6 Strings and Screen I/O

OBJECTIVES

- *Understand C++ strings.*
- *Use character arrays.*
- *Use console I/O for input and output.*

Overview

You learned about the character data type in Chapter 4 and that a group of characters put together to create text is called a string. Two types of strings are discussed in this chapter: string literals and strings stored in character arrays. You will learn how each type of string is stored and how they can be used in your programs.

You will also learn more about cin and cout as well as how to make the best use of them, including formatting your output.

CHAPTER 6, SECTION 1

Strings

Strings are one of the most useful kinds of data in a program. Strings help programs communicate with the user and allow computers to process data other than numbers. For example, when you prompt the user with a statement like "Enter the cow's weight:" you are using a string. Or when you ask the user to enter the cow's name, the name is a kind of string.

You must thoroughly understand how strings are stored and used in C++ so that you can avoid any difficulty or frustration. C++, in reality, provides the programmer with a flexibility in working with strings that you may come to appreciate.

WHAT IS A STRING AND HOW IS IT STORED?

A string is a group of characters that are put together to create text. In C++, you will be concerned with two ways of storing strings: string literals and character arrays.

UNDERSTANDING STRING LITERALS

The text "Hello" in the statement below is a string, specifically, a ***string literal.*** You used string literals with cout statements in earlier chapters.

```
cout << "Hello";
```

Notice that a string literal is "hard coded" into your program and is enclosed in quotation marks. A string literal is similar to a constant because it is part of the program's source code and remains the same while the program runs.

A lone character that appears in single quotes (') is a ***character literal.*** Single quotes are used only when a single character is between the quotes. String literals must be in double quotes. Figure 6-1 shows an example of a character literal and a string literal.

String literals end with an invisible character called a ***null terminator.*** The null terminator is the character represented by the ASCII value zero. For example, Figure 6-2 shows how the string literal in Figure 6-1 is stored in memory.

```
initial = 'T';              // 'T' is a character literal

cout << "Mark Gentry";      // "Mark Gentry" is a string literal
```

FIGURE 6-1
Character literals are enclosed in single quotes. String literals are enclosed in double quotes.

Each character in the string literal, including the null terminator, appears sequentially in memory. The null character tells the compiler the string has ended.

Note

In C++, \0 represents the null terminator. Together, the \ and 0 represent one character: the ASCII value zero.

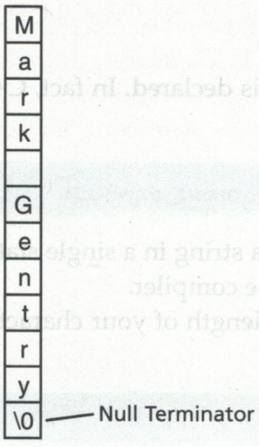

FIGURE 6-2
Strings end with a character called the null terminator.

String literals are different from an array of characters. An array of characters is used to store strings that change as the program runs. A string literal remains the same throughout the program's execution.

UNDERSTANDING ARRAYS OF CHARACTERS

String literals are very useful. But just like you need variables for numbers, you often need the equivalent of a variable for strings. However, C++ lacks a dedicated string data type, at least in the same way as there is an int or float type for numbers. Arrays of characters are used to store strings in C++.

An ***array*** is a group of variables of the same data type that appear together in the computer's memory. Later in this book, you will learn how to use all kinds of arrays. For now we will be concerned only with arrays of characters.

An array of characters is a group of variables of the type char. When used together, each variable holds a character and the last variable in the string holds the null terminator. Remember, each char variable holds an ASCII value for the character, and the null terminator is the ASCII value zero. Figure 6-3 shows how the name Kaley would be stored as a character array.

USING C++ STRINGS

Because C++ strings are stored in character arrays, programmers have a great deal of flexibility when using strings. You can devote exactly the amount of memory you need to strings and easily manipulate individual characters in strings. But like other data types, using character arrays requires that you know how to properly declare and initialize them for use.

DECLARING CHARACTER ARRAYS

Declaring a character array is easy. The statement below declares a character array.

```
char student_name[21];
```

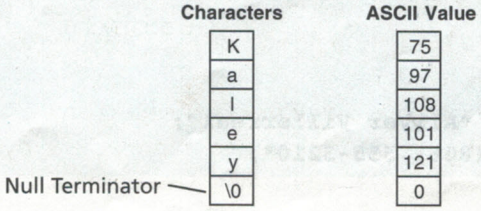

FIGURE 6-3
An array of characters holds the ASCII value of each character in the string. The null terminator has the ASCII value of zero.

The name of the character array is **student_name**. Twenty-one char variables are set aside for the array. The character array **student_name** can hold a string of up to 20 printable characters plus the null terminator. You must always save room for the null terminator when declaring character arrays for strings.

Extra for Experts

A character array and a string are different. In C++, you declare a character array rather than a string. A string may then be stored in a character array.

INITIALIZING CHARACTER ARRAYS WITH STRINGS

A character array can be initialized when the array is declared. In fact, C++ makes the job easy. Consider the statement below.

```
char student_name[21] = "Aliver Villarreal";
```

The character array is declared and initialized with a string in a single statement. The null terminator is automatically entered by the compiler.

You may even use an initialization string to set the length of your character array if you are using fixed data as shown below:

```
char phone_number[] = "(806) 555-3210";
```

Note

Character arrays are the best data type to use with ZIP codes and phone numbers because ZIP codes and phone numbers are, in essence, characters. Phone numbers stored in character arrays, for example, can include parentheses and hyphens which are not allowed in numeric data types.

Because all of the phone numbers use the same format, the character array may be declared without specifying the exact number of characters. When you leave out the length of the array, the compiler creates an array just large enough for your string and the null terminator.

WARNING

Do not leave the brackets empty when declaring a character array if you have no initialization string. A statement like `char phone_number[];`*, by itself, will fail to allocate storage space for the array and on some compilers will cause an error.*

Also realize that once an initialization string is used to declare a character array its size is fixed. Attempting to store larger strings can cause problems in your program's execution and the output.

EXERCISE 6-1 DECLARING AND INITIALIZING CHARACTER ARRAYS

1. Enter the following program and save the source code as *CHARRAY.CPP*.

```
// CHARRAY.CPP
#include <iostream.h>

main()
{
  char student_name[21] = "Aliver Villarreal";
  char phone_number[] = "(806) 555-3210";
```

```
    cout << student_name << '\n';
    cout << phone_number << '\n';
    return 0;
}
```

2. Compile and run the program to see if the character arrays are correctly initialized.
3. Leave the source code file open for the next exercise.

THE strcpy FUNCTION

If you try to assign a string to a variable as shown below you will get an error message. You can use an assignment operator only to store a string in a variable in the declaration statement.

```
student_name = "Aliver Villarreal";    // NOT LEGAL!!!
```

To get a string into an existing character array, you must use a special function: the **strcpy** function.

The **strcpy** function (pronounced "string copy") is very convenient when working with strings in C++. With the **strcpy** function, a string literal can be stored in a character array, or one character array can be copied to another. However, you must be sure the character array receiving the new data is long enough to accommodate it. The statement below shows how **strcpy** could be used to store a name in the **student_name** character array.

```
strcpy(student_name,"Aliver Villarreal");
```

The **strcpy** function can also be used to initialize a character array or to make a character array empty. The statement below initializes the **student_name** character array.

```
strcpy(student_name, "");
```

Shown below is the format of a statement that copies the contents of one character array to another.

```
strcpy(destination_array, source_array);
```

In order for the **strcpy** function to work, the compiler directive **#include <string.h>** must appear at the top of the program.

EXERCISE 6-2 USING strcpy

1. Add the following compiler directive at the top of the program you prepared in Exercise 6-1.

```
#include <string.h>
```

2. Add the following lines at the end of the program, before the closing brace.

```
strcpy(student_name, "Tan Pham");
cout << student_name << '\n';
```

3. Compile and run the program to see that the string "Aliver Villarreal" was replaced with "Tan Pham" in the character array.
4. Save the source code and close.

GETTING INTO TROUBLE

The flexibility of C++ strings can sometimes get you into trouble. The reason is that C++ does not prevent you from storing a string that is too long for the character array. What happens to the part of the string that extends beyond your array? It spills over into other storage spaces in memory and writes over them. This can cause real problems. In most cases, other variables or character arrays are overwritten. The result can be unpredictable at best. At worst, the computer can crash.

EXERCISE 6-3 STRING ERROR

1. Enter the following program and save the source code file as *STRERROR.CPP*.

```
// STRERROR.CPP
// Program that demonstrates string pitfalls.

#include <iostream.h>
#include <string.h>

main()
{
  char student_name[21] = "Brandon Adkins";  // declare four
  char city[] = "Lubbock";                   // character strings
  char state[3] = "TX";
  char ZIP_code[11] = "79413-3640";

  cout << student_name << '\n'; // display the strings
  cout << city << '\n';
  cout << state << '\n';
  cout << ZIP_code << '\n';

  strcpy(state,"California");   // copy a string that is too long
                                // into the state character array

  cout << student_name << '\n'; // display the strings again
  cout << city << '\n';         // showing the overwritten array
  cout << state << '\n';
  cout << ZIP_code << '\n';
  return 0;
}
```

2. Compile and run the program. The results will vary depending on how your compiler orders the data in memory. But when the word *California* is copied into an array that is intended to hold only a two-character abbreviation, part of the word *California* will probably end up in another character array. Check the output from the second set of output statements to see if you can find the array that was overwritten.
3. Close the source code file.

SECTION 6.1 QUESTIONS

1. What signifies the end of a C++ string?
2. Write a statement that prints the string literal "ABC" to the screen.
3. Write a statement that declares a character array of length 10 named ZIP.
4. Write a statement that declares a character array and initializes it to "555-55-5555". Let the compiler determine the length of the array.
5. Write a statement that copies the string in **`new_string`** to the character array **`my_string`**.
6. Write a statement that stores the string literal "New York" into the character array **`state`**.

PROBLEM 6.1.1

Write a program that declares a character array of length 20 and initializes it with your name. Next, have the program replace your name with the name Betsy Little. Have the program print the contents of the character array to the screen before and after replacing your name with Betsy Little's name. Save the source code file as *BETSY.CPP*.

PROBLEM 6.1.2

Write a program that declares two character arrays named A and B, and make each of length 10. Initialize B with the string "Hello", then copy the contents of B to A. Finally, copy the string "World" into B and print the contents of both strings to the screen. Save the source code file as *HELLO.CPP*.

CHAPTER 6, SECTION 2

Screen Input/Output

You have used simple input and output statements like the ones below without knowing all of the details.

```
cout << i << '\n';
cin >> j;
```

In this section, you will learn how to get the most out of input and output statements by learning about the missing details.

USING cin AND cout

We have treated cin and cout (pronounced *see-in* and *see-out*) as commands up to this point. You may be surprised, however, to learn that the << and >> symbols actually represent the action. Consider the simple statements below.

```
cout << j;
cin >> i;
```

The << and >> symbols are actually operators, like + and * are operators. The << symbol is the output operator, and >> is the input operator. As you know, the variable to the right of the << or >> operator is what is being input or output. So what are cout and cin? Well, cout is the *destination* of the output, and cin is the source of the input. The words "cin" and "cout" represent sources and destinations for data.

Note

The << operator is also referred to as the **extraction operator.** *The >> operator is also referred to as the* **insertion operator.**

Some beginning programmers find it difficult to remember when to use << and when to use >>. There is a method you can use to help you remember. The symbols in the input and output operators point in the direction that the data is flowing. For example, in the statement `cout << j;`, the data is flowing from the variable `j` to the destination of the output (cout). When you use the input operator (as in `cin >> i;`), the data flows from the source of the input (cin) to the variable `i`.

STREAMS

When you think of a stream, you probably think of water flowing from one place to another. In C++, a ***stream*** is data flowing from one place to another. You should think of C++ streams as channels that exist to provide an easy way to get data to and from devices. The stream that brings data from your keyboard is cin, and the stream that takes data to your screen is cout.

For example, your monitor (screen) is a device. You don't have to understand exactly how output gets to the screen. You just have to know that cout is the stream that leads to your screen. When you use the output operator to place something in the cout stream, your screen is the destination.

There is almost always an exception to the rule, right? You should know that cin and cout may represent devices other than the keyboard and screen. The cin stream reads from what is called the ***standard input device,*** and the cout stream leads to the ***standard output device.*** By default, the standard input device is the keyboard and the standard output device is the screen. There are other streams that you will learn about in later chapters.

WARNING

The `#include <iostream.h>` *directive is required to use streams.*

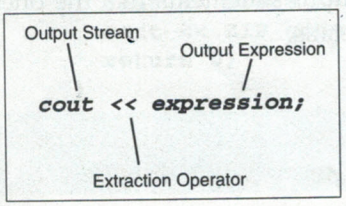

FIGURE 6-4 The << operator is used for output.

USING CONSOLE I/O

The term ***console I/O*** refers to using the screen and keyboard for input and output (I/O is an abbreviation of input/output). In other words, the standard use of cin and cout is console I/O. Let's look at some examples of console I/O to make sure you understand the role of each part of the statements.

Figure 6-4 illustrates the general form of the << operator. The << operator indicates to the compiler that the statement is producing output. The destination of the output is the standard output device (the screen). The output can be any valid C++ expression.

The examples in Figure 6-5 show how the output can be a string literal, a variable, or a mathematical expression. The figure also shows that more than one item can be output in a single statement by using multiple output operators.

```
cout << "This string literal will appear on the screen. \n";
cout << distance;
cout << length_of_room * width_of_room;
cout << "The room is " << area << " square feet.\n";
```

FIGURE 6-5
The output operator can be used in a variety of ways.

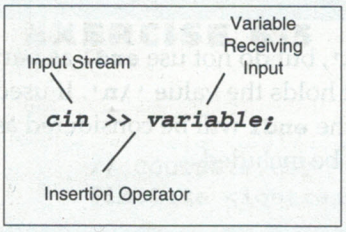

FIGURE 6-6
The >> operator is used for input.

Note

Remember, << is an operator. Therefore you can use it many times in the same output expression, just like you can use a mathematical operator multiple times in the same expression. For example, the statement `n = 2 + 4 + 5`, *uses the addition operator twice. In the same way, the output operator can appear more than once in a statement.*

Figure 6-6 illustrates the general form of the >> operator. The >> operator tells the compiler that the statement is requesting input. The source of the input is the standard input device (the keyboard). The destination of the input must be a variable or variables.

EXERCISE 6-4

BASIC CONSOLE I/O

1. Load, compile, and run *BASICIO.CPP* to review the basic use of input and output operators.
2. Close the source code file.

You will learn more about using the input operator later. For now, let's consider the options associated with the output operator.

FORMATTING OUTPUT

The appearance of the text on the screen can be controlled by formatting the output of your program. Usually, the output operator does a good job of formatting whatever data you output. There are times, however, when you need more control over the appearance of the output. C++ provides some options to help you get the output you want.

On the Net

The end-of-line character can mean different things on different operating systems. To learn more about how computers handle the end-of-line character, see http://www.ProgramCPP.com. See topic 6.2.1.

WHAT IS '\n'?

You have been including `'\n'` in output statements without a good explanation of what `'\n'` does. It is an important part of formatting output because it causes the cursor to return to the next line of the screen. The `\n` character is called the new line character or the end-of-line character. Use it in any output statement that completes a line. The new line character has the same effect in an output statement as pressing the Return or Enter key in a word processor.

The `\n` character must appear in double quotes if it is used in conjunction with other characters or may be used with single quotes if it appears alone. See the examples below.

```
cout << i << '\n';  // single quotes because it is a single character
cout << "String\n"; // double quotes because it is part of a string
```

The `\n` character can be enclosed in double quotes, even when it appears alone. The compiler will treat the character as a string because it is in double quotes. The statement will, however, produce the same result.

There is an alternative to `\n` that you may find easier to enter and more readable. You can enter `endl` in the place of `'\n'`. For example, the two statements below are functionally identical.

```
cout << i << '\n';
cout << i << endl;
```

You can use `endl` in place of the character `'\n'`, but do not use `endl` as part of a larger string. Think of `endl` as a constant that holds the value `'\n'`. If used in a statement like `cout << "String endl";`, the `endl` will be considered as part of the string and no end-of-line character will be included.

Other Special Characters

In addition to \n, there are other special characters (sometimes called escape sequences) you can use in output statements. The first one generates a tab character. The others allow you to print characters to the screen which would otherwise be unprintable because of the way they would be interpreted by the compiler.

The table below lists the important escape sequences.

Character Sequence	Result
\t	*Generates a tab character to move the cursor to the next tab stop.*
\\	*Prints a backslash (\).*
\'	*Prints a single quote mark (').*
\"	*Prints a double quote mark (").*

USING setf AND unsetf

Each stream has format options that can be changed. Table 6-1 lists the options that can be used.

OPTION	DESCRIPTION
`left`	Left-justifies the output
`right`	Right-justifies the output
`showpoint`	Displays decimal point and trailing zeros for all floating-point numbers, even if the decimal places are not needed
`uppercase`	Displays the "e" in E-notation as "E" rather than "e"
`showpos`	Displays a leading plus sign before positive values
`scientific`	Displays floating-point numbers in scientific ("E") notation
`fixed`	Displays floating-point numbers in normal notation

TABLE 6-1

Now, examine how the format option *right* (indicating that the output is to be right justified) is used in the expanded format statement below.

```
cout.setf(ios::right);
```

You will learn more about how to use statements like the one above later. What is important now is that you understand that the word *right* is the format option.

Extra for Experts

If neither the scientific nor fixed option is set, the compiler decides the method to display floating-point numbers based on whether the number can be displayed more efficiently in scientific or fixed notation.

You can remove the options by replacing **setf** with **unsetf**, as in the example below.

```
cout.unsetf(ios::scientific);
```

EXERCISE 6-5 USING setf AND unsetf

1. Enter the following program and save the source code file as *COUTSETF.CPP*.

```
// COUTSETF.CPP
#include <iostream.h>

main()
{
  float x = 24.0;

  cout << x << '\n';              // displays 24

  cout.setf(ios::showpoint);
  cout << x << '\n';              // displays 24.000000

  cout.setf(ios::showpos);
  cout << x << '\n';              // displays +24.000000

  cout.setf(ios::scientific);
  cout << x << '\n';              // displays +2.400000e+01

  cout.setf(ios::uppercase);
  cout << x << '\n';              // displays +2.400000E+01

  cout.unsetf(ios::showpoint);
  cout << x << '\n';              // displays +2.4E+01

  cout.unsetf(ios::showpos);
  cout << x << '\n';              // displays 2.4E+01

  cout.unsetf(ios::uppercase);
  cout << x << '\n';              // displays 2.4e+01

  cout.unsetf(ios::scientific);
  cout << x << '\n';              // displays 24
  return 0;
}
```

2. Compile and run the program to see the effects of the format options.
3. Close the source code file.

USING THE I/O MANIPULATORS

Another set of format options are available in C++: the ***I/O manipulators.*** They may be placed directly in the output statement. Look at the example below which uses the **setprecision** I/O manipulator.

```
cout << setprecision(2) << price << '\n';
```

The **setprecision** manipulator sets the number of digits displayed to the number provided in the parentheses. The **setprecision** manipulator affects all floating-point numbers that follow it in the statement. It also affects any floating-point numbers output to the cout stream until **setprecision** is called again to set another precision. Table 6-2 shows two of the manipulators available.

Note

In order to use I/O manipulators, you must use the directive `#include <iomanip.h>` *to include the necessary code to make the manipulators available.*

WARNING

The `setprecision` *manipulator may have a different effect depending on your compiler. Most compilers use* `setprecision` *to set the total number of digits displayed. For example, the value 5.2 will display as 5.20 if the precision is set to 3. Some compilers (particularly older compilers) use* `setprecision` *to set the number of digits to the right of the decimal place. For example, the value 5.2 will display as 5.200 if the precision is set to 3.*

The **setw** manipulator can be used to change default field widths. You can use **setw** to set a minimum field width or use it to format numbers.

TABLE 6-2

MANIPULATOR	DESCRIPTION
`setprecision(digits)`	Set the number of digits to be displayed
`setw(width)`	Set the width of the field allocated for the output

For example, if i = 254, j = 44, and k = 6, the statement `cout << i << j << k << '\n';` produces the output **254446** because only the space necessary to output the numbers is used. The statement below, however, adds spaces to the left of each number to give formatted output.

```
cout << setw(10) << i << setw(10) << j << setw(10) << k << endl;
```

The output of the statement above appears as shown below.

```
254        44         6
```

The best way to see the difference is to try it yourself in the next exercise.

EXERCISE 6-6 I/O MANIPULATORS

1. Enter the following program and save the source code as *IOMANIP.CPP*.

```
// IOMANIP.CPP
#include <iostream.h>
#include <iomanip.h>

main()
```

```
{
  int i = 1499;
  int j = 618;
  int k = 2;
  float a = 34.87432;

  cout << setw(10) << i << setw(10) << j << setw(10) << k << endl;
  cout << setw(10) << k << setw(10) << i << setw(10) << j << endl;

  cout << setprecision(3) << setw(10) << a << endl;
  return 0;
}
```

2. Compile and run the program to see how the **setw** manipulators affect the output of the integers. Notice how the last statement combines the **setprecision** and **setw** manipulators to properly align columns and decimal places.
3. Close the source code file.

GETTING INPUT WITH >>

You have learned most of what there is to know about using the >> operator. However, you must still learn how to input more than one variable in a single statement and understand the limitations of the >> operator.

GETTING MORE THAN ONE VALUE

More than one value can be input from the keyboard using a statement like the one below.

```
cin >> i >> j >> k;
```

When the statement above is executed, the program will pause and wait for three inputs. The three inputs could be entered all on the same line with spaces or tabs between them. Or the user can press Enter between the entries. A space, tab, or Enter is considered whitespace, and therefore is a delimiter or separator between the data. The use of a comma to separate data is prohibited in C++.

Note

*A **delimiter** is a character that signals the computer that one piece of data is ending and another one is beginning. In C++, spaces, tabs, and the Enter key create what is called **whitespace**.*

EXERCISE 6-7 MULTIPLE INPUTS IN A STATEMENT

1. Open *MULTI_IN.CPP*.
2. Compile and run the program. Enter **23 4 786** as input.
3. Run again and this time delimit the data by pressing Enter between the numbers.
4. Close the source file.

INPUTTING CHARACTERS

The >> operator can be used to input characters. If the user enters more than one character, only the first character will be stored in the variable.

EXERCISE 6-8 INPUTTING CHARACTERS

1. Enter the following program. Save the source code file as *INCHAR.CPP*.

```
// INCHAR.CPP
#include <iostream.h>

main()
{
  char c;

  cout << "Enter a single character: ";
  cin >> c;
  cout << "You entered " << c << '\n';
  return 0;
}
```

2. Compile and run the program. Provide different input (both single characters and complete words) and observe how they are processed similarly.
3. Close the source file.

INPUTTING STRINGS

The >> operator can be used to input strings, but there is a problem. As soon as a space or tab is reached (delimiters), the rest of the string is ignored. If you want to input a single word, you can use the same kinds of statements you use for numeric or character input. If you need to input strings with whitespace in them, you will have to use a function called **get**, which you will learn about after the next exercise.

EXERCISE 6-9 INPUTTING WORDS

1. Open *INWORD.CPP*.
2. Compile and run the program. Enter *Heath Keene* at the prompt. Notice that only the first name *Heath* is stored in the string. The space between *Heath* and *Keene* caused the rest of the string to be ignored.
3. Close the source code file.

USING get

The **get** function allows a string containing spaces and almost any other character to be entered into a character array. Therefore, using the **get** function is a better way to get a string from the user. The string input ends when you press Enter. You get to specify the maximum number of characters in your statement. Consider the example below.

```
cin.get(student_name, 20);
```

The identifier **student_name** is the character array and 20 is the maximum allowed length. The maximum allowed length includes one character for the null terminator, which the **get** function adds to the end of the string. So if you need a string of 20 printable characters, you should declare your character array for 21 characters and set the maximum length in the **get** function call to 21.

FLUSHING THE INPUT STREAM

Statements like **cin >> first_name;** and **cin.get(name, 21);** pull characters out of the standard input stream and place them in a character array. In both cases, it is possible for characters to be left over in the input stream after the statement is complete.

For example, in the statement **cin >> first_name;**, if the user enters *Becky Dailey*, only the first name will be placed in the character array because the space between *Becky* and *Dailey* terminates the operation. The last name and carriage return remain in the input stream.

A similar problem occurs with the **get** function. If the user enters a string that is longer than the length specified in the call to the **get** function, the remaining characters are left in the input stream. Furthermore, the **get** function always leaves the new line character in the input stream.

The problem with leaving characters in the input stream is that the next statement that asks for input will receive the characters left over from the previous input statement. To avoid this, you can use the **ignore** function to flush the contents of the input stream (also called the buffer). Consider the statement below.

```
cin.ignore(80, '\n');
```

The 80 tells the program to ignore the next 80 characters in the stream. The **'\n'** tells the function to stop ignoring characters when it gets to a carriage return. You could use a number smaller than 80 in most cases. The function will usually stop ignoring after only a few characters, because it will find a carriage return.

Flush the contents of the buffer after a call to the **get** function because the **get** function always leaves a new line character in the stream.

EXERCISE 6-10 USING THE get FUNCTION

1. Enter the following program. Save the source code file as *INLINE.CPP*.

```
// INLINE.CPP
#include <iostream.h>

main()
{
  char name[21];

  cout << "Enter a name: ";
  cin.get(name, 21);
  cin.ignore(80, '\n');
  cout << "You entered " << name << '\n';
  return 0;
}
```

2. Compile and run the program. Enter *Heath Keene* at the prompt. Notice that the entire string is stored in the character array.
3. Close the source code file.

USING DESCRIPTIVE PROMPTS

When writing programs that interact with the user, be sure to output prompts that clearly explain the input the program is requesting. For example, if prompting the user for their name, use a descriptive prompt like the one below.

```
Please enter your last name:
```

If prompting for a telephone number or some other formatted data, you may want to use the prompt to give an example.

```
Please enter your phone number using the format (555) 555-5555:
```

The more descriptive and clear your prompts, the more likely the user is to enter the information in the form your program is expecting.

CLEARING THE SCREEN AND PRINTING A HARDCOPY

The techniques required to clear the screen and print to a printer vary, depending on the compiler and operating system you are using. Your compiler may have a function available for clearing the screen, or you may have to use another technique. Modern operating systems sometimes require special programming in order to send output to a printer. You may wish to send output to a text file on disk and then use a text editor to print the contents of the file.

On the Net

For more information about clearing the screen and printing, including sample code specific to your compiler, see http: //www.ProgramCPP.com. See topic 6.2.1.

SECTION 6.2 QUESTIONS

1. What is the purpose of **cout**?
2. What is another name for the input operator (>>)?
3. What is another name for the output operator (<<)?
4. What format option displays floating-point numbers in E-notation?
5. What I/O manipulator sets the number of digits to be displayed after the decimal point?
6. Write a statement that gets a number from the keyboard and stores it in a variable **i**.
7. Write a statement that outputs the result of the expression *PI * diameter*.
8. Write a statement that uses **setf** to display a leading plus sign before positive values.

9. Write a statement that outputs the variable **cost** with two decimal places.
10. Write a statement that inputs a line of text of maximum length 25 and stores it to a character array named **address**.

PROBLEM 6.2.1

Write a program that asks the user for the diameter of a circle and returns the circumference of the circle. First, store the user's input (diameter) in a floating point variable. Next, the program should calculate PI * diameter using a declared constant for PI of 3.14159. Output the result of PI * diameter in normal notation with a precision of four digits to the right of the decimal point. Save the source code file as *CIRCUMFR.CPP*.

KEY TERMS

array
character literal
console I/O
delimiter
extraction operator
I/O manipulators
insertion operator
null terminator
standard input device
standard output device
stream
string literal
whitespace

SUMMARY

- A string is a group of characters put together to make words or other text. A string in C++ can be any length and ends with a character called the null terminator.
- A string literal is any string of characters in a C++ program that appears between two quotation marks.
- Character arrays are used to store strings.
- Character arrays are declared by providing a name for the array and its length given in spaces.
- Character arrays can be initialized with a string when declared or a string can be copied to a character array after the declaration using **strcpy**.
- Because C++ allows you to store a string that is too long for an array, you must be careful when using strings.
- Streams as well as the << and >> operators are used for input and output in C++.
- Output can be formatted using **setf** and I/O manipulators.
- The input operator (>>) can be used to get more than one value in a statement and to input numbers or characters.
- If you need to input strings, you should use **get** rather than the >> operator.

PROJECTS

PROJECT 6-1

Write a program that asks the user for a name, address, city, state, ZIP code, and phone number and stores each in appropriate character arrays. Use the ignore function to flush the input stream between inputs. After the strings are stored in the arrays, print the information back to the screen in the following format:

Name

Address

City, State, ZIP Code

Phone Number

PROJECT 6-2

Write a program that asks the user for three floating-point numbers in a single statement. Print the numbers back to the screen with a precision of one decimal point. Use a field width for the output that places the three numbers across the screen as in the example below.

```
Input: 123.443   33.22   1.9

Output: 123.4      33.2      1.9
```

PROJECT 6-3

Write a program that asks the user for two floating-point numbers. The program should multiply the numbers together and print the product to the screen. Next, ask the user how many digits to display to the right of the decimal point and print the product again with the new precision.

PROJECT 6-4 • THE STOCK MARKET

Write a program takes in the name, opening value, closing value, and number of shares owned for a stock. Have the program print the stock name, opening value, closing value, and amount of value the stock gained or lost in a formatted line.

PROJECT 6-5 • CALCULATING COSTS

A photoshop frames pictures as part of its business. All frames are one inch wide. The shop charges 25 cents for each inch of frame and 10 cents for each square inch of matting material. Write a program that takes the length and width of the picture as input and calculates the cost of framing the picture.

PROJECT 6-6 • CALCULATING COSTS

Write a program that calculates the cost of painting a room. Ask the user for the number of square feet of surface to be painted, the number of square feet per gallon intended (typically varies between 200 and 400 square feet, depending on thickness of the coat), and the cost per gallon of paint. Calculate the number of gallons required and the cost, assuming that the paint must be purchased in even gallons.

Overview

In this case study, you will examine a program that analyzes the cost of an airline flight. The program asks for the number of passengers on the flight, the length of the flight, and the average ticket price. The program then calculates and outputs several values, including the time required for the flight, the cost of the flight, total fares collected from ticket sales, and the profit for the flight.

Obviously, there is much more to take into consideration when calculating the profit an airline makes on a specific flight. The program, however, demonstrates many of the topics covered in the previous chapters.

Let's begin by loading, compiling, and running the program. Then we'll analyze the source code.

EXERCISE

1. Retrieve the source code file *AIRLINE.CPP*.
2. Compile and run the program. The program is configured for the Boeing 747-400 jet.
3. Enter 300 for the number of passengers, 2500 for the length of the flight, and 349.25 as the average ticket price.
4. The program provides the following as output:

```
Analysis for Boeing 747-400

The flight will take approximately 4.69 hours.
The cost of the flight will be $32546.90, with a
cost per passenger of $108.49.
The total fares collected from ticket sales is $104775.00,
resulting in a profit of $72228.09.
```

5. Leave the source code file open.

Analyzing the Program

The complete source code for the program appears below. Spend a few minutes looking over the complete source code before reading the analysis that follows.

```
// Airline Flight Cost Analysis
// By Brian Davis and Todd Woolery

#include<iostream.h>   // necessary for input/output
#include<iomanip.h>    // necessary for setprecision manipulator
```

```
// main function
main ()
{
 // constants to set specifications for a Boeing 747-400
 // Source: The World Almanac and Book of Facts 1995
 char const plane_name[] = "Boeing 747-400";
 int const plane_speed = 533;
 int const number_of_seats = 398;
 int const max_flight_length = 4331;
 int const cost_per_hour = 6939;

 int num_pass;           // number of passengers on the plane
 float num_miles;        // flight distance
 float avg_ticket_price; // average ticket price for flight
 float flight_cost;      // cost for the flight
 float cost_per_pass;    // cost per passenger
 float total_fares;      // total fares collected for the flight
 float profit;           // profit for the flight
 float hours;            // length of flight in hours

 cout << "\nAIRLINE FLIGHT ANALYSIS\n";
 cout << "Airplane name: " << plane_name << endl;
 cout << "Enter the number of passengers on the flight (maximum "
      << number_of_seats << "): ";
 cin >> num_pass;
 cout << "Enter the distance (in miles) of the flight (maximum "
      << max_flight_length << "): ";
 cin >> num_miles;
 cout << "Enter the average ticket price: ";
 cin >> avg_ticket_price;

 hours = num_miles / plane_speed;  // calculate time required for flight
 flight_cost = hours * cost_per_hour; // calculate cost of flight
 cost_per_pass = flight_cost / num_pass; // calculate cost per passenger
 total_fares = num_pass * avg_ticket_price; // calculate total fares
 profit = total_fares - flight_cost; // calculate flight profit

 cout.setf(ios::showpoint);  // force decimal point to be displayed
 cout.setf(ios::fixed);      // prevent scientific notation
 cout << "\nAnalysis for " << plane_name << endl;
 cout << "\nThe flight will take approximately " << setprecision(2)
      << hours << " hours.\n";
 cout << "The cost of the flight will be $" << flight_cost
      << ", with a \n";
 cout << "cost per passenger of $" << cost_per_pass << ".\n";
 cout << "The total fares collected from ticket sales is $"
      << total_fares << ",\n";
 cout << "resulting in a profit of $" << profit << ".\n";
 return 0;
}
```

The program begins with two compiler directives that include **iostream.h** and **iomanip.h**. They are both necessary to make the input and output functions available.

Next, the main function begins. The remainder of the program is contained in the main function. Notice that the **#include** directives must appear before the main function.

The first set of statements in the main function (shown again below) define constants for a particular aircraft. Currently, the constants are based on a particular model of the Boeing 747. The plane name is stored in a constant character array. The brackets are left empty, which causes the compiler to allocate the appropriate array length based on the string being stored in the array. The other constants are integer values. These values will be used in calculations later in the program.

```
// constants to set specifications for a Boeing 747-400
// Source: The World Almanac and Book of Facts 1995
char const plane_name[] = "Boeing 747-400";
int const plane_speed = 533;
int const number_of_seats = 398;
int const max_flight_length = 4331;
int const cost_per_hour = 6939;
```

The next step is to declare variables. The variable that holds the number of passengers on the flight is declared as an integer; the others are floating-point variables. All of the constants and variables could have been declared as floating-point types to avoid the mixing of data types. Here, we will use integers where appropriate and allow the integer data types to be promoted during calculation.

```
int num_pass;           // number of passengers on the plane
float num_miles;        // flight distance
float avg_ticket_price; // average ticket price for flight
float flight_cost;      // cost for the flight
float cost_per_pass;    // cost per passenger
float total_fares;      // total fares collected for the flight
float profit;           // profit for the flight
float hours;            // length of flight in hours
```

You should always clarify the use for each variable using comments, as was done in the declarations above.

The next group of statements (shown below) gets the required input from the user. Remember, **\n** or **endl** can be used to force the cursor to the next line of the screen. To give the user more information about the range of possible input, the number of seats and maximum flight length for the particular aircraft are part of the prompt for the input.

```
cout << "\nAIRLINE FLIGHT ANALYSIS\n";
cout << "Airplane name: " << plane_name << endl;
cout << "Enter the number of passengers on the flight (maximum "
     << number_of_seats << "): ";
cin >> num_pass;
cout << "Enter the distance (in miles) of the flight (maximum "
     << max_flight_length << "): ";
cin >> num_miles;
cout << "Enter the average ticket price: ";
cin >> avg_ticket_price;
```

After the input is gathered, the calculations must be performed. First, the time required for the flight is calculated using the statement below.

```
hours = num_miles / plane_speed;  // calculate time required for flight
```

The constant **plane_speed** is an integer, but **num_miles** and **hours** are both floating-point variables. Therefore, **plane_speed** is promoted to a floating-point type for the calculation, and the result, also a floating-point value, is stored in **hours**.

A similar promotion of an integer type occurs in the other calculations that follow. In each case, the result is a floating-point value. As shown in the statements below, the cost of the flight is calculated by multiplying the time required by the cost of operating the aircraft for an hour. That flight cost is divided by the number of passengers to get a cost per passenger. The total number of dollars collected from ticket sales is estimated by multiplying the average ticket price by the number of passengers. Finally, the projected profit for the flight is calculated by subtracting the cost of the flight from the dollars collected from ticket sales.

```
flight_cost = hours * cost_per_hour; // calculate cost of flight
cost_per_pass = flight_cost / num_pass; // calculate cost per passenger
total_fares = num_pass * avg_ticket_price; // calculate total fares
profit = total_fares - flight_cost; // calculate flight profit
```

The only task remaining is printing the output to the screen. The statements below format the output in paragraph form. The first two statements below are necessary to have the numbers appear in the desired format. The **showpoint** format option causes the decimal point to be displayed, even if a non-fractional value is being printed. The **fixed** option prevents numbers from appearing in scientific or "E" notation. In the fourth statement, the **setprecision** manipulator is used to specify that only two digits should be displayed to the right of the decimal point.

```
cout.setf(ios::showpoint);  // force decimal point to be displayed
cout.setf(ios::fixed);      // prevent scientific notation
cout << "\nAnalysis for " << plane_name << endl;
cout << "\nThe flight will take approximately " << setprecision(2)
     << hours << " hours.\n";
cout << "The cost of the flight will be $" << flight_cost
     << ", with a \n";
cout << "cost per passenger of $" << cost_per_pass << ".\n";
cout << "The total fares collected from ticket sales is $"
     << total_fares << ",\n";
cout << "resulting in a profit of $" << profit << ".\n";
```

Modifying the Program

As an additional exercise, modify the program to analyze another aircraft. Choose one of the four airplanes in the following table. *Hint:* Only the constants need to be modified to change the aircraft for the analysis.

AIRCRAFT	SEATS	SPEED	FLIGHT LENGTH	COST PER HOUR
L-1011	288	496	1498	4564
DC-10-10	281	492	1493	4261
B737-500	113	408	532	1594
F-100	97	366	409	1681

Source: The World Almanac and Book of Facts 1995.

When you run the program with specifications from an airplane other than the 747-400, notice how the input prompts change to provide you with the allowable range of values.