant for the number of employees.
   const int SIZE = 3;

   // Declare an array to hold the number of hours
   // worked by each employee.
```

```
    int hours[SIZE];

    // Get the hours worked by employee 1.
    cout << "Enter the hours worked by employee 1." << endl;
    cin >> hours[0];

    // Get the hours worked by employee 2.
    cout << "Enter the hours worked by employee 2." << endl;
    cin >> hours[1];

    // Get the hours worked by employee 3.
    cout << "Enter the hours worked by employee 3." << endl;
    cin >> hours[2];

    // Display the values entered.
    cout << "The hours you entered are:" << endl;
    cout << hours[0] << endl;
    cout << hours[1] << endl;
    cout << hours[2] << endl;
    return 0;
}
```

Program Output

```
Enter the hours worked by employee 1.
40 [Enter]
Enter the hours worked by employee 2.
20 [Enter]
Enter the hours worked by employee 3.
15 [Enter]
The hours you entered are:
40
20
15
```

Using a Loop to Process an Array

It is usually much more efficient to use a loop to access an array's elements, rather than writing separate statements to access each element. Program 8-2 demonstrates how to use a loop to step through an array's elements. This is the C++ version of pseudocode Program 8-3 in your textbook.

Program 8-2

This is the C++ version of **Program 8-3** in your textbook.

```
#include <iostream>
using namespace std;

int main()
{
    // Create a constant for the number of employees.
```

```
   const int SIZE = 3;

   // Declare an array to hold the number of hours
   // worked by each employee.
   int hours[SIZE];

   // Declare a variable to use in the loops.
   int index;

   // Get the hours for each employee.
   for (index = 0; index <= SIZE - 1; index++)
   {
      cout << "Enter the hours worked by "
           << "employee number "
           << (index + 1) << "." << endl;
      cin >> hours[index];
   }

   // Display the values entered.
   cout << endl;
   for (index = 0; index <= SIZE - 1; index++)
      cout << hours[index] << endl;

   return 0;
}
```

Program Output

```
Enter the hours worked by employee 1.
40 [Enter]
Enter the hours worked by employee 2.
20 [Enter]
Enter the hours worked by employee 3.
15 [Enter]

40
20
15
```

Initializing an Array

You can initialize an array with values when you declare it. Here is an example:

```
const int SIZE = 12;
int days[SIZE] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

This statement declares `days` as an array of `int`s, and stores initial values in the array. The series of values inside the braces and separated with commas is called an *initialization list*. These values are stored in the array elements in the order they appear in the list. (The first value, 31, is stored in `days[0]`, the second value, 28, is stored in `days[1]`, and so forth.)

When initializing an array, it is not necessary to specify a size declarator. The C++ compiler will determine the size of the array by counting the number of items in the initialization list. For example, the previous declaration could be written like this:

```
int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Sequentially Searching an Array

Section 8.2 in your textbook discusses the sequential search algorithm, in which a program steps through each of an array's elements searching for a specific value. Program 8-3 shows an example of the sequential search algorithm. This is the C++ version of pseudocode Program 8-6 in the textbook.

Program 8-3

This is the C++ version of **Program 8-6** in your textbook.

```
#include <iostream>
using namespace std;

int main()
{
   // Constant for the array size.
   const int SIZE = 10;

   // Declare an array to hold test scores.
   int scores[SIZE] = { 87, 75, 98, 100, 82,
                        72, 88, 92, 60,  78 };

   // Declare a Boolean variable to act as a flag.
   bool found;

   // Declare a variable to use as a loop counter.
   int index;

   // The flag must initially be set to False.
   found = false;

   // Set the counter variable to 0.
   index = 0;

   // Step through the array searching for a
   // score equal to 100.
   while (found == false && index < SIZE)
   {
      if (scores[index] == 100)
        found = true;
```

```
        else
            index = index + 1;
    }

    // Display the search results.
    if (found)
        cout << "You earned 100 on test number "
             << (index + 1) << endl;
    else
        cout << "You did not earn 100 on any test." << endl;

    return 0;
}
```

Program Output

```
You earned 100 on test number 4
```

Searching a String Array

Program 8-4 demonstrates how to find a string in a string array. This is the C++ version of pseudocode Program 8-7 in the textbook.

Program 8-4

This is the C++ version of **Program 8-7** in your textbook.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   // Declare a constant for the array size.
   const int SIZE = 6;

   // Declare a String array initialized with values.
   string names[SIZE] = { "Ava Fischer", "Chris Rich",
                          "Gordon Pike", "Matt Hoyle",
                          "Rose Harrison", "Giovanni Ricci" };

   // Declare a variable to hold the search value.
   string searchValue;

   // Declare a Boolean variable to act as a flag.
   bool found;

   // Declare a counter variable for the array.
   int index;

   // The flag must initally be set to False.
   found = false;

   // Set the counter variable to 0.
   index = 0;

   // Get the string to search for.
```

```
   cout << "Enter a name to search for in the array." << endl;
   getline(cin, searchValue);

   // Step through the array searching for
   // the specified name.
   while (found == false && index < SIZE)
   {
      if (names[index] == searchValue)
        found = true;
     else
        index = index + 1;
   }

   // Display the search results.
   if (found)
      cout << "That name was found in element "
           << index << endl;
   else
      cout << "That name was not found in the array." << endl;

   return 0;
}
```

Program Output

```
Enter a name to search for in the array.
Matt Hoyle [Enter]
That name was found in element 3
```

Program Output

```
Enter a name to search for in the array.
Terry Thompson [Enter]
That name was not found in the array.
```

Passing an Array as an Argument to a Function

When passing an array as an argument to a function in C++, you should also pass a separate `int` argument indicating the array's size. The following code shows a function that has been written to accept an array as an argument:

```
void showArray(int array[], int size)
{
   for (int i = 0; i < size; i++)
      cout << array[i] << " ";
}
```

Notice that the parameter variable, `array`, is declared as an `int` array, without a size declarator. When we call this function we must pass an `int` array to it as an argument. Let's assume that `numbers` is the name of an `int` array, and `SIZE` is a constant that specifies the size of the array. Here is a statement that calls the `showArray` function, passing the `numbers` array and `SIZE` as arguments:

```
showArray(numbers, SIZE);
```

Program 8-5 gives a complete demonstration of passing an array to a function. This is the C++ version of pseudocode Program 8-13 in your textbook.

Program 8-5

This is the C++ version of **Program 8-13** in your textbook.

```
#include <iostream>
#include <string>
using namespace std;

// Function prototye
int getTotal(int[], int );

int main()
{
   // A constant for the array size
   const int SIZE = 5;

   // An array initialized with values
   int numbers[SIZE] = { 2, 4, 6, 8, 10 };

   // A variable to hold the sum of the elements
   int sum;

   // Get the sum of the elements.
   sum = getTotal(numbers, SIZE);

   // Display the sum of the array elements.
   cout << "The sum of the array elements is "
       << sum << endl;

   return 0;
}

// The getTotal function accepts an Integer array, and the
// array's size as arguments. It returns the total of the
// array elements.
int getTotal(int arr[], int size)
{
   // Loop counter
   int index;

   // Accumulator, initialized to 0
   int total = 0;

   // Calculate the total of the array elements.
   for (index = 0; index < size; index++)
   {
      total = total + arr[index];
```

```
    }

    // Return the total.
    return total;
}
```

Program Output

```
The sum of the array elements is 30
```

Two-Dimensional Arrays

Here is an example declaration of a two-dimensional array with three rows and four columns:

```
double scores[3][4];
```

The two sets of brackets in the data type indicate that the scores variable will reference a two-dimensional array. The numbers 3 and 4 are size declarators. The first size declarator specifies the number of rows, and the second size declarator specifies the number of columns. Notice that each size declarator is enclosed in its own set of brackets.

When processing the data in a two-dimensional array, each element has two subscripts: one for its row and another for its column. In the `scores` array, the elements in row 0 are referenced as follows:

```
scores[0][0]
scores[0][1]
scores[0][2]
scores[0][3]
```

The elements in row 1 are as follows:

```
scores[1][0]
scores[1][1]
scores[1][2]
scores[1][3]
```

And the elements in row 2 are as follows:

```
scores[2][0]
scores[2][1]
scores[2][2]
scores[2][3]
```

To access one of the elements in a two-dimensional array, you must use both subscripts. For example, the following statement stores the number 95 in `scores[2][1]`:

```
scores[2][1] = 95;
```

Programs that process two-dimensional arrays can do so with nested loops. For example, the following code prompts the user to enter a score, once for each element in the array:

```
const int ROWS = 3;
const int COLS = 4;
double scores[ROWS][COLS];
for (int row = 0; row < ROWS; row++)
{
   for (int col = 0; col < COLS; col++)
   {
      cout << "Enter a score." << endl;
      cin >> scores[row][col];
   }
}
```

And the following code displays all the elements in the `scores` array:

```
for (int row = 0; row < ROWS; row++)
{
   for (int col = 0; col < COLS; col++)
   {
      cout << scores[row][col] << endl;
   }
}
```

Program 8-6 shows a complete example. It declares an array with three rows and four columns, prompts the user for values to store in each element, and then displays the values in each element. This is the C++ example of pseudocode Program 8-16 in your textbook.

Program 8-5

This is the C++ version of **Program 8-16** in your textbook.

```
#include <iostream>
#include <string>
using namespace std;

// Function prototye
int getTotal(int[], int );

int main()
{
   // Create a 2D array
   const int ROWS = 2;
   const int COLS = 3;
   int values[ROWS][COLS];

   // Counter variables for rows and columns
   int row, col;

   // Get values to store in the array.
   for (row = 0; row <= ROWS - 1; row++)
   {
```

```
      for (col = 0; col <= COLS - 1; col++)
      {
        cout << "Enter a number." << endl;
        cin >> values[row][col];
      }
   }

   // Display the values in the array.
   cout << "Here are the values you entered." << endl;
   for (row = 0; row <= ROWS - 1; row++)
   {
      for (col = 0; col <= COLS - 1; col++)
      {
        cout << values[row][col] << endl;
      }
   }

   return 0;
}
```

Program Output

```
Enter a number.
1 [Enter]
Enter a number.
2 [Enter]
Enter a number.
3 [Enter]
Enter a number.
4 [Enter]
Enter a number.
5 [Enter]
Enter a number.
6 [Enter]
Here are the values you entered.
1
2
3
4
5
6
```

Arrays with Three or More Dimensions

C++ allows you to create arrays with virtually any number of dimensions. Here is an example of a three-dimensional array declaration:

```
double seats[3][5][8];
```

This array can be thought of as three sets of five rows, with each row containing eight elements. The array might be used to store the prices of seats in an auditorium, where there are eight seats in a row, five rows in a section, and a total of three sections.

Chapter 9

This chapter accompanies Chapter 9 of Starting Out with Programming Logic and Design, 5th Edition

Sorting and Searching Arrays

Chapter 9 discusses the following sorting algorithms:

- Bubble Sort
- Selection Sort
- Insertion Sort

The Binary Search algorithm is also discussed. The textbook chapter examines these algorithms in detail, and no new language features are introduced. For these reasons we will simply present the C++ code for the algorithms in this chapter. For more in-depth coverage of the logic involved, consult the textbook.

Bubble Sort

Program 9-1 is only a partial program. It shows the C++ version of pseudocode Program 9-1, which is the Bubble Sort algorithm.

Program 9-1

This is the C++ version of **Program 9-1** in your textbook.

```
// Note: This is not a complete program.
// bubbleSort function
void bubbleSort(int array[], int size)
{
   int maxElement;  // Marks the last element to compare
   int index;       // Index of an element to compare

   // The outer loop positions maxElement at the last element
   // to compare during each pass through the array. Initially
   // maxElement is the index of the last element in the array.
   // During each iteration, it is decreased by one.
   for (maxElement = size - 1; maxElement >= 0; maxElement--)
   {
      // The inner loop steps through the array, comparing
      // each element with its neighbor. All of the elements
      // from index 0 thrugh maxElement are involved in the
      // comparison. If two elements are out of order, they
      // are swapped.
      for (index = 0; index <= maxElement - 1; index++)
      {
         // Compare an element with its neighbor.
         if (array[index] > array[index + 1])
         {
            // Swap the two elements.
            swap(array[index], array[index+1]);
         }
      }
   }
```

```
}

// The swap function swaps the contents of the two
// arguments passed to it.
void swap(int &a, int &b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

Selection Sort

Program 9-2 is also a partial program. It shows the C++ version of the `selectionSort` pseudocode module that is shown in Program 9-5 in your textbook.

This is the C++ version of the `selectionSort` Module shown in **Program 9-5** in your textbook.

Program 9-2

```
//   Note: This is not a complete program.
//
//   The selectionSort function performs a selection sort on an
//   int array. The array is sorted in ascending order.

void selectionSort(int array[], int size)
{
   int startScan;   // Starting position of the scan
   int index;       // To hold a subscript value
   int minIndex;    // Element with smallest value in the scan
   int minValue;    // The smallest value found in the scan

   // The outer loop iterates once for each element in the
   // array. The startScan variable marks the position where
   // the scan should begin.
   for (startScan = 0; startScan < (size-1); startScan++)
   {
      // Assume the first element in the scannable area
      // is the smallest value.
      minIndex = startScan;
      minValue = array[startScan];

      // Scan the array, starting at the 2nd element in
      // the scannable area. We are looking for the smallest
      // value in the scannable area.
      for(index = startScan + 1; index < size; index++)
      {
         if (array[index] < minValue)
         {
            minValue = array[index];
            minIndex = index;
         }
      }

     // Swap the element with the smallest value
      // with the first element in the scannable area.
     swap(array[minIndex], array[startScan]);
```

```
   }
}

// The swap function swaps the contents of the two
// arguments passed to it.
void swap(int &a, int &b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

Insertion Sort

Program 9-3 is also a partial program. It shows the C++ version of the `insertionSort` pseudocode module that is shown in Program 9-6 in your textbook.

This is the C++ version of the `insertionSort` Module shown in **Program 9-6** in your textbook.

Program 9-3

```
// Note: This is not a complete program.
//
// The insertionSort function performs an insertion sort on
// an int array. The array is sorted in ascending order.

void insertionSort(int array[], int size)
{
   int unsortedValue;  // The first unsorted value
   int scan;           // Used to scan the array

   // The outer loop steps the index variable through
   // each subscript in the array, starting at 1. This
   // is because element 0 is considered already sorted.
   for (int index = 1; index < size; index++)
   {
      // The first element outside the sorted subset is
      // array[index]. Store the value of this element
      // in unsortedValue.
      unsortedValue = array[index];

      // Start scan at the subscript of the first element
      // outside the sorted subset.
      scan = index;

      // Move the first element outside the sorted subset
      // into its proper position within the sorted subset.
      while (scan > 0 && array[scan-1] > unsortedValue)
      {
         array[scan] = array[scan - 1];
```

```
            scan--;
         }

         // Insert the unsorted value in its proper position
         // within the sorted subset.
        array[scan] = unsortedValue;
     }
 }
```

Binary Search

Program 9-4 is also a partial program. It shows the C++ version of the `binarySearch` pseudocode module that is shown in Program 9-7 in your textbook.

Program 9-4

This is the C++ version of the `binarySearch` Module shown in **Program 9-7** in your textbook.

```
//   Note: This is not a complete program
//
//   The binarySearch function performs a binar
//   string array. The array is searched for th
//   to value. If the string is found, its array subscript
//   is returned. Otherwise, -1 is returned indicating the
//   value was not found in the array.

int binarySearch(string array[], string value, int size)
{
   int first;        // First array element
   int last;         // Last array element
   int middle;       // Midpoint of search
   int position;     // Position of search value
   bool found;       // Flag

   // Set the inital values.
   first = 0;
   last = size - 1;
   position = -1;
   found = false;

   // Search for the value.
   while (!found && first <= last)
   {
      // Calculate midpoint
      middle = (first + last) / 2;

      // If value is found at midpoint...
      if (array[middle] == value)
      {
         found = true;
         position = middle;
      }
      // else if value is in lower half...
      else if (array[middle] > value)
         last = middle - 1;
      // else if value is in upper half....
```

```
         else
            first = middle + 1;
      }

      // Return the position of the item, or -1
      // if it was not found.
      return position;
   }
```

Chapter 10

This chapter accompanies Chapter 10 of Starting Out with Programming Logic and Design, 5th Edition

Files

Opening a File and Writing Data to It

To work with files in a C++ program you first write the following `#include` directive at the top of your program:

```
#include <fstream>
```

Then, in the function where you wish to open a file and write data to it you will declare an `ofstream` object. Here is an example of a statement that declares an `ofstream` object:

```
ofstream outputFile;
```

This statement declares an `ofstream` object named `outputFile`. Next, you use the `ofstream` object to open a file. Here is an example:

```
outputFile.open("StudentData.txt");
```

After this statement has executed, we will be able to use the `ofstream` object that is named `outputFile` to write data to the `StudentData.txt` file. You can think of it this way: In memory we have an `ofstream` object that we refer to in our code as `outputFile`. That object is connected to a file on the disk named `StudentData.txt`. If we want to write data to the `StudentData.txt` file, we will use the `ofstream` object. (Note that if the `StudentData.txt` file does not exist, this statement will create the file. If the file already exists, its contents will be erased. Either way, after this statement executes an empty file named `StudentData.txt` will exist on the disk.)

Once you have created an `ofstream` object and opened a file, you can write data to the file using the stream insertion operator (<<). You already know how to use the << operator with `cout` to display data on the screen. It is used the same way with an `ofstream` object to write data to a file. For example, assuming that `outputFile` is an `ofstream` object, the following statement writes the string `"Jim"` to the file:

```
outputFile << "Jim" << endl;
```

Assuming that `payRate` is a variable, the following statement writes the value of the `payRate` variable to the file:

```
outputFile << payRate << endl;
```

Closing a File

When the program is finished writing data to the file, it must close the file. Assuming that `outputFile` is the name of an `ofstream` object, here is an example of how to call the `close` function to close the file:

```
outputFile.close();
```

Once a file is closed, the connection between it and the `ofstream` object is removed. In order to perform further operations on the file, it must be opened again.

Program 10-1 demonstrates how to create an `ofstream` object (and open a file for output), write some data to the file, and close the file. This is the C++ version of pseudocode Program 10-1 in your textbook.

Program 10-1

This is the C++ version of **Program 10-1** in your textbook.

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main()
6 {
7    // Declare an ofstream object named myFile and open a
8    // file named philosphers.txt.
9    ofstream myFile;
10   myFile.open("philosophers.txt");
11
12   // Write the names of three philosophers to the file.
13   myFile << "John Locke" << endl;
14   myFile << "David Hume" << endl;
15   myFile << "Edmund Burke" << endl;
16
17   // Close the file.
18   myFile.close();
19   return 0;
20 }
```

When this program executes, line 9 declares an `ofstream` object named `myFile`, and line 10 uses that object to open a file named `philosophers.txt` on the disk. Lines 13 through 15 write the strings "John Locke", "David Hume", and "Edmund Burke" to the file. Line 18 closes the file.

Program 10-2 shows another example. This program opens a file named numbers.txt, and uses a loop to write 10 random numbers to the file.

Program 10-2

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
   // Constant for the maximum number of numbers.
   const int MAX_NUMS = 10;

   // Variables
   int counter, number;

   // Declare an ofstream object named myFile and open a
   // file named numbers.txt.
   ofstream myFile;
   myFile.open("numbers.txt");

   // Get the system time.
   unsigned seed = time(0);

   // Seed the random number generator.
   srand(seed);

   // The following loop writes random numbers
   // numbers to the file.
   for (counter = 1; counter <= MAX_NUMS; counter++)
   {
      // Generate a random number.
      number = 1 + rand() % 100;

      // Write the number to the file.
      myFile << number << endl;
   }

   // Close the file.
   myFile.close();
   return 0;
}
```

Opening a File and Reading Data From It

Now we will discuss first write how you can read data from a file in C++. The following `#include` directive is needed in your program:

```
#include <fstream>
```

Then, in the function where you wish to open a file and read data from it, you will declare an `ifstream` object. Here is an example of a statement that declares an `ifstream` object:

```
ifstream inputFile;
```

This statement declares an `ifstream` object named `inputFile`. Next, you use the `ifstream` object to open a file. Here is an example:

```
inputFile.open("numbers.txt");
```

After this statement has executed, we will be able to use the `ifstream` object that is named `inputFile` to read data from the `numbers.txt` file. You can think of it this way: In memory we have an `ifstream` object that we refer to in our code as `inputFile`. That object is connected to a file on the disk named `numbers.txt`. If we want to read data from the `numbers.txt` file, we will use the `ifstream` object.

Once you have created an `ifstream` object and opened a file, you can read an item of data from the file using the stream extraction operator (>>). You already know how to use the >> operator with `cin` to read input from the keyboard. It is used the same way with an `ifstream` object to read data from a file. For example, assuming that `inputFile` is an `ifstream` object, the following statement reads a piece of data from the file, and stores that piece of data in the `value` variable:

```
inputFile >> value;
```

Closing the File

When the program is finished reading data from the file, it must close the file. Assuming that `inputFile` is the name of an `ifstream` object, here is an example of how to call the `close` function to close the file:

```
inputFile.close();
```

Once a file is closed, the connection between it and the `ifstream` object is removed. In order to perform further operations on the file, it must be opened again.

Previously, in Program 10-2, you saw an example that created a file and wrote 10 random numbers to the file. Program 10-3 demonstrates how to read the list of numbers from the file and display them on the screen.

Program 10-3

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   // Constant for the maximum number of numbers.
   const int MAX_NUMS = 10;

   // Variables
   int counter, number;

   // Declare an ifstream object named myFile and open a
   // file named numbers.txt, for reading.
   ifstream myFile;
   myFile.open("numbers.txt");

   // The following loop reads 10 numbers from the file
   // and displays them.
   for (counter = 1; counter <= MAX_NUMS; counter++)
   {
      // Read a number from the file.
      myFile >> number;

      // Display the number.
      cout << number << endl;
   }

   // Close the file.
   myFile.close();
   return 0;
}
```

Program Output (These numbers are random. Yours will be different.)

```
54
40
37
15
92
57
20
63
66
48
```

Using `getline` To Read Strings

You've already learned that when you want to read a string that contains multiple words, separated by spaces, from the keyboard, you must use the `getline` function. The same is true when you want to read a string from a file, and the string contains spaces. The `getline` function can read an entire line of input from a file, including embedded spaces, storing the input in a `string` variable.

Assume that `inputFile` is the name of an `ifstream` object, and that you have opened a file. Also assume that `line` is the name of a `string` variable. The following statement shows how you can use the `getline` function to read a line of input from the file, storing that line of input in the `line` variable:

```
getline(inputFile, line);
```

Program 10-2 shows an example. This program opens the `philosophers.txt` file that was created by Program 10-1. This is the C++ version of pseudocode Program 10-2 in your textbook.

Program 10-4

This is the C++ version of **Program 10-2** in your textbook.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
   // Declare three variables that will hold the values
    // read from the file.
    string name1, name2, name3;

   // Declare an ifstream object named myFile and open a
   // file named philosphers.txt.
   ifstream myFile;
   myFile.open("philosophers.txt");

   // Read the names of three philosophers from the file
   // into the variables.
   getline(myFile, name1);
   getline(myFile, name2);
   getline(myFile, name3);

   // Display the names that were read.
   cout << "Here are the names of three philosophers:" << endl;
   cout << name1 << endl;
   cout << name2 << endl;
   cout << name3 << endl;

   // Close the file.
   myFile.close();
```

```
   return 0;
}
```

Program Output

```
Here are the names of three philosophers:
John Locke
David Hume
Edmund Burke
```

Appending Data to an Existing File

When you use an ofstream object to open a file, and that file already exists, its contents will be erased. Sometimes you want to open an existing file without erasing its current contents, and write new data to the end of the file. This is called *appending* data to the file.

In C++ you declare an `fstream` object when you want to append data to a file's existing contents. Here is an example of how you would declare an `fstream` object named `myFile`, and then use that object to open a file named `friends.txt`. The file's existing contents will not be erased.

```
fstream myFile;
myFile.open("friends.txt", ios::app);
```

Notice that two arguments are passed to the `open` function :

- The name of the file. In this case, the file's name is `friends.txt`.
- The special value `ios::app`. This specifies that any data that is written to the file should be appended to the file's existing contents.

Using Loops to Process Files

You've already seen some examples of how a loop can be used to process a file's contents (Programs 10-2 and 10-3 in this booklet). Program 10-5 shows another example. It demonstrates how a loop can be used to collect items of data to be stored in a file. This is the C++ version of pseudocode Program 10-3 in your textbook.

Program 10-5

This is the C++ version of **Program 10-3** in your textbook.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   // Variable to hold the number of days.
   int numDays;

   // Counter variable for the loop.
```

```
   int counter;

   // Variable to hold an amount of sales.
   double sales;

   // Declare an output file.
   ofstream salesFile;

   // Get the number of days.
   cout << "For how many days do you have sales?" << endl;
   cin >> numDays;

   // Open a file named sales.txt.
   salesFile.open("sales.txt");

   // Get the amount of sales for each day and write
   // it to the file.
   for (counter = 1; counter <= numDays; counter++)
   {
      // Get the sales for a day.
      cout << "Enter the sales for day #"
          << counter << endl;
      cin >> sales;

      // Write the amount to the file.
      salesFile << sales << endl;
   }

   // Close the file.
   salesFile.close();
   cout << "Data written to sales.txt." << endl;
   return 0;
}
```

Program Output

For how many days do you have sales?
5 [*Enter*]
Enter the sales for day #1
1000 [*Enter*]
Enter the sales for day #2
2000 [*Enter*]
Enter the sales for day #3
3000 [*Enter*]
Enter the sales for day #4
4000 [*Enter*]
Enter the sales for day #5

```
5000 [Enter]
Data written to sales.txt.
```

Detecting the End of a File

Sometimes you need to read a file's contents, and you do not know the number of items that are stored in the file. You can open the file, and then use a loop to repeatedly read an item from the file and display it. However, an error will occur if the program attempts to read beyond the end of the file. The program needs some way of knowing when the end of the file has been reached so it will not try to read beyond it. Fortunately, the >> operator not only reads data from a file, but also returns a true or false value indicating whether the data was successfully read or not. If the operator returns true, then a value was successfully read. If the operator returns false, it means that no value was read from the file.

Program 10-6 demonstrates how to use this technique. This is the C++ version of pseudocode Program 10-4 in your textbook. The program opens the `sales.txt` file that was created by Program 10-5 (in this booklet). It reads and displays each item of data in the file.

Program 10-4

This is the C++ version of **Program 10-4** in your textbook.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   // Declare an input file object.
   ifstream salesFile;

   // Declare a variable to hold a sales amount
   // that is read from the file.
   double sales;

   // Open the sales.txt file.
   salesFile.open("sales.txt");

   cout << "Here are the sales amounts:" << endl;

   // Read all of the items in the file
   // and display them.
   while (salesFile >> sales)
   {
      cout << sales << endl;
   }

   // Close the file.
   salesFile.close();
   return 0;
}
```

Program Output

```
Here are the sales amounts:
1000
2000
3000
4000
5000
```

Chapter 11

This chapter accompanies Chapter 11 of Starting Out with Programming Logic and Design, 5th Edition

Menu-Driven Programs

Chapter 11 in your textbook discusses menu-driven programs. A menu-driven program presents a list of operations that the user may select from (the menu), and then performs the operation that the user selected. There are no new language features introduced in the chapter, so here we will simply show you a C++ program that is menu-driven. Program 11-1 is the C++ version of the pseudocode Program 11-3.

Program 11-1

This is the C++ version of **Program 11-3** in your textbook.

```
#include <iostream>
using namespace std;

int main()
{
   // Declare a variable to hold the
   // user's menu selection.
   int menuSelection;

   // Declare variables to hold the units
   // of measurement.
   double inches, centimeters, feet, meters,
          miles, kilometers;

   // Display the menu.
   cout << "1. Convert inches to centimeters." << endl;
   cout << "2. Convert feet to meters." << endl;
   cout << "3. Convert miles to kilometers." << endl;
   cout << endl;

   // Prompt the user for a selection
   cout << "Enter your selection." << endl;
   cin >> menuSelection;

   // Validate the menu selection.
   while (menuSelection < 1 || menuSelection > 3)
   {
      cout << "That is an invalid selection." << endl;
```

```
      cout << "Enter 1, 2, or 3." << endl;
      cin >> menuSelection;
   }

   // Perform the selected operation.
   switch(menuSelection)
   {
      case 1:
        // Convert inches to centimeters.
        cout << "Enter the number of inches." << endl;
        cin >> inches;
        centimeters = inches * 2.54;
        cout << "That is equal to " << centimeters
             << " centimeters." << endl;
        break;

      case 2:
        // Convert feet to meters.
        cout << "Enter the number of feet." << endl;
        cin >> feet;
        meters = feet * 0.3048;
        cout << "That is equal to " << meters
             << " meters." << endl;
        break;

      case 3:
        // Convert miles to kilometers.
        cout << "Enter the number of miles." << endl;
        cin >> miles;
        kilometers = miles * 1.609;
        cout << "That is equal to " << kilometers
             << " kilometers." << endl;
        break;
   }
   return 0;
}
```

Program Output

```
1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.
```

```
Enter your selection.
1 [Enter]
Enter the number of inches.
10 [Enter]
That is equal to 25.4 centimeters.
```

Program Output

```
1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.

Enter your selection.
2 [Enter]
Enter the number of feet.
10 [Enter]
That is equal to 3.048 meters.
```

Program Output

```
1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.

Enter your selection.
4 [Enter]
That is an invalid selection.
Enter 1, 2, or 3.
3 [Enter]
Enter the number of miles.
10 [Enter]
That is equal to 16.09 kilometers.
```

Chapter 12

This chapter accompanies Chapter 12 of Starting Out with Programming Logic and Design, 5th Edition

Text Processing

Chapter 12 in your textbook discusses programming techniques for working with the individual characters in a string. C++ allows you to work with the individual characters in a string using subscript notation, as described in the book. For example, Program 12-1 shows the C++ version of pseudocode Program 12-1 in the textbook.

Program 12-1

This is the C++ version of **Program 12-1** in your textbook.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   // Declare and initialize a string.
   string name = "Jacob";

   // Use subscript notation to display the
   // individual characters in the string.
   cout << name[0] << endl;
   cout << name[1] << endl;
   cout << name[2] << endl;
   cout << name[3] << endl;
   cout << name[4] << endl;
   return 0;
}
```

Program Output

```
J
a
c
o
b
```

Variables of the string type have a built-in `length()` function that returns the number of characters in the string. Program 12-2, which is the C++ version of pseudocode Program 12-1 in your textbook, demonstrates the `length()` function. This program uses a loop to step through all of the characters in a `string`.

Program 12-2

This is the C++ version of **Program 12-2** in your textbook.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   // Declare and initialize a string.
   string name = "Jacob";

   // Declare a variable to step through the string.
   int index;

   // Display the individual characters in the string.
   for (index = 0; index < name.length(); index++)
      cout << name[index] << endl;

   return 0;
}
```

Program Output

```
J
a
c
o
b
```

Program 12-3 shows how subscript notation can be used to change a specific character in a string. This is the C++ version of pseudocode Program 12-3 in your textbook.

Program 12-3

This is the C++ version of **Program 12-3** in your textbook.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   // Declare a string to hold input.
   string input;

   // Declare a variable to step through the string.
   int index;
```

```
12
13     // Prompt the user to enter a sentence.
14     cout << "Enter a sentence." << endl;
15     getline(cin, input);
16
17     // Change each 't' to a 'd'.
18     for (index = 0; index < input.length(); index++)
19     {
20        if (input[index] == 't')
21        {
22          input[index] = 'd';
23        }
24     }
25
26     // Display the modified string.
27     cout << input << endl;
28
29     return 0;
30  }
```

Program Output

```
Enter a sentence.
Look at that kitty cat! [Enter]
Look ad dhad kiddy cad!
```

Character Literals in C++

You probably noticed that in line 20 we are using single quotes around the character literal `'t'`, and in line 22 we are using single quotes around the character literal `'d'`. In C++, there is a difference between a string literal and a character literal. String literals are encloses in double quotes, and character literals are enclosed in single quotes. If you are writing a relational expression, using one of the relational operators such as `==`, and the operand on the left side is a character, then the operand on the right side must also be a character or an error will occur. In line 20, the expression `input[index]` returns a character, so we have to compare it to the character literal `'t'`. If we mistakenly compare it to the string literal `"t"` an error will occur. In line 22 we are assigning a value to the character at `input[index]`, so the value on the right side of the = operator must be a character literal. If we mistakenly assign the string literal `"d"`, an error will occur.

Character Testing Functions

C++ provides functions that are similar to the character testing library functions shown in Table 12-2 in your textbook. The C++ functions that are similar to those functions are shown here, in Table 12-1. (To use these functions, be sure to write the `#include <cctype>` directive in your program.)

Table 12-1 Character Testing Functions

Function	Description
`isalnum()`	Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise.
`isalpha()`	Returns true if the string contains only alphabetic letters, and is at least one character in length. Returns false otherwise.
`isdigit()`	Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise.
`islower()`	Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise.
`isspace()`	Returns true if the string contains only whitespace characters, and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (\n), and tabs (\t).
`isupper()`	Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise.

Program 12-4 demonstrates how the `isupper()` function is used. This program is the C++ version of Program 12-4 in your textbook.

Program 12-4

This is the C++ version of **Program 12-4** in your textbook.

```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main()
{
   // Declare a string to hold input.
   string str;

   // Declare a variable to step through the string.
   int index;

   // Declare an accumulator variable to keep count
   // of the number of uppercase letters.
```

```
   int upperCaseCount = 0;

   // Prompt the user to enter a sentence.
   cout << "Enter a sentence: " << endl;
   getline(cin, str);

   // Count the number of uppercase letters.
   for (index = 0; index < str.length(); index++)
   {
      if (isupper(str[index]))
      {
         upperCaseCount = upperCaseCount + 1;
      }
   }

   // Display the number of uppercase characters.
   cout << "That string has " << upperCaseCount
        << " uppercase letters." << endl;

   return 0;
}
```

Program Output

```
Enter a sentence.
Mr. Jones will arrive TODAY! [Enter]
That string has 7 uppercase letters.
```

Inserting and Deleting Characters in a `string`

There are built-in `string` functions for inserting and deleting characters in a string. These functions are similar to the library modules that are shown in Table 12-3 in your textbook. The `string` functions that are similar to those functions are shown here, in Table 12-2.

Table 12-2 `string` Insertion and Deletion Functions

Function	Description
`stringName.insert(position, string2)`	*`stringName`* is the name of a `string` variable, *`position`* is an `int`, and *`string2`* is a `string`. The function inserts *`string2`* into the `string` variable, beginning at *`position`*.
`stringName.erase(start, numChars)`	*`stringName`* is the name of a `string` variable, *`start`* is an `int`, and *`numChars`* is an `int`. The function erases the number of characters specified by *`numChars`*, beginning

	at the position specified by*start*.

Here is an example of how we might use the `insert` function:

```
string str = "New City";
str.insert(4, "York ");
cout << str << endl;
```

The second statement inserts the string `"York "` into the `string`, beginning at position 4. The characters that are currently in the `string` beginning at position 4 are moved to the right. In memory, the `string` is automatically expanded in size to accommodate the inserted characters. If these statements were a complete program and we ran it, we would see `New York City` displayed on the screen.

Here is an example of how we might use the `erase` function:

```
string str = "I ate 1000 blueberries!";
str.erase(8, 2);
cout << str << endl;
```

The second statement deletes 2 characters, beginning at position 8in the `string`. The characters that previously appeared beginning at position 10 are shifted left to occupy the space left by the two deleted characters. If these statements were a complete program and we ran it, we would see `I ate 10 blueberries!` displayed on the screen.

Chapter 13

This chapter accompanies Chapter 13 of Starting Out with Programming Logic and Design, 5th Edition

Recursion

A C++ function can call itself recursively, allowing you to design algorithms that recursively solve a problem. Chapter 13 in your textbook describes recursion in detail, discusses problem solving with recursion, and provides several pseudocode examples. Other than the technique of a function recursively calling itself, no new language features are introduced. In this chapter we will present C++ versions of two of the pseudocode programs that are shown in the textbook. Both of these programs work exactly as the algorithms are described in the textbook. Program 13-1 is the C++ version of pseudocode Program 13-2.

Program 13-1

This is the C++ version of **Program 13-2** in your textbook.

```
#include <iostream>
using namespace std;

// Function prototype
void message(int);

int main()
{
   // By passing the argument 5 to the message function
   // we are telling it to display the message 5 times.
   message(5);

   return 0;
}

void message(int n)
{
   if (n > 0)
   {
      cout << "This is a recursive function." << endl;
      message(n - 1);
   }
}
```

Program Output

```
This is a recursive function.
This is a recursive function.
This is a recursive function.
```

```
This is a recursive function.
This is a recursive function.
```

Next, Program 13-2 is the C++ version of pseudocode Program 13-3. This program recursively calculates the factorial of a number.

Program 13-2

This is the C++ version of **Program 13-3** in your textbook.

```
#include <iostream>
#include <string>
using namespace std;

// Function prototype
int factorial(int);

int main()
{
   int number;       // To hold a number entered by the user
   int numFactorial; // To hold the factorial of the number

   // Get a number from the user.
   cout << "Enter a nonnegative integer." << endl;
   cin >> number;

   // Get the factorial of the number.
   numFactorial = factorial(number);

   // Display the factorial of the number.
   cout << "The factorial of " << number
        << " is " << numFactorial << endl;

   return 0;
}

//   The factorial function uses recursion to calculate
//   the factorial of its argument, which is assumed
//   to be a nonnegative number.
int factorial(int n)
{
   if (n == 0)
      return 1;    // Base case
   else
      return n * factorial(n - 1);
}
```

Program Output

```
Enter a non-negative integer.
```
7 [*Enter*]
```
The factorial of 7 is 5040
```

Chapter 14

This chapter accompanies Chapter 14 of Starting Out with Programming Logic and Design, 5th Edition

Object-Oriented Programming

C++ is a powerful object-oriented language. An object is an entity that exists in the computer's memory while the program is running. An object contains data and has the ability to perform operations on its data. An object's data is commonly referred to as the object's fields, and the operations that the object performs are the object's methods. In C++, an object's methods are commonly called *member functions*.

In the object-oriented way of programming, objects are used to perform many of the program's tasks. For example, in C++, `string` variables are actually objects. In addition, `cout` and `cin` are objects. You have also used `ofstream`, `ifstream`, and `fstream` objects to work with files.

In addition to the objects that are provided by the C++ language, you can create objects of your own design. The first step is to write a class. A class is like a blueprint. It is a declaration that specifies the methods for a particular type of object. When the program needs an object of that type, it creates an instance of the class. (An object is an instance of a class.)

Here is the general format of a class declaration in C++:

```
class ClassName
{
    Field declarations and member function definitions go here…
};
```

Notice that a class declaration in C++ ends with a semicolon.

Chapter 14 in your textbook steps through the design of a `CellPhone` class. The following `CellPhone` class is the C++ version of Class Listing 14-3. Notice that line 3 reads `private:`, and line 9 reads `public:`. These are *access specifiers*, and they control how class fields and member functions can be accessed by code outside the class. All of the field declarations that appear after the `private:` access specifier in line 3 are private. They can be accessed only by code inside the class. All of the member functions that appear after the `public:` access specifer in line 9 are public, and can be called by code outside the class.

Program 14-1 also has a `main` function to demonstrate the class, like that shown in pseudocode Program 14-3 in your textbook.

Program 14-1

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
```

This is the C++ version of **Class Listing 14-3** and **Program 14-1** in your textbook.

```
class CellPhone
{
private:
   // Field declarations
   string manufacturer;
   string modelNumber;
   double retailPrice;

public:
   // Member functions
   void setManufacturer(string manufact)
   {
      manufacturer = manufact;
   }

   void setModelNumber(string modNum)
   {
      modelNumber = modNum;
   }

   void setRetailPrice(double retail)
   {
      retailPrice = retail;
   }

   string getManufacturer()
   {
      return manufacturer;
   }

   string getModelNumber()
   {
      return modelNumber;
   }

   double getRetailPrice()
   {
      return retailPrice;
   }
};
```

Notice that the C++ class declaration ends with a semicolon!

```
46 int main()
47 {
48    // Declare a variable that can reference
49    // a CellPhone object.
50    CellPhone myPhone;
51
52    // Store values in the object's fields.
53    myPhone.setManufacturer("Motorola");
54    myPhone.setModelNumber("M1000");
55    myPhone.setRetailPrice(199.99);
56
57    // Display the values stored in the fields.
58    cout << "The manufacturer is "
59         << myPhone.getManufacturer() << endl;
60    cout << "The model number is "
61         << myPhone.getModelNumber() << endl;
62    cout << "The retail price is "
63         << myPhone.getRetailPrice() << endl;
64
65    return 0;
66 }
```

Program Output

```
The manufacturer is Motorola
The model number is M1000
The retail price is 199.99
```

Inside the `main` function, line 50 creates an instance of the `CellPhone` class in memory and assigns it to the `myPhone` variable. We say that the object is referenced by the `myPhone` variable. (Notice that C++ does not require the `New` keyword, as discussed in your textbook.) Lines 53 through 55 call the object's `setManufacturer`, `setModelNumber`, and `setRetailPrice` member functions, passing arguments to each.

Constructors

A class constructor in C++ is a member function that has the same name as the class. The following is a version of the `CellPhone` class that has a constructor. This is the C++ version of Class Listing 14-4 in your textbook, combined with pseudocode Program 14-2 from your textbook. The constructor appears in lines 9 through 14.

Program 14-2

```
 1 #include <iostream>
 2 #include <string>
```

```
using namespace std;

class CellPhone
{
private:
   // Field declarations
   string manufacturer;
   string modelNumber;
   double retailPrice;

public:
   // Constructor
   CellPhone(string manufact, string modNum, double retail)
   {
      manufacturer = manufact;
      modelNumber = modNum;
      retailPrice = retail;
   }

   // Member functions
   void setManufacturer(string manufact)
   {
      manufacturer = manufact;
   }

   void setModelNumber(string modNum)
   {
      modelNumber = modNum;
   }

   void setRetailPrice(double retail)
   {
      retailPrice = retail;
   }

   string getManufacturer()
   {
      return manufacturer;
   }

   string getModelNumber()
   {
      return modelNumber;
   }

   double getRetailPrice()
   {
```

This is the C++ version of **Class Listing 14-4** and **Program 14-2** in your textbook.

```
        return retailPrice;
    }
};

int main()
{
    // Create a CellPhone object and initialize its
    // fields with values passed to the constructor.
    CellPhone myPhone("Motorola", "M1000", 199.99);

    // Display the values stored in the fields.
    cout << "The manufacturer is "
          << myPhone.getManufacturer() << endl;
    cout << "The model number is "
        << myPhone.getModelNumber() << endl;
    cout << "The retail price is "
         << myPhone.getRetailPrice() << endl;

    return 0;
}
```

Program Output

```
The manufacturer is Motorola
The model number is M1000
The retail price is 199.99
```

Inheritance

The inheritance example discussed in your textbook starts with the `GradedActivity` class (see Class Listing 14-8), which is used as a superclass. The `FinalExam` class is then used as a subclass (see Class Listing 14-9) The C++ versions of these classes are shown in Program 14-3. This program also has a `main` function that demonstrates how the inheritance works.

Notice that in line 47, in the class header, the `: public GradedActivity` clause specifies that the `FinalExam` class extends the `GradedActivity` class.

Program 14-3

This is the C++ version of **Class Listing 14-8**, **Class Listing 14-9**, and **Program 14-3** in your textbook.

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class GradedActivity
6 {
7 private:
8    // The score field holds a numeric score.
9    double score;
```

```

public:
   // Mutator
   void setScore(double s)
   {
      score = s;
   }

   // Accessor
   double getScore()
   {
      return score;
   }

   // getGrade function
   string getGrade()
   {
      // Local variable to hold a grade.
      string grade;

      // Determine the grade.
      if (score >= 90)
         grade = "A";
      else if (score >= 80)
         grade = "B";
      else if (score >= 70)
         grade = "C";
      else if (score >= 60)
         grade = "D";
      else
         grade = "F";

      // Return the grade.
      return grade;
   }
};

class FinalExam : public GradedActivity
{
private:
   // Fields
   int numQuestions;
   double pointsEach;
   int numMissed;

public:
   // The constructor sets the number of
```

```
   // questions on the exam and the number
   // of questions missed.
   FinalExam(int questions, int missed)
   {
      // Local variable to hold the numeric score.
      double numericScore;

      // the numQuestions and numMissed fields.
      numQuestions = questions;
      numMissed = missed;

      // Calculate the points for each question
      // and the numeric score for this exam.
      pointsEach = 100.0 / questions;
      numericScore = 100.0 - (missed * pointsEach);

      // Call the inherited setScore function to
      // set the numeric score.
      setScore(numericScore);
   }

   // Accessors
   double getPointsEach()
   {
      return pointsEach;
   }

   int getNumMissed()
   {
      return numMissed;
   }
};

int main()
{
   // Variables to hold user input.
   int questions, missed;

   // Prompt the user for the number of questions
   // on the exam.
   cout << "Enter the number of questions on the exam."
        << endl;
   cin >> questions;

   // Prompt the user for the number of questions
   // missed by the student.
   cout << "Enter the number of questions that the "
```

```
          << "student missed." << endl;
     cin >> missed;

     // Create a FinalExam object.
     FinalExam exam(questions, missed);

     // Display the test results.
     cout << "Each question on the exam counts "
          << exam.getPointsEach() << " points." << endl;
     cout << "The exam score is "
         << exam.getScore() << endl;
     cout << "The exam grade is "
         << exam.getGrade() << endl;
     return 0;
}
```

Program Output

```
Enter the number of questions on the exam.
20 [Enter]
Enter the number of questions that the student missed.
3 [Enter]
Each question on the exam counts 5 points.
The exam score is 85
The exam grade is B
```

Polymorphism

Your textbook presents a polymorphism demonstration that uses the `Animal` class (Class Listing 14-10) as a superclass, and the `Dog` class (Class Listing 14-11) and `Cat` class (Class Listing 14-12) as subclasses of `Animal`. The C++ versions of those classes are shown here, in Program 14-4. The `main` function and the `showAnimalInfo` functions are the C++ equivalent of Program 14-6 in your textbook.

Notice that the key word `virtual` appears in the function headers for the `showSpecies` and `makeSound` functions (lines 9, 15, 25, 31, 40, and 46). The `virtual` key word tells the compiler to expect the function to be redefined in a subclass.

Also, notice that in line 82, the `showAnimalInfo` function accepts an Animal object by reference. In C++, polymorphic behavior is possible only when an object is passed by reference .

Program 14-4

```
 1 #include <iostream>
 2 #include <string>
 3 using namespace std;
 4
 5  class Animal
 6 {
 7 public:
 8    // showSpecies function
```

This is the C++ version of **Class Listing 14-10**, **Class Listing 14-11**, **Class Listing 14-12**, and **Program 14-6** in your textbook.

```
   virtual void showSpecies()
   {
      cout << "I'm just a regular animal." << endl;
   }

   // makeSound function
   virtual void makeSound()
   {
      cout << "Grrrrrr" << endl;
   }
};

class Dog : public Animal
{
public:
   // showSpecies function
   virtual void showSpecies()
   {
      cout << "I'm a dog." << endl;
   }

   // makeSound function
   virtual void makeSound()
   {
      cout << "Woof! Woof!" << endl;
   }
};

class Cat : public Animal
{
   // showSpecies function
   virtual void showSpecies()
   {
      cout << "I'm a cat." << endl;
   }

   // makeSound function
   virtual void makeSound()
   {
      cout << "Meow" << endl;
   }
};

// Function prototype
void showAnimalInfo(Animal &creature);

int main()
```

```
{
   // Declare three class variables.
   Animal myAnimal;
   Dog myDog;
   Cat myCat;

   // Show info about an animal.
   cout << "Here is info about an animal." << endl;
   showAnimalInfo(myAnimal);
   cout << endl;

   // Show info about a dog.
   cout << "Here is info about a dog." << endl;
   showAnimalInfo(myDog);
   cout << endl;

   // Show info about a cat.
   cout << "Here is info about a cat." << endl;
   showAnimalInfo(myCat);

   return 0;
}

// The showAnimalInfo function accepts an Animal
// object as an argument and displays information
// about it.
void showAnimalInfo(Animal &creature)
{
   creature.showSpecies();
   creature.makeSound();
}
```

Program Output

```
Here is info about an animal.
I am just a regular animal.
Grrrrrrr

Here is info about a dog.
I am a dog.
Woof! Woof!

Here is info about a cat.
I am a cat.
Meow
```

Chapter 15

This chapter accompanies Chapter 15 of Starting Out with Programming Logic and Design, 5th Edition

GUI Applications and Event-Driven Programming

Visual C++ Tutorial

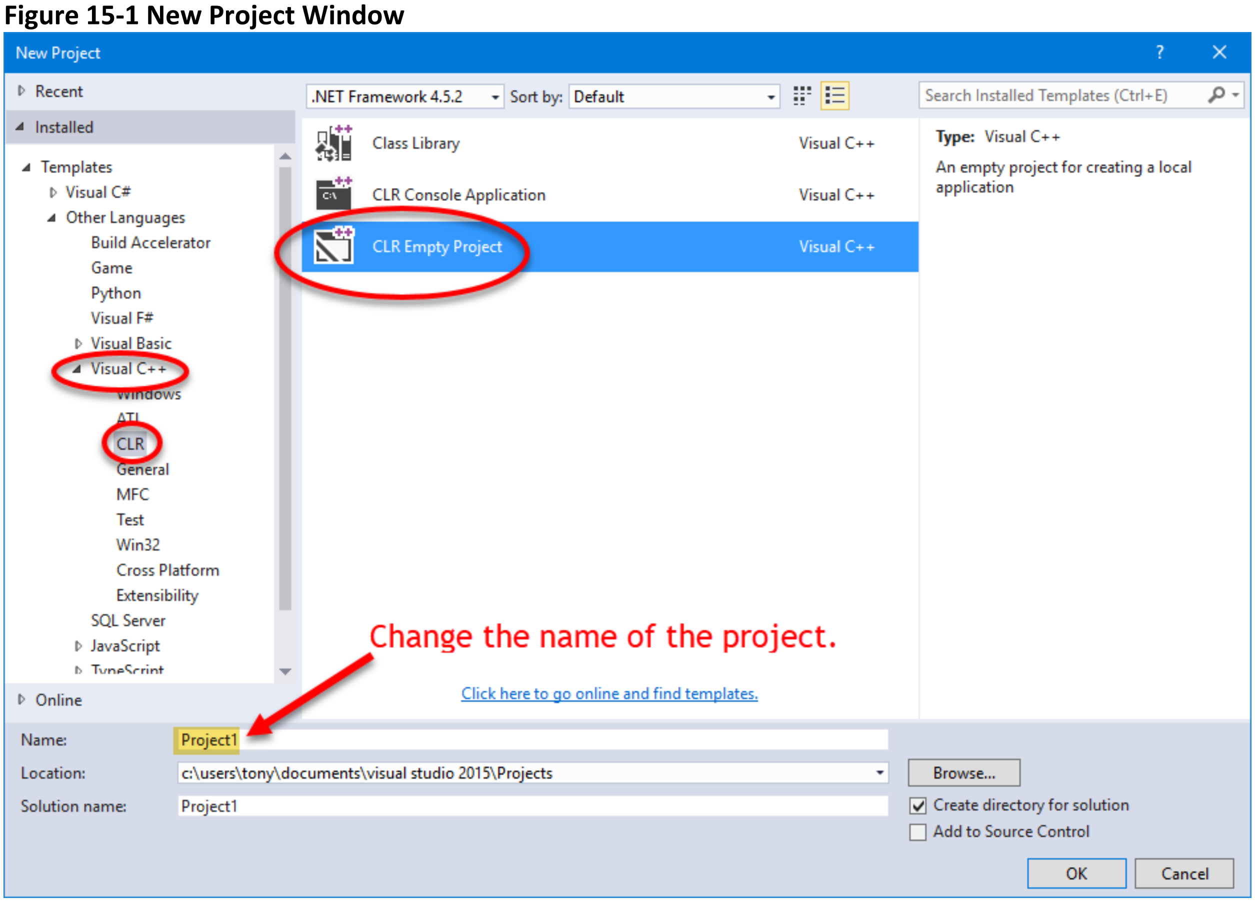
In this tutorial you will build a simple Hello World application in Visual C++. Make sure you have downloaded Visual Studio 2015 Community edition (or a later version of Visual Studio). You can download Visual Studio from `www.visualstudio.com`.

Step 1: Start Visual Studio

Step 2: To start a new project, On the menu bar, click *File*, then select *New*, then select *Project…*

Step 3: The *New Project* window will appear. As shown in Figure 15-1, Select *Visual C++*, *CLR* and *CLR Empty Project*. At the bottom of the window, change the name of the project to *MyFirstProject*, and click the *OK* button.

Figure 15-1 New Project Window



Step 4: On the menu bar, click *Project*, then select *Add New Item…*

Step 5: The *Add New Item* window will appear. As shown in Figure 15-2, under *Visual C++* select *UI*, and select *Windows Form*. Click the *Add* button. The form (window) will appear as shown in Figure 15-3.

Figure 15-2 Add New Item Window

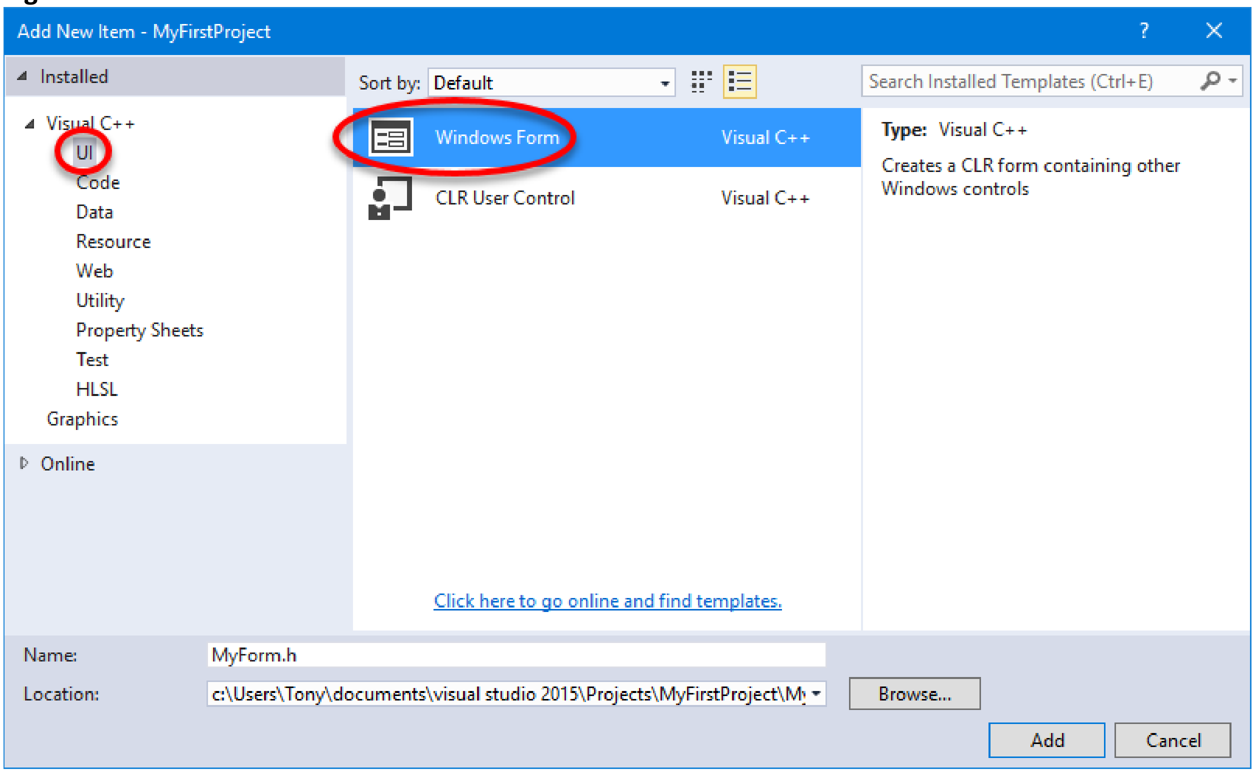
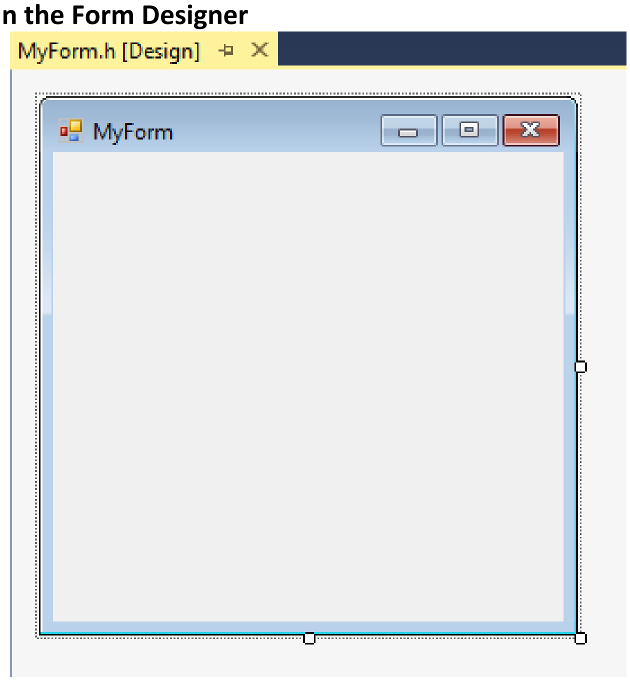
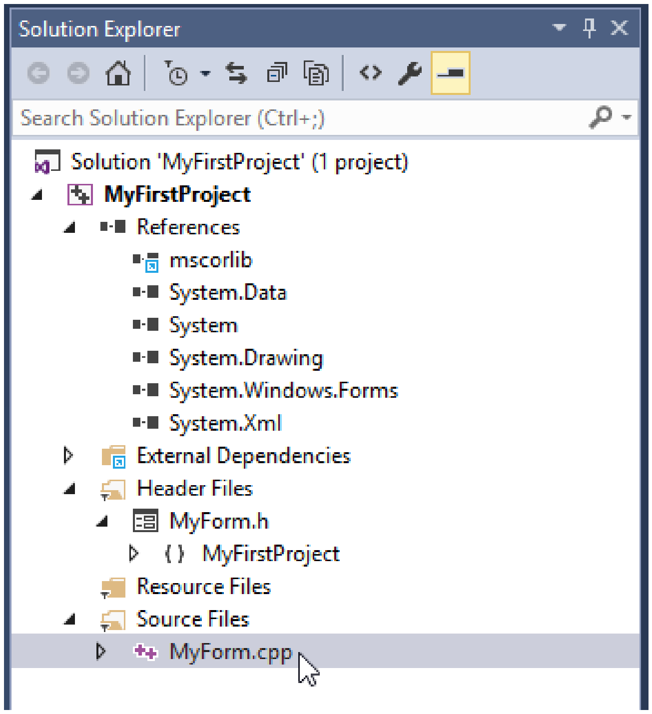


Figure 15-3 Form shown in the Form Designer



Step 6: In the Solution Explorer (on the right side of the screen), double click the entry for *MyForm.cpp*, as shown in Figure 15-4.

Figure 15-4 Solution Explorer



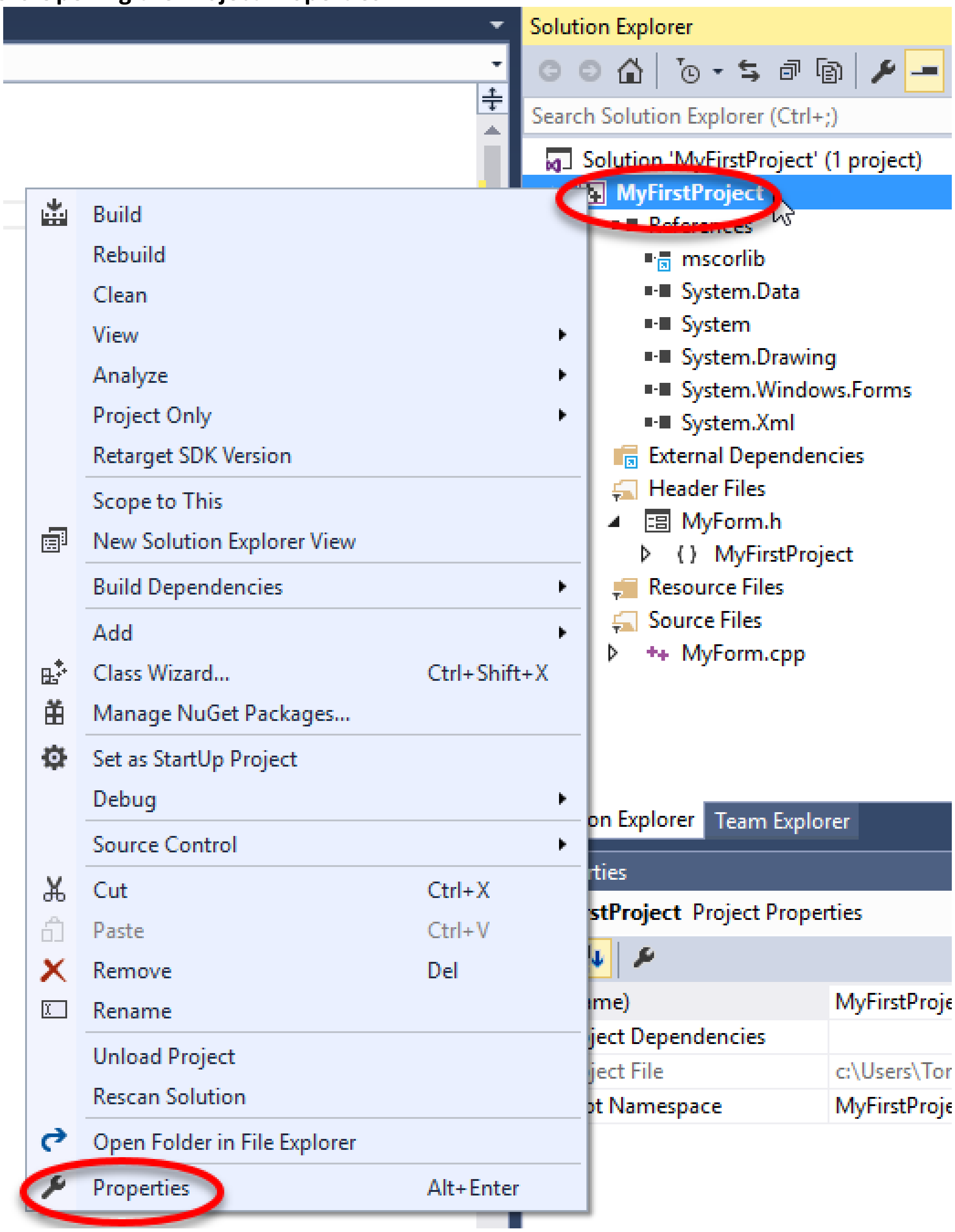
Step 7: The MyForm.cpp file should now be open in the code editor. Modify the file so it appears as shown in Figure 15-5.

Figure 15-5 The modified code for MyForm.cpp

```
#include "MyForm.h"

using namespace System;
using namespace System::Windows::Forms;


[STAThread]
void Main(array<String^>^ args)
{
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault(false);

    MyFirstProject::MyForm form;
    Application::Run(%form);
}

```

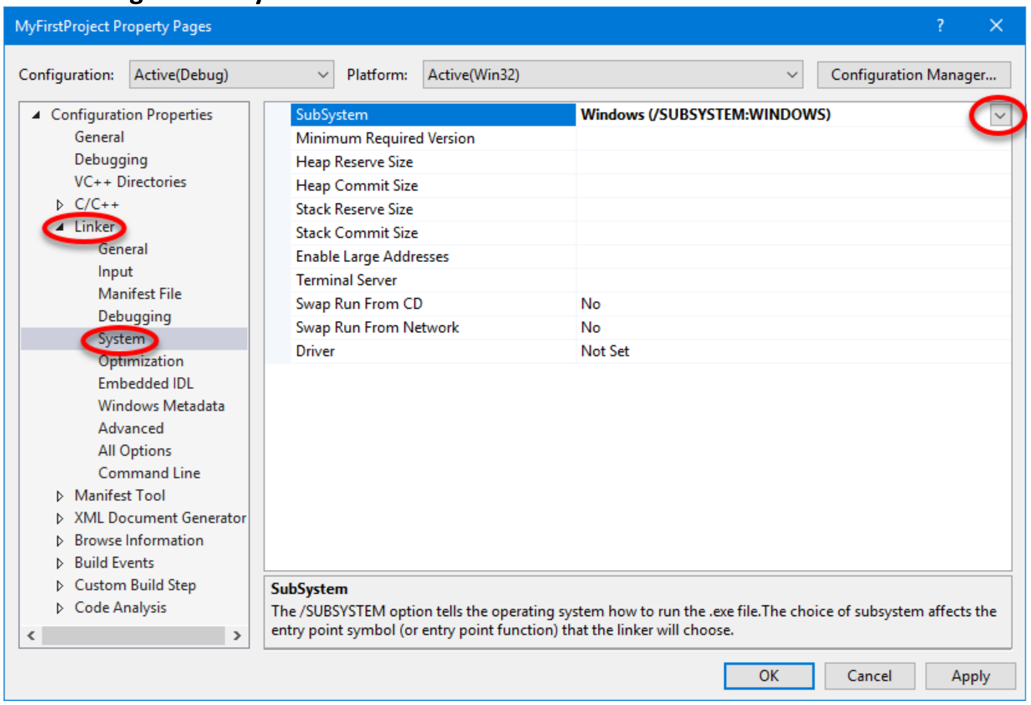
Step 8: In the Solution Explorer on the right side of the screen, right-click MyFirstProject (as shown in Figure 15-6) and then select Properties from the menu (also shown in Figure 15-6).

Figure 15-6 Opening the Project Properties



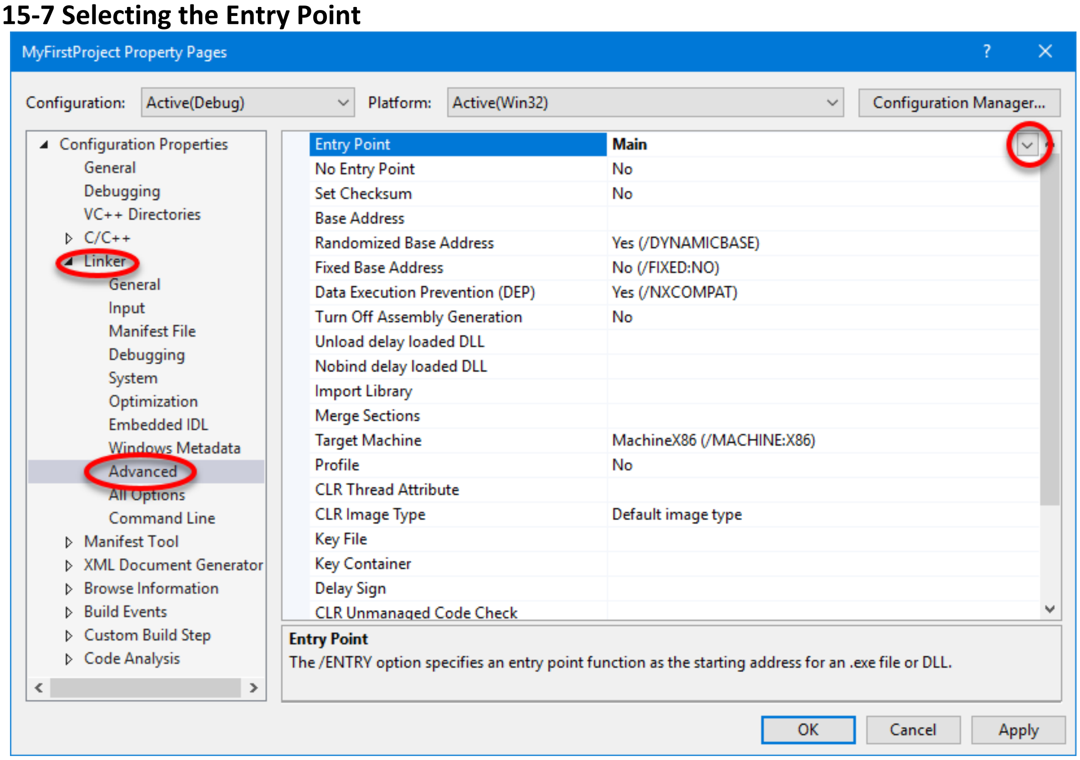
Step 9: In the Property Pages window, select *Linker*, then select *System* (as shown in Figure 15-7). Then click the down-arrow for *SubSystem* (also shown in Figure 15-7) and select *Windows (/SUBSYSTEM:WINDOWS)*

Figure 15-7 Selecting the subsystem



Step 10: Under Linker, select *Advanced* (as shown in Figure 15-8). Then click the down-arrow for *Entry Point* (also shown in Figure 15-8) and enter *Main*. (Be sure to capitalize the "M" in "Main".)

Figure 15-7 Selecting the Entry Point



Step 11: Click the *MyForm.h [Design]* tab, as shown in Figure 15-9. This opens the form in the Form Designer as shown in Figure 15-10. Click Toolbox, which appears along the left edge of the window (also shown in Figure 15-10).

Figure 15-9 Switching to the Form Designer

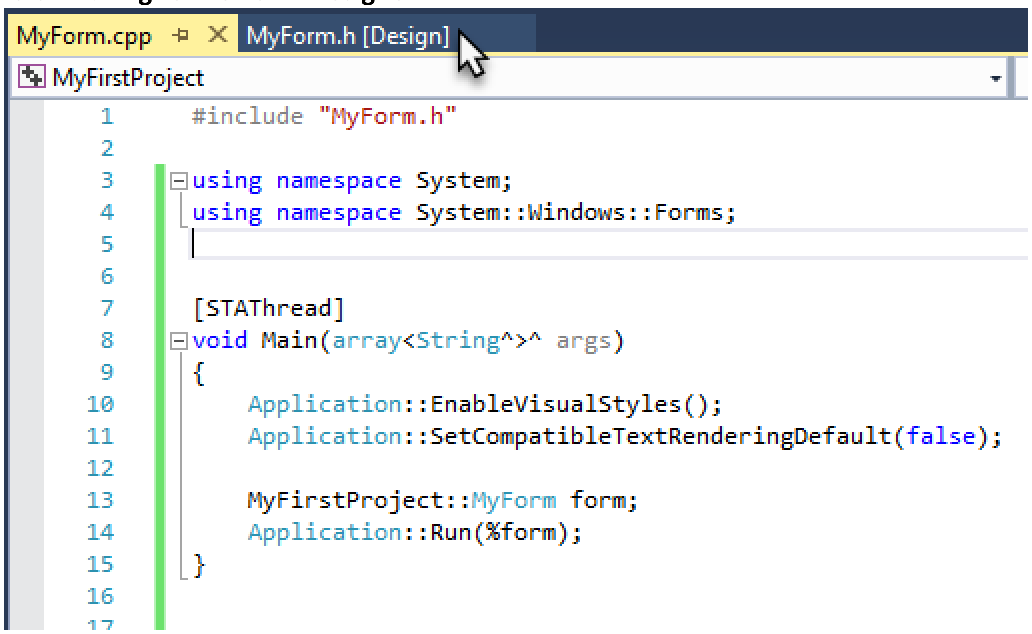
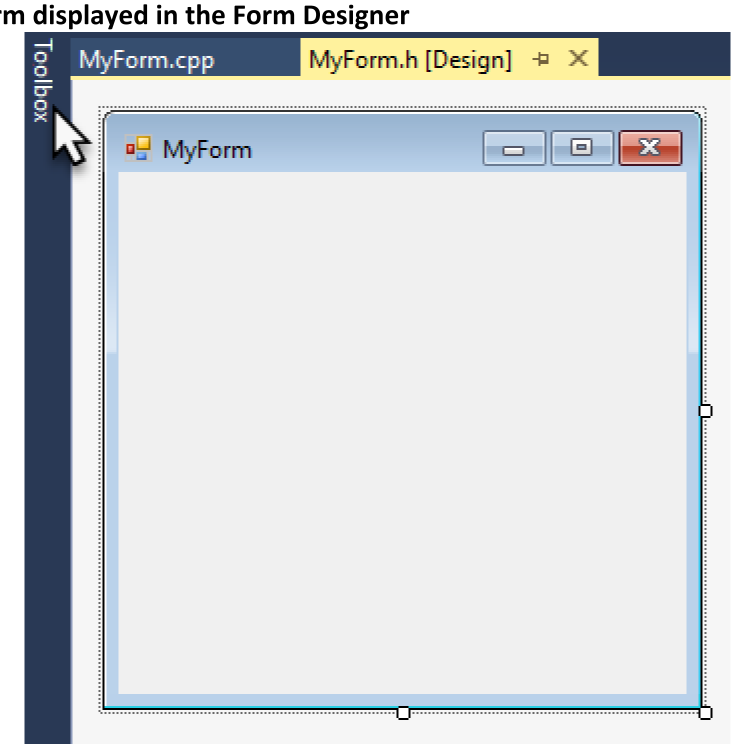
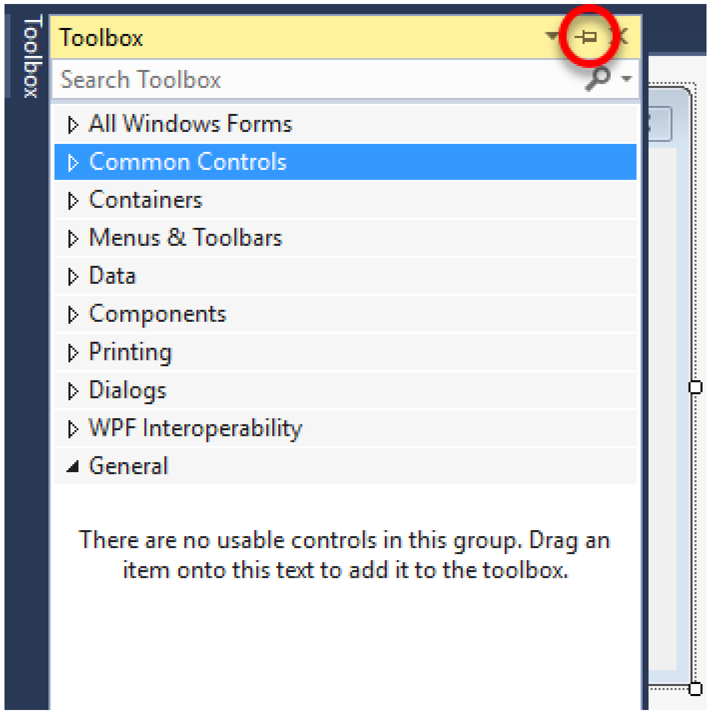


Figure 15-10 The Form displayed in the Form Designer



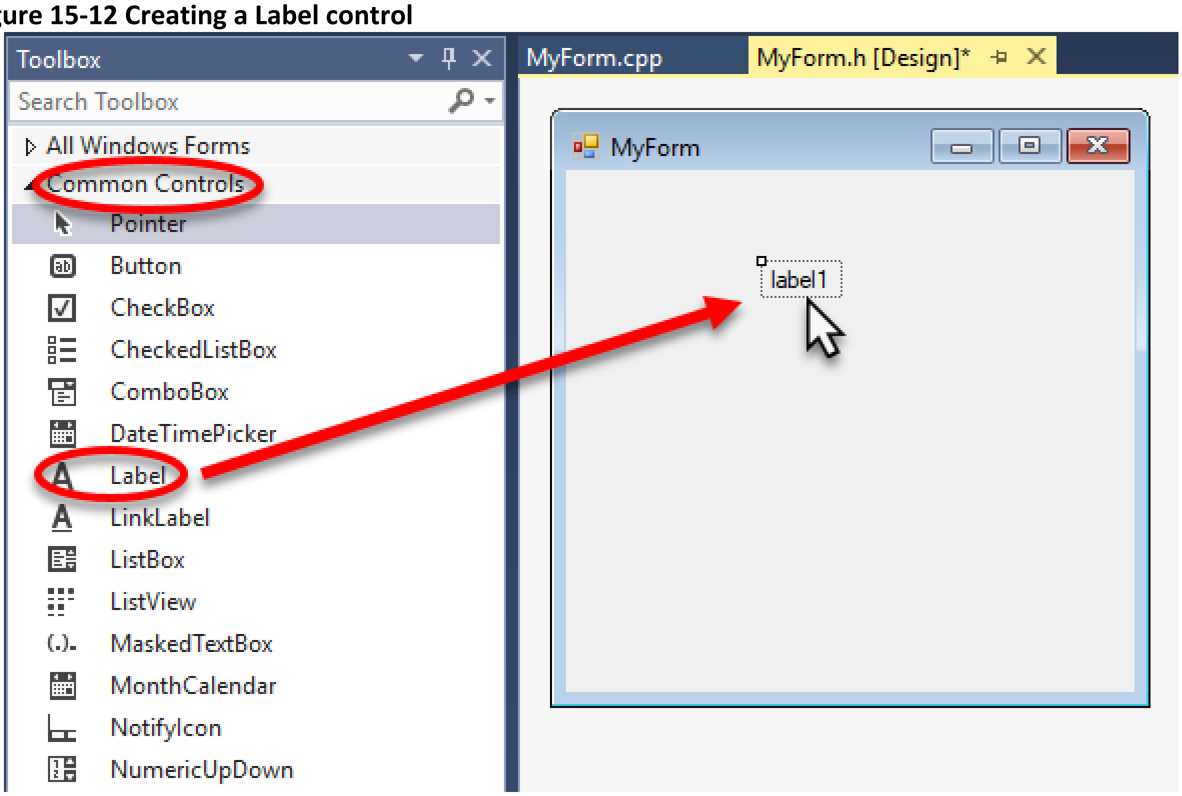
Step 12: The Toolbox will open as shown in Figure 15-11. Click the pushpin icon (also shown in Figure 15-11) to disable autohide mode.

Figure 15-11 Disabling Autohide for the Toolbox



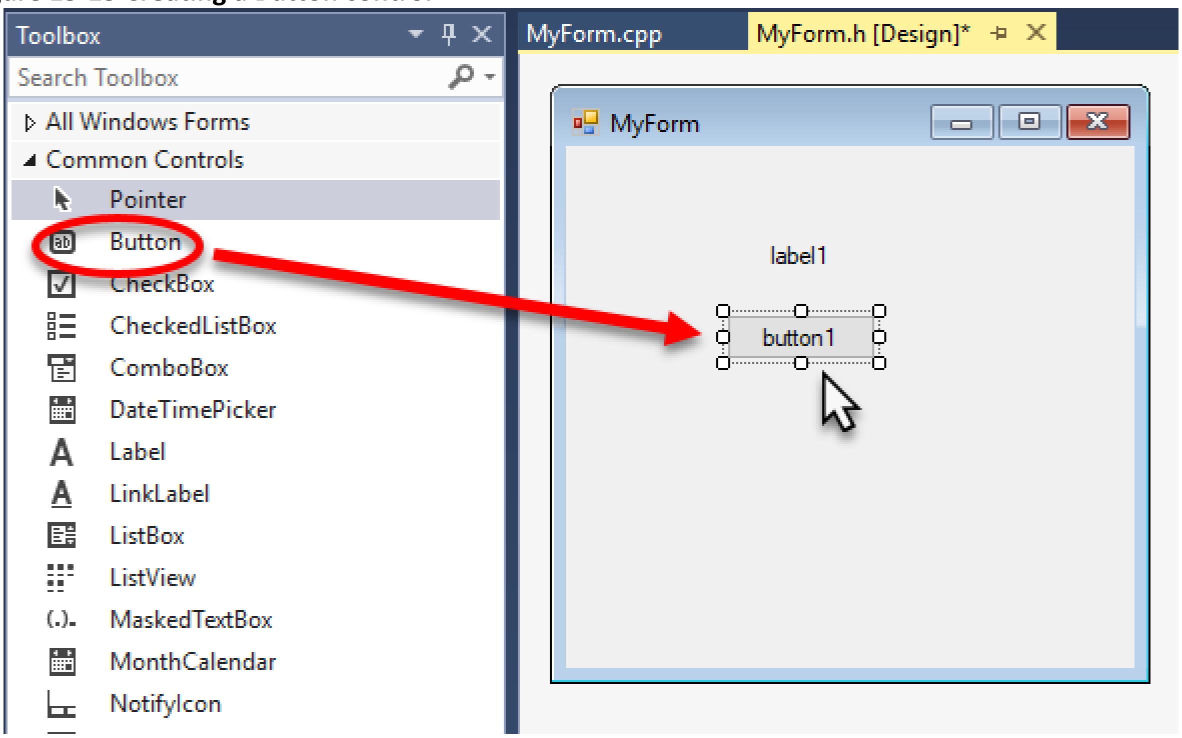
Step 13: Open the *Common Controls* section of the Toolbox (as shown in Figure 15-12) and then click and drag a *Label* from the Toolbox to the form (also shown in Figure 15-12). This creates a Label control named Label1.

Figure 15-12 Creating a Label control



Step 14: Click and drag a *Button* from the Toolbox to the form (as shown in Figure 15-13). This creates a Button control named Button1.

Figure 15-13 Creating a Button control



Step 15: Now you will write the event handling code for the Button control. Make sure the Button control is selected in the Designer window, as shown in Figure 15-14. In the Properties window, next to the *Click* property, type *button1_click* (also shown in Figure 15-14). This causes an empty function named `button1_click` to be created in the MyForm.h file, as shown in Figure 15-15.

Figure 15-14 Specifying a Click Event Handler

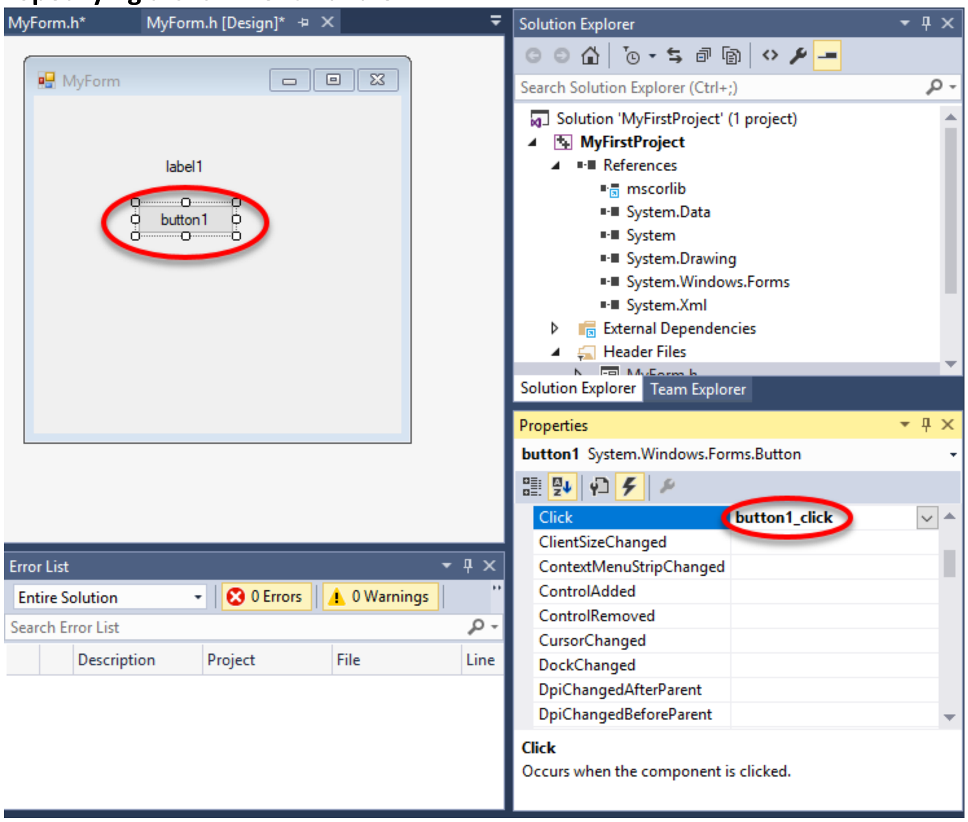
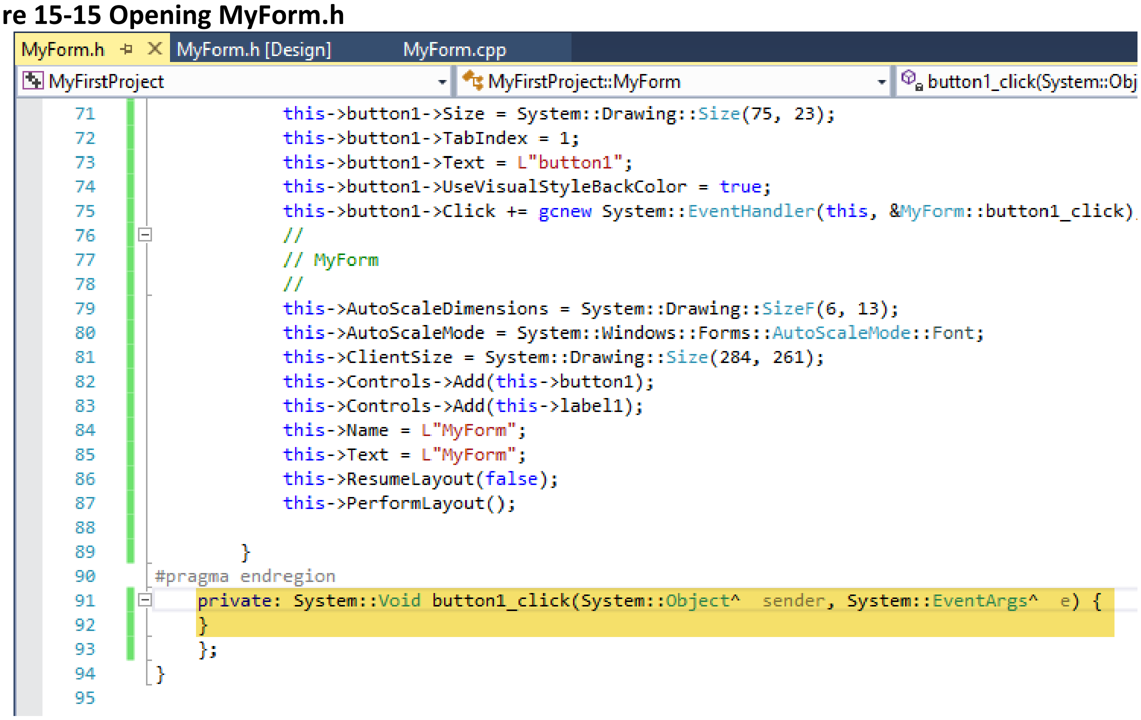


Figure 15-15 Opening MyForm.h



Step 16: Inside the button1_click function, add the following line of code:

```
this->label1->Text = "Hello world!";
```

The function should now appear as:

```
private: System::Void button1_click(System::Object^  sender, System::EventArgs^  e) {
    this->label1->Text = "Hello world!";
}
```

Step 17: Press the F5 key on your keyboard to compile and execute the program. You will see a window displaying the message "The project is out of date. Would you like to build it?" Click the Yes button. The window shown in Figure 15-16 should appear. Click the Button control and you will see the message Hello World displayed in the Label control, as shown in Figure 15-17.

Figure 15-16 The application's window

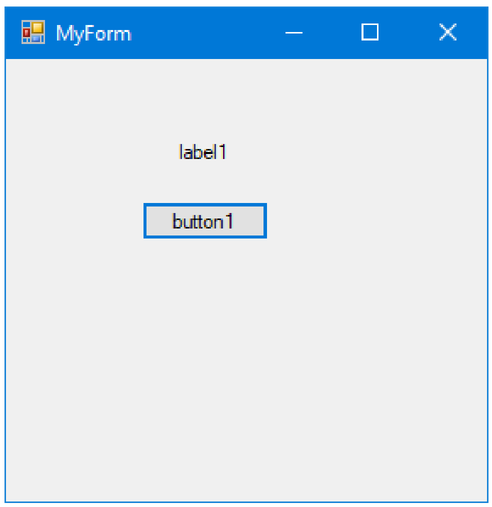
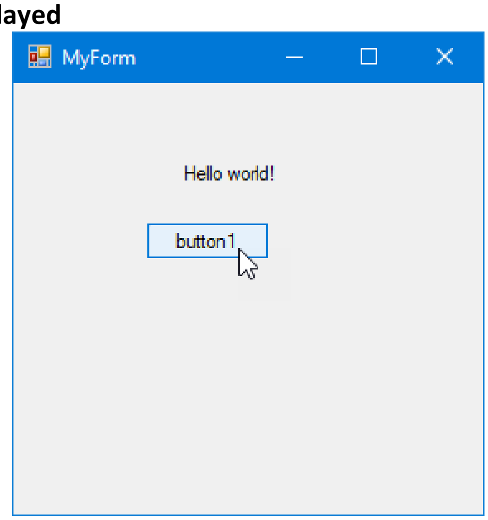


Figure 15-17 The message displayed



Step 18: Close the application's window.